# Studies in Computational Intelligence 415

For further volumes:
http://www.springer.com/series/7092

Francisco Fernández de Vega,
José Ignacio Hidalgo Pérez,
and Juan Lanchares (Eds.)

# Parallel Architectures and Bioinspired Algorithms

Springer

*Editors*
Francisco Fernández de Vega
Centro Universitario de Mérida
Universidad de Extremadura
Mérida
Spain

José Ignacio Hidalgo Pérez
Facultad de Informática
Universidad Complutense de Madrid
Calle del Profesor José García
Madrid
Spain

Juan Lanchares
Facultad de Informática
Universidad Complutense de Madrid
Calle del Profesor José García
Madrid
Spain

ISSN 1860-949X e-ISSN 1860-9503
ISBN 978-3-642-28788-6 e-ISBN 978-3-642-28789-3
DOI 10.1007/978-3-642-28789-3
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2012933085

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

# Contents

**Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
*Francisco Fernández de Vega, J. Ignacio Hidalgo, Juan Lanchares*

**Creating and Debugging Performance CUDA C** . . . . . . . . . . . . . . . . . . . . . 7
*W.B. Langdon*

**Optimizing Shape Design with Distributed Parallel Genetic Programming on GPUs** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
*Simon Harding, W. Banzhaf*

**Characterizing Fault-Tolerance in Evolutionary Algorithms** . . . . . . . . . . . 77
*Daniel Lombraña González, Juan Luis Jiménez Laredo, Francisco Fernández de Vega, Juan Julián Merelo Guervós*

**Comparison of Frameworks for Parallel Multiobjective Evolutionary Optimization in Dynamic Problems** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
*Mario Cámara, Julio Ortega, Francisco de Toro*

**An Empirical Study of Parallel and Distributed Particle Swarm Optimization** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
*Leonardo Vanneschi, Daniele Codecasa, Giancarlo Mauri*

**The Generalized Island Model** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151
*Dario Izzo, Marek Ruciński, Francesco Biscani*

**Evolutionary Associative Memories through Genetic Programming** . . . . 171
*Juan Villegas-Cortez, Gustavo Olague, Humberto Sossa, Carlos Avilés*

**Parallel Architectures for Improving the Performance of a GA Based Trading System** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 189
*Iván Contreras, J. Ignacio Hidalgo, Laura Nuñez-Letamendía, Yiyi Jiang*

**A Knowledge-Based Operator for a Genetic Algorithm which Optimizes the Distribution of Sparse Matrix Data** . . . . . . . . . . . . . . . . . . . 219
*Una-May O'Reilly, Nadya Bliss, Sanjeev Mohindra, Julie Mullen, Eric Robinson*

**Evolutive Approaches for Variable Selection Using a Non-parametric Noise Estimator** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 243
*Alberto Guillén, Dušan Sovilj, Mark van Heeswijk, Luis Javier Herrera, Amaury Lendasse, Héctor Pomares, Ignacio Rojas*

**A Chemical Evolutionary Mechanism for Instantiating Service-Based Applications** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 267
*Maurizio Giordano, Claudia Di Napoli*

**Author Index** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 287

# Introduction

Francisco Fernández de Vega, J. Ignacio Hidalgo, and Juan Lanchares

For many years, computer performance improvement was based on technological innovations that allowed to dramatically increase the chip's transistor count. Moreover, architectural progress aimed at organizing processors structure have allowed to overcome traditional sequential execution of programs by exploiting instruction level parallelism. Yet, last decade has shown that Moore's Law is reaching its natural breaking point and maintaining the performance improvement rate by decreasing transistor's size will no longer be possible. Main manufacturers have thus decided to offer more processor kernels in a single chip, opening the way to the multi core era. Examples of that are: the Intel core i3 (2 cores), i5 (4 cores), and i7 (4 cores) architectures, AMD Zambezi, phenom iii (8 cores), phenom ii (6 cores).

But this is not the only effort coming from the hardware industry. Another clear example are the Graphics Processing Units (GPUs). Initially conceived for speeding up image processing, those systems have become a standard for parallel execution of applications in the last few years.

On the other hand, the development of internet has leaded to the emergence of the Cloud concept and cloud computing technology, which provides distributed computing and storage resources to be effortlessly accessed through the web. A number of abilities must be considered when deploying cloud applications: remote computers availability, applications dependability and fault tolerance, to name but a few. Summarizing, the possibility of using parallel architectures is a common practice today, not only for the most complex systems but also when the simplest ones are deployed.

On the other hand, bioinspired algorithms are being also influenced by this paradigm shift: research is moving from sequential implementations to

Francisco Fernández de Vega
Universidad de Extremadura, Spain

J. Ignacio Hidalgo · Juan Lanchares
Universidad Complutense de Madrid, Spain

F.F. de Vega et al. (Eds.): Parallel Architectures & Bioinspired Algorithms, SCI 415, pp. 1–6.
springerlink.com

parallel and distributed versions of these typically population based techniques that inherently exploit the parallel nature of the underlying models. Although the benefit of using structured populations was foreseen by the pioneers, several decades have been necessary for the industry to provide accessible commodities for parallel and distributed implementations -including GPUs, multi and many cores, clouds, etc.- thus producing the grow of a trend in the field. The combination of Parallel Architectures and Bioinspired Algorithms is attracting an attention that will continue growing in the coming years.

We are thus editing this book with the goal of gathering examples of best practices when combining bioinspired algorithms with parallel architectures. Leading researchers in the field have contributed: some of the chapters summarize work that has been ongoing for several years, while others describe more recent exploratory work. The book thus offers a map with the main paths already explored and new ways towards the future.

We hope this volume will be of value to both specialists in Bioinspired Algorithms, Parallel and Distributed Computing, as well as computer science students trying to understand the present and the future of Parallel Architectures and Bioinspired Algorithms.

This book is a collective effort, and we must thank all the contributing authors, whose effort and dedication have given rise to the present work. We also thank institutions that have funded our effort, Extremadura Government and EDERF under project GR10029 and Spanish Ministry of Science and Techonology, project TIN2011-28627-C04-03.

Last but not least we appreciate the encouragement, support and patience offered by Professor Janusz Kacprzyk, as well as by Springer during the editing process.

## Road Map

This book is organized in chapters that shows some of the best know efforts for exploiting the parallel nature of Bioinspired algorithms in combination with parallel computer architectures. The chapters are logically grouped around a number of topics: hardware, algorithms and applications. Although no explicit sections have been established, readers can follow this path selecting those chapters that better fits with their interest. On the other hand, a sequential reading will provide a general view of the field going from hadware to software and applications. The reminder of this section includes brief summaries of each chapter.

**Chapter 1.** *Creating and Debugging Performance CUDA C* by W. B. Langdon

General Purpose computation on Graphic Hardware has attracted the attention of researchers that routinely apply Evolutionary Algorithms to hard real-life problems. The large number of processing cores included in standard GPUs allows us to obtain large speedups when parallel applications are run on them. Nevertheless, the new model requires extra skills from programmers. Although manufacturers provide frameworks and languages specifically devoted to program GPGPU applications, a number of issues must be considered for properly developing parallel EAs that profit from GPUs. This chapter presents various practical ways of testing, locating and removing bugs in parallel general-purpose computation on graphics hardware GPGPU applications, with attention to the relationship with stochastic bioinspired techniques, such as genetic programming. The author presents the experience on software engineering lessons learnt during CUDA C programming and ways to obtain high performance from nVidia GPU and Tesla cards including examples of both successful and less successful recent applications.

**Chapter 2.** *Optimizing Shape Design with Distributed Parallel Genetic Programming on GPUs* by Simon Harding, W. Banzhaf

This chapter applies a special version Cartesian Genetic Programming to optimize shape desing. Optimized shape design is used for such applications as wing design in aircraft, hull design in ships, and more generally rotor optimization in turbomachinery such as that of aircraft, ships, and wind turbines. By applying self-modifying Cartesian Genetic Programming (SMCGP) -which is well suited for distributed parallel systems, authors evolve shapes with specific criteria, such as minimized drag or maximized lift. GPUs are employed for fitness evaluation, using aN implementation of fluid dynamic solver.

**Chapter 3.** *Characterizing Fault-tolerance in Genetic Algorithms and programming* by D. Lombraña González, Juan L. Laredo , F. Fernández de Vega and J.J. Merelo

Genetic Algorithms (GAs) and Genetic Programming (GP) are a sub-class of Evolutionary Algorithms (EAs). In both classes, when the complexity is a key problem, a large amount of computing resources -and time- are required. In order to reduce execution time, both GAs and GP can benefit from parallel and distributed computing infrastructure. One of the most popular distributed infrastructure is the Desktop Grid System (DGS). The term desktop grid is used to refer to distributed networks of heterogeneous single systems that contribute idle processor cycles for computing. In DGSs, computers join the system, contribute some resources and leave it afterwards causing a collective effect known as churn. Churn is an inherent property of DGSs and has to be taken into account when designing applications, as these interruptions (computer powered off, busy CPUs, etc.) are interpreted by the application as a failure. To cope with failures, researchers have studied different mechanisms to circumvent them or restore the system once a failure occurs. These techniques are known as Fault-Tolerance mechanisms and enforce that an application behave in a well-defined manner when a failure occurs. This chapter is a summary of the obtained results for Parallel GAs and GP, presenting the study of fault-tolerance in PGAs and PGP in order to know if it is feasible to run them in parallel or distributed systems, without having to implement any fault tolerance mechanism.

**Chapter 4.** *Comparison of Frameworks for Parallel Multiobjective Evolutionary Optimization in Dynamic Problems* by Mario Cámara, Julio Ortega, Francisco de Toro

The main feature of Dynamic Multi-Objective optimization problems (DMO) is that the optimization is performed in dynamics environments so the cost function and constraints are time dependent. The main interest in this kind of problems is the wide range of real world applications with socio-economic relevance that have this feature. In this chapter the authors present two frameworks for Dynamic Multi Objective Evolutionary Algorithms (MOEA). The first is a generic master-worker framework called parallel dynamic MOEA (pdMOEA), that allows the execution of the distributed fitness computation model and the concurrent execution model. The second one, a fully distributed framework called pdMOEA+, is an improvement that avoid bottleneck caused by the master processor in pdMOEA . Both approaches have time constraints in order to reach the solutions. These frameworks are used to compare the performance of four algorithms: SFGA, SFGA2, SPEA2 and NSGA-II. The authors also propose a model to interpret the advantages of parallel processing in MOEA

**Chapter 5.** *An Empirical Study of Parallel and Distributed Particle Swarm Optimization* by Leonardo Vanneschi, Giancarlo Mauri and Daniele Codecasa.

Particle swarm optimization (PSO) is a bioinspired heuristic based on the social behavior of flocks of birds or shoals of fish. Among its features includes easy implementation, intrinsic parallelism and few parameters to adjust. This is the reason why in recent times the researchers are focusing their interest in these algorithms. In the chapter the authors present four parallel and distributed PSO methods that are variants of multi-swarm and attractive/repulsive PSO. Different features are added in order to study the algorithms performance. In the Particle Swarm Evolver (PSE) the authors use a genetic algorithm in which the individuals are swarms. Next the authors present the Repulsive PSE (RPSE) that added a repulsive factor. The third proposal is the Multi-warm PSO (MPSO) using an island model, where the swarms interact by means of particle migration at regular time steps. And finally, a variation of MPSO that incorporates a repulsive component named Multi-swarm Repulsive PSO (MRPSO). To study the different algorithms the author used a set of theoretical hand tailored test functions and five complex real-life applications showing that the best proposal is the MRSPO.

**Chapter 6.** *The generalized Island Model* by Dario Izzo and Marek Rucinski and Francesco Biscani

Authors introduce in this chapter the generalized island model, studying the effects on several well-known optimization metaheuristics: Differential Evolution, Genetic Algorithms, Harmony Search, Artificial Bee Colony, Particle Swarm Optimization and Simulated Annealing. A number of benchmark problems are employed to compare multi-start schemes with migration. An heterogeneous model is analyzed, which includes several "archipelagos" with different optimization algorithms on different islands.

**Chapter 7.** *Genetic Programming for the Evolution of Associative Memories* by J. Villegas-Cortez, G. Olague, H. Sossa, C. Avilés

Natural systems apply learning during the process of adaptation, as a way of developing strategies that help to succeed them in highly complex scenarios. In particular, it is said that the plans developed by natural systems are seen as a fundamental aspect in survival. Today, there is a huge interest in attempting to replicate some of their characteristics by imitating the processes of evolution and genetics in artificial systems using the very well-known ideas of evolutionary computing. For example, some models for learning adaptive process are based on the emulation of neural networks that are further evolved by the application of an evolutionary algorithm. This chapter presents the evolution of Associative Memories (AMs), which demonstrates useful for addressing learning tasks in pattern recognition problems. AMs can be considered as part of artificial neural networks (ANN) although their mathematical formulation allows to reach specific goals. A sequential implementation has been applied; nevertheless, the underlying coevolutionary approach will allow to easily benefit from parallel architectures, thus emulating natural parallel behavior of associative memories.

**Chapter 8.** *Parallel Architectures for Improving the Performance of a GA based trading System* by Ivan Contreras, J.Ignacio Hidalgo, Laura Nunez-Letamenda and Yiyi Jiang

The use of automatic trading systems are becoming more frequent, as they can reach a high potential for predicting market movements. The use of computer systems allows to manage a huge amount of data related to the factors that affect investment performance (macroeconomic variables, company information, industry indicators, market variables, etc.), while avoiding psychological reactions of traders when investing in financial markets. Movements in stock markets are continuous throughout each day, which requires trading systems must be supported by more powerful engines, since the amount of data to process grows, while the response time required to support operations is shortened. This chapter explains two parallel implementations of a trading system based on evolutionary computation: a Grid Volunteer System based on *BOINC* and an implementation using a Graphic Processing Unit (GPU) in combination with a CPU.

**Chapter 9.** *A Knowledge-Based Operator for a Genetic Algorithm which Optimizes the Distribution of Sparse Matrix Data* by Una-May O'Reilly, Nadya Bliss, Sanjeev Mohindra, Julie Mullen, Eric Robinson

A framework for optimizing the distributed performance of sparse matrix computations is presented in this chapter. An optimal distribution of operations accross the processor nodes is required. An intelligent operation-balancing mutation operator is applied to balance swaps data blocks between hogs and slackers to explore new balances. Authors study the performance of the algorithm introduced -HaSGA- when compared with a baseline GA. The HaSGA is itself a parallel algorithm that achieves approximate linear speedup on a large computing cluster. Network, memory, bandwidth and latency are parameters that have been taken into account.

**Chapter 10.** *Evolutive approaches for Variable Selection using a Non-parametric Noise Estimator* by A. Guillen, D. Sovilj, M. van Heeswijk, L.J. Herrera, A. Lendasse, H. Pomares and I. Rojas

This chapter considers the problem of designing models to approximate functions. The selection of an adequate set of variables heavily influences the results obtained: If the number of variables is high, the number of samples needed to design the model becomes too large and the interpretability of the model is lost. Authors present several methodologies -that apply parallel paradigms in different architectures- to perform variable selection using a non-parametric noise estimator to determine the quality of a subset of variables.

**Chapter 11.** *A chemical evolutionary mechanism for instantiating service-based applications* by M. Giordano and C. di Napoli

This chapter focuses on Service Oriented Architecture (SOA) -the *de facto* paradigm for the Internet of Services (IoS)-, i.e. a virtual space where information and content is stored, exchanged and manipulated by software and human entities through services. Compositions of services on demand in response to dynamic requirements and circumstances is required in this scenario, and the process of selection required service instances is modelled as an evolving chemical process that can react to environmental changes. The chemical metaphor allows to approach the composition of services as a decentralized and incremental aggregation mechanism governed by local rules such that environmental changes affecting any part of SBA may be processed at any time.

# Creating and Debugging Performance CUDA C

W.B. Langdon

**Abstract.** Various practical ways of testing, locating and removing bugs in parallel general-purpose computation on graphics hardware GPGPU applications are described. Some of these are generic whilst other relate directly to stochastic bioinspired techniques, such as genetic programming. We pass on software engineering lessons learnt during CUDA C programming and ways to obtain high performance from nVidia GPU and Tesla cards including examples of both successful and less successful recent applications.

**Keywords:** C programming, GPU, GPGPU, GPPPU, parallel computing, computer game hardware, graphics controller, parallel computing, rcs, randomised search.

## 1 Introduction

The absence of sustained increases in computer clock speed which characterised the second half of the twenty century is starting to force even consumer mass-market applications to consider parallel hardware. The availability of cheap high speed networks makes loosely linked CPUs, in either Beowulf, grid or cloud based clusters attractive. Even more so since they run operating systems and programming development environments which are familiar to most programmers. However their performance and cost advantages lie mostly in spreading overheads (e.g. space and power) across multiple CPUs. In contrast, in theory, a single high end graphics card (GPU) can provide similar computing power and indications are that GPU performance increases will continue to follow Moore's law [24] for some years. The competitive home computer games market has driven and paid for GPU development.

W.B. Langdon
CREST, Computer Science, Department of Computer Science,
University College London, Gower Street, London WC1E 6BT, UK
e-mail: W.Langdon@cs.ucl.ac.uk

F.F. de Vega et al. (Eds.): Parallel Architectures & Bioinspired Algorithms, SCI 415, pp. 7–50.
springerlink.com

For example, nVidia has sold hundreds of millions of CUDA compatible cards [8]. Engineers and scientists have taken advantage of this cheap and accessible computer power to run parallel computing. nVidia is now actively encouraging them by marketing GPUs dedicated to computation rather than graphics. Indeed the field of general purpose computation on graphics hardware GPGPU has been established [26].

The next section will give a brief summary of a few recent successful Bioinspired applications running on GPUs or nVidia's Tesla cards. Also, to illustrate there are pitfalls, we also include one less successful GPGPU application.

I shall assume the reader is already familiar with nVidia's parallel computing architecture, CUDA. Nonetheless Section 3 gives a quick introduction to it. Section 4 gives some ideas on how to produced reasonably fast GPGPU applications. In practice this always requires interaction between implementing "improvements" and measuring your software's performance to see if they really did have the desired effect (speeding up your code). Section 5 describes practical ways to measure performance.

There are many documents and tutorials on programming graphics hardware for general purpose computing. Mostly they are concerned with perfect high performance code. Most software engineering effort is not about writing code but about testing it, debugging it, etc., etc. Development of GPGPU software remains an art, often at the edge of feasibility. Testing and debugging are key to any software development but little has been published about getting non-trivial CUDA applications to work.

Although tools are improving, we concentrate upon how debugging is done for real. Many of the lessons are general. However the examples use nVidia's GPUs with their CUDA C compiler, nvcc, and some examples assume the reader is familiar with the Unix operating system. Section 6 describes coding techniques to aid debugging. Section 7 describes testing CUDA C applications, whilst Section 8 describes some bugs, the techniques used to find them and how they were fixed.

This is not a general tutorial on CUDA, however the last two sections give practical advice for when you get started (Section 9) and some ideas for where to look for help if you hit problems and discuss alternative approaches (Section 10).

## 2 GPGPU Bioinspired Algorithms

For a long time bioinspired algorithms were limited by the need to be sparing in their use of computer resources. As time has progressed computer power has increased enormously and so more and more realistic models of nature have been applied. Many of the natural phenomena which have inspired computer scientists concern multiple agents, each of which has to be simulated. For example, groups of nerve cells, swarms of insects, populations of plants or animals and diverse antibodies. Typically each agent is more-or-less independent and to some extent can be simulated independently of the others. At present each simulation is still often done one after another on a single computer. Since such simulations need a lot of computer time, this has tended to limit: the size of neural networks, the size of swarms, the number of simulated antibodies and the number of individuals in simulated populations. However in almost all cases, where parallel computers are available, the simulations can readily be run in parallel (rather than sequentially). The ease of with which this can be done has lead to many bioinspired algorithms being classified as "embarrassingly parallel" [23, p182]. Recently there has been considerable interest in using graphics hardware (GPUs) which readily provide cheap parallel hardware. Even a humble laptop can contain a low cost but powerful GPU.

Artificial neural networks come in a variety of flavours. We shall only discuss two. Perhaps the most realistic and hence the most computationally demanding are known as spiking neural networks. Whilst many flavours of ANN represent nerve cell activity as a continuous valued activation level, spiking networks represent nerve synapse activity as individual spikes. Given the computational complexity of even approximate chemical/electrical models of synapses, it is not surprising that the computational power of GPUs have been harnessed by several research teams. Yudanov *et al.* [36] showed fairly realistic (IZ) models of a few thousand neurons could be run in real-time by using CUDA and an nVidia GTX 260 GPU. A rather different approach is used by self organising maps (SOMs) or Kohonen networks. These can be thought of as unsupervised or clustering techniques which after multiple training periods learn to group similar concepts. Prabhu [28] used Microsoft's Accelerator GPGPU tool to get substantial speed increases from what is now modest hardware (an nVidia GeForce 6150 Go).

Some of the first uses of GPUs in evolutionary algorithms used them for graphics processing. This is closer to the original purpose of graphics hardware, nevertheless Ebner *et al.* [5] show genetic programming could evolve GPU code (vertex and pixel shaders written in Cg [6]) to generate images. However Fok *et al.* [7] were the first to implement a general purpose evolutionary algorithm on a GPU. They showed a complete evolutionary algorithm, including population mutation (but not crossover) and selection, as well as fitness evaluation running on an nVidia GeForce 6800 Ultra and obtained substantial speedups on a number of benchmarks with populations of several thousands. They also used the GPU to visualise their evolving populations. (Some animations of distributed genetic programming populations evolving under crossover and selection [21] can be found via http://www.cs.ucl.ac.uk/staff/W.Langdon/gp_on_gpu.html.) Harding was the first to show general purpose genetic programming running on GPUs [10]. Harding has considered a number of approaches however mostly he has required populations of GP individuals to be compiled [11]. Since the nVidia compiler is designed to optimise the speed of the GPU code it generates, rather than its own run time, it is often faster to interpret GP code rather than compile it [21]. Indeed the fastest single computer GP system uses a parallel GPU interpreter [17].

Bioinformatics contains many computationally demanding problems. Many of these are naturally parallel and so bioinformaticians are increasingly using GPUs. Restricting ourselves to bioinspired algorithms, there are several examples. For example in [19] we used an interpreted GP system built on RapidMind software running on an nVidia 8800 GTX to datamine human breast cancer biopsys to predict survival following surgery. Using a cascade of populations containing 5 million programs, a small intelligible model was distilled from noisy Affymetrix HG-U133A and HG-U133B GeneChip gene activity measurements. Whilst in [15] we used GP and public datasets to model factors influencing noise in the GeneChip's themselves. (In [18] we made a start at looking at automatic generation of GPU code.) Sinnott-Armstrong *et al.* have twice won the GPU competition at the GECCO conference for innovative uses of GPUs. In 2010 for a GPU based artificial immune system (AIS) [32] and in 2009 for epistasis analysis in human genetics. Their published work includes using three nVidia GeForce 295 (a total of 6 GPUs) to datamine a dataset of 547 people each having more than half a million genetic variations (SNPs). They were looking for gene-gene interactions to help treat sporadic amyotrophic lateral sclerosis (ALS) [9].

Rieffel *et al.* [31] showed an nVidia 9800GT could be used to evolve movement in a soft robot. The target pneumatic robot was simulated using PhysX. Such a soft bodied robot requires even more computational power than simulating a rigid robot. Realism was further enhanced by evolving a spiking neural network controller for the robot. As computer games continue to demand increased realism, dedicated "physics engines" (PPUs) will be used to offload from the CPU simulations of the physics of games, e.g. rock falls, in the same way that dedicated graphics processors (GPUs) are used now to offload graphics processing from the CPU. It is anticipated that PPUs will also contain substantial computing power and that this too will be used for algorithmic computing. Thus GPPPU will become popular in the same way that GPGPU has taken off.

Particle swarm optimisation (PSO) is a successful bioinspired algorithm in which a swarm moves under the influence of a fitness function. Mussi *et al.* [25] used a PSO to locate road signs in video images. With nVidia's CUDA they showed a swarm of particles was able to locate road signs in synthetic road images. A single GeForce 8800 GT GPU was powerful enough to run their PSO system at better than real-time (up to 150 video frames/second).

In ant colony optimisation (ACO) the swarm of flying insects is replaced by a colony of ants which navigate by following chemical trails left by other ants. There are various schemes so that successful ants guide the others but ACO explicitly includes the notion of forgetting as it requires the chemical to disperse over time. This ensures the ants do not get locked into the current best trail forever. The notion of exploiting (i.e. searching near the best solution found so far) versus exploring (searching more widely) comes up repeatedly (in different guises) in search and optimisation. Zhu and Curry [37] again used CUDA this time with a GeForce GTX 280 and show it considerably sped up their ACO on a wide range of continuous optimisation benchmarks.

GPUs have even been used to speed of simulations of artificial chemistries and regulatory networks [35].

While fuzzification is perhaps not normally thought of as "bioinspired", it too has substantial parallel components. Anderson *et al.* [12, 1] were the first to show fuzzy logic running substantially faster by running it in parallel on a GPU.

Although not a bioinspired approach, it is worth considering an unsuccessful approach. It is unclear exactly why [20] failed to achieve a big speed up. It may be that the underlying "close-by-one" FCA algorithm does not have sufficient arithmetic intensity (Section 4.1). Unlike the approaches described above, its inner loop only requires one Boolean logical operation per data item, whereas in the bioinspired approaches each data item may refer to an agent whose complete lifetime many have to be simulated. I.e. typically there is a huge volume of computing per data item. Thus even though the GPU beam search approach succeeded in parallelising the work over millions of threads this did not solve the problem that each data item had to be moved but only acted upon once. This in turn suggests, at least in this application, an arithmetic intensity of 1.0 is too low to make the GPU approach attractive. We now turn to the problems of actually getting code to work and getting the best from your parallel hardware.

## 3 CUDA – nVidia's Compute Unified Device Architecture

Although the reader will need to be familiar with nVidia's parallel computing architecture, we start with Figure 1 which shows how a CUDA application must make a trade off between the various storage areas, parallel computation threads and how having very many threads ready to run helps keep the many computation stream processors busy and the whole application efficient.

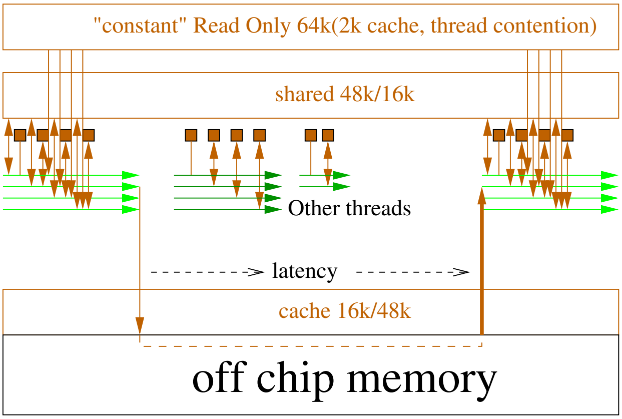


**Fig. 1** nVidia CUDA mega threading (Fermi, compute level 2.0 version). Each thread in a warp (32 threads) executes the same instruction. When a program branches, some threads advance and others are held. This is known as thread divergence. Later the other branches are run to catch up. Only the 32 768 registers per block (brown □) can be accessed at full processor speed. If threads in a warp are blocked waiting for off chip memory (i.e. local, global or texture memory) another warp of threads can be started. The examples assumes the requested data are not in a cache. Shared memory and cache can be traded, either 16 Kbytes or 48 Kbytes. Constant memory appears as up to 64 Kbytes via a series of small on chip caches [3], Section 8.4.

Figure 2 emphasises the need to divide the work between many threads. As expected performance rises more or less linearly as more threads are used. However notice that this continues even when the number of threads exceed the number of processing elements. While application and GPU specific, a rule of thumb suggests maximum performance needs at least 10 threads per stream processing core.

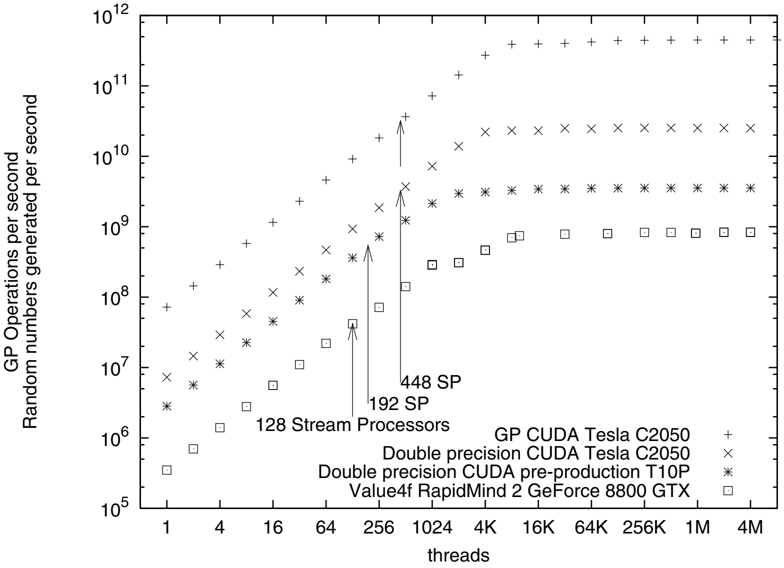


**Fig. 2** Speed of genetic programming interpreter [17] and Park-Miller random numbers [16] (excluding host-GPU transfer time) versus number of parallel threads used on a range of nVidia GPUs. Top 3 plots refer to CUDA implementations and lowest one to RapidMind code. Code available via ftp `cs.ucl.ac.uk` `/genetic/gp-code/`.

## 4 Performance

As novice programmers we were taught that we should get the code working before we worried about performance. However typically as CUDA developers we approach the code from the other direction. Typically there is a working serial version of the application which may need porting to CUDA. Ideally we should start by planning how the code will be run in parallel. This and the next section are about designing CUDA applications for performance, whilst Sections 4.2–4.4 deal with what happens when you try to run your initial design on your GPU and Section 5 describe some practical ways to locate and fix performance problems when pure design collides with real GPU hardware and software.

A high performance design will need to consider how many threads are to be used and how they are to be grouped into blocks. (A block of threads all execute the same kernel code on the same multiprocessor. They can pass data rapidly between themselves via shared memory, Section 6.6. High end GPUs typically have several multiprocessors, so multiple blocks of threads are needed to keep them all busy.) You will also need to consider where data will be stored, how much memory will they occupy and how and in what way memory will be accessed. In other words we should start by designing for performance. However coding a subroutine which runs on the GPU (known as a kernel) remains difficult and no software plan survives first contact with the GPU hardware. The alternative of developing prototype kernels has its attractions however getting a perfect prototype kernel is not necessarily a lot easier than coding the real kernel. In practice GPGPU software production tends to fall between the two. That is as problems arise, some can be fixed immediately, while others cause more drastic changes to the plan. These problems need not cause the wrong answer to be calculated but may be performance related or because, for a particular new work load, it is realised that some data will not fit into an available memory store. Since faulty kernels tend to give little indication of ultimate performance it becomes necessary to debug each new implementation of each new design. This is time consuming.

## *4.1 Performance by Design*

We have the usual problem that we do want to spend ages debugging a poor design and we do not know for sure how software will perform until we have written it. This section gives some rules of thumb to consider when designing your CUDA application. These might also be illuminating when trying to tune it.

- How much of your application can be run in parallel? If it it less than 90% then stop. Even if you are able to speed up the parallel part infinitely, so that it takes no time at all, you will still only increase the whole application ten fold. This is not worth your effort.
- In Bioinspired applications the resource consuming part is the fitness evaluation. Usually the fitness of each member of the population can be run independently in parallel and so fitness evaluation is an ideal candidate for parallel computation. This has been repeatedly recognised [30, 33, 4]. Indeed the comparative ease of parallelising population based algorithms has lead to them being called "embarrassingly parallel" [23, p182].
  Recall from Figure 2, CUDA applications typically need thousands of threads to get the best of GPUs. If your network or population does not contain thousands of cells or individuals, perhaps there are aspects of each individual fitness evaluation or learning which could be run in parallel? Obviously this is application specific.
- Estimate how much computation your application will need. Express this as a fraction of your GPU's performance. Is the fraction low enough to make the GPU a viable approach? Remember nVidia's performance figures are the best that the GPU can do and so are typically much more than your GPU kernel will get in practice.

- It is worth considering how much computation is needed per data item. I.e. the "arithmetic intensity". Often in Bioinspired algorithms we are concerned with computationally intensive tasks that most be done for every for every member of a network, swarm or population but only a few bytes are needed to represent the individual. Thus arithmetic intensity is usually high. However if only a few instructions are needed per word, arithmetic intensity should be considered carefully at the design stage. Effectively arithmetic intensity is another way of looking at the problem of communications bandwidth bottle necks.

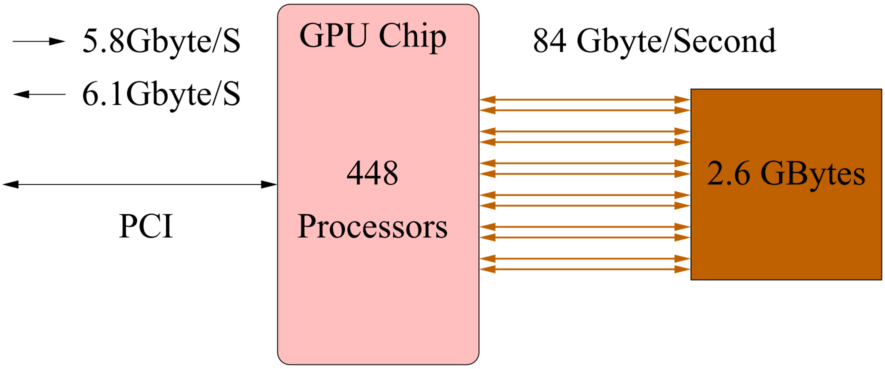


**Fig. 3** Links from GPU chip to host computer via PCIe bus and to memory on the GPU board. Fermi C2050.

- From your block level design locate its bottle neck. See Figure 3. We can try and find the limiting part of your design in advance of coding by estimating:
  1. The number of bytes of data uploaded into your GPU.
  2. The number of bytes from your GPU back to your PC.
  3. How many times the PC interacts with the GPU (either to transfer data or to start kernels).
  4. Do the same for global data flows from global memory into your kernel and from it back to global memory. Assume you are going to code your kernel so it uses registers rather than local memory.
  5. In principle we could consider other bottle necks but already we are getting into detail and relying on assumptions which may turn out to be wrong.

  For GPUs connected to a traditional PC via a PCIe bus we can get a good estimate of the time taken to transfer data across the PCIe by dividing the size of the data to be passed by the advertised speed of the bus. Take the lower estimate of your bus's speed and your GPU's PCI interface speed. Remember the speed into the GPU can be different from the speed back from it. If you already have the hardware, nVidia's bandwidthTest program will report the actual speeds. (bandwidthTest will also give you the maximum speed of transfers between global memory inside your GPU.)

For PCIe transfers, with good coding, the estimates can be accurate enough. With internal transfers so much will depend upon the details: how well the threads overlap computation with fetching data, how effective are the various caches.

- Normally the ratio of the volume of PCIe data size to the size of PCIe data buffers will give the number of times the operating system has to wake up your PC code so that it can transfer data. Typically there are a few data transfers before and after each time your GPU kernel software is launched. Usually the system overheads of rescheduling your process and CUDA starting your kernel are both well under a millisecond. Nonetheless if your design requires more than a thousand PCIe I/O operations or kernel launches per second it is probably worth considering the initiation overhead.
- This should have given you an idea of where the bottle neck is in your design and if your design is feasible.

  If the bottle neck is the GPU's computational speed, then it probably makes sense to proceed. It probably means your application is sufficiently compute intensive that it needs to be run in parallel. If it still not going to be fast enough then a redesign could consider a GPU upgrade, multiple GPUs and/or traditional code optimisation.

  If the bottle neck is bandwidth, which bus is limiting? Concentrate upon the most constricting part of the design. There are two things to consider: passing less through the bottle neck and making the bottle neck wider.
- In the case of the PCIe bus, only hardware upgrades can widen the bottle neck.

  Can you compress your data in some way? Often a huge fraction of computer data is zero. Do you need to pass so many zero's? Can you pack data more tightly? Can you use char rather than int? (Will the cost of compress/decompress be excessive?)

  Does your application need so much data to be passed? Could you pass some of it to the GPU once, when the application starts, and leave it on the the GPU to be reused, rather than being passed to the GPU each time the kernel is used?

  The host–GPU bottle neck can be critical to the whole GPU approach. The above calculations have the advantage of often being feasible to estimate in advance and typically applications really do get the host–GPU advertised bandwidth. So you can get good estimates of its impact on your application at the design stage. However the PCIe bus is inflexible. Unlike internal GPU buses, there is no coding to increase its bandwidth. If your design requires 110% of the PCI's bandwidth it is not going to get more than 100%. At this point many GPU designs fail and alternatives must be considered.
- As already mentioned with internal GPU transfers design stage calculations are much trickier. Perhaps consider algorithm or design level changes, e.g. splitting kernels, spreading the work differently across different kernels. Again can the bottle neck be made wider? E.g. by larger data transfers and/or coalesced transfers. Remember advertised figures and data reported by bandwidthTest have already taken into account such optimisations.

With the much lower bandwidth of PCIe it might make sense to reduce data transfer size by compression, e.g. using 8-bit bytes rather than 32 bits. This is probably not true within the GPU. Although the full range of C types are supported by the CUDA C/C++ compiler nvcc, the hardware works on multiples of 32-bits.

It is usually better to read data once, process it (without re-reading), then write the processed data once. Although nVidia's recent Fermi architecture caches local and global data and most GPUs cache textures such caches are quickly overwhelmed by the sheer volume of data to be processed. It is better to "cache at the design stage" rather than hope the data will still be in a cache if it is needed a second time. This is unlike traditional CPU coding, where it appears to cost nothing to read and write to program variables. On the CPU it is often better to calculate intermediate results, save them, then read them back and use them again. Whereas in a GPU it might be better to recalculate rather than save–re-read.

## *4.2 Performance by Hacking*

The previous section has talked about designing high performance GPGPU applications. Essentially the same basic idea applies whilst writing the GPGPU program code: Is performance good enough? Stop. Can performance be made good enough? If not then also stop. Identify and remove the bottle neck (e.g. by using the techniques to be described in Section 5). Before Section 5, the next section reminds us that it is not always necessary to implement everything the existing serial version does, whilst Section 4.4 considers how to include multiple GPUs into your design.

## *4.3 Performance by Omission*

Fundamentally the best way to improve performance is not by doing things better but by doing less.

The following need not be the best example but it is real. It turned out that about 30% of the time used by a kernel was spent looking for just one case in hundreds of thousands. It was not even a particularly interesting case and it was guaranteed to be found eventually. So a 30% speed up could be made by ignoring it. Further, once it was treated as impossible other parts of the kernel could be simplified giving a further speed up. By leaving out something unimportant to the users, the code went about twice as fast.

## *4.4 Multiple GPUs*

The processing power and capacity of single GPU cards continues to grow as new hardware is announced. However CUDA supports multiple GPUs per host PC and it may be attractive to use multiple cards. There are many twin GPU systems but three and even four card systems are also in use. (Be sure your host PC has sufficient power to support the additional hardware.)

To take advantage of multiple GPUs, parts of the host application must be run in parallel. That is the host programmer must explicitly organise the parallel operation of the PC's GPUs. CUDA does not (yet) allow you to launch a kernel across multiple GPUs or retrieve its results from multiple GPUs. Instead the programmer has to explicitly launch the kernel on each GPU. This is done in the same way as for one GPU but it does force explicit parallel multi-threaded code on the PC. Although CUDA provides some support for multi-threading of your PC code, it may be better to use your operating system's multi-threading support (e.g. the p-threads library). The standard advice is that your PC should have one CPU core per GPU card plugged into it. However the host multi-threading support should ensure 1) this is not absolutely necessary 2) your application will be able to take advance of dual or quad core CPUs without coding changes.

To avoid the surprisingly high CUDA initialisation overhead it is a good idea to start one host thread per GPU and repeatedly use it to pass data between the host and the thread's GPU and to launch kernels on its GPU. (I.e. the host threads live as long as your application itself.) Dual cards like the 295 GTX are programmed as two CUDA devices and so should have two threads (one each) in your host code. It is a good idea to record which devices your application is using.

```
cudaDeviceProp deviceProp;
cutilSafeCall( cudaSetDevice( dev ));
cutilSafeCall( cudaGetDeviceProperties(&deviceProp,0));
printf("Using CUDA device %d: \"%s\"\n",
       dev, deviceProp.name);
```

## 5 Measuring Performance

The main tool for measuring performance is the CUDA profiler (next section) but timing operations on the host (Section 5.2) yourself can also be useful. These give kernel level statistics but Section 5.3 will describe some ways to estimate the performance impact of program statements within your kernel. Obviously consider if there is a need for tuning and higher level aspects of tuning before getting sucked into the details (as described in Section 5.3).

### *5.1 CUDA Profiler*

nVidia's CUDA profiling tools can be downloaded from their web pages. As with other parts of CUDA, nVidia also freely provides downloadable documentation.

There are two parts to the CUDA performance profiler. The part on the GPU which records when certain operation took place. It logs the time of host-GPU data transfers and when kernel start and when they finish. It also counts other GPU operations. E.g. it can count the number of local or global memory cache hits and misses. Finally it transfers the logged data to the host PC. The second part runs on the PC. It can control the GPU based profile logging and also display both this data and previously logged data. Unfortunately certain Linux versions of this part (known was the CUDA visual profiler) are not stable.

As may be imagined the GPU part of the profiler is limited. Its job is to monitor performance not to interfere with it. Top end GPU contain several multiprocessors, since they are identical it is assumed their workloads and hence performance will be similar, therefore only one of them is monitored. Also the GPU profiler can gather a range of statistics but not all of them simultaneously. One of the main jobs of the visual profiler is to allow you to easily specify which data should be collected. (Different GPUs support different counters. Sometimes counters are not supported on a particular GPU because the counter was introduced to monitor a particular performance bottle neck which has been removed from the new GPU.)

If you specify more counters than the GPU can manage in one go, the visual profiler automatically runs your application multiple times collecting different profile data each time and then integrating them for you. Again the number of simultaneous counters depends on which type of GPU you are using. The visual profiler has the great advantage that it knows which GPUs support which counters and which can be simultaneously active. It also provides a wide range of plots and tables. A few of the interactive menus are a bit difficult to navigate and the documentation and menu layout may be slightly out of step.

When testing stochastic algorithms, such as Monte Carlo sampling or evolutionary computation, it is much easier if your code does exactly the same thing when run again. E.g. a genetic programming system should be coded so that its use of pseudo random numbers (PRNGs) can be controlled via the command line (see Section 7.1). By telling the visual profiler to pass the same PRNG initial seed to your GP when it runs it multiple times in order to collect a number of performance indicators, you should be able to ensure that these indications are consistent with those gathered by it on other runs.

Under the Linux operating system you can also control the GPU profiler directly by using environment variables, see Table 1.

The CUDA profiler gives some performance information which could be very useful but which would be either difficult or impossible to get elsewhere (e.g. cache hits). It also gives ready access to some critical information about the code that the compiler, nvcc, generated for your kernel. E.g. the number of registers the kernel needs.

If using CUDA_PROFILE_LOG directly, some counters become very large and difficult to comprehend. It would probably be worth using a spread sheet or simple script to rescale counters by the “instruction” count. (E.g. divide warp_serialize count by total number of instructions.) This helps make clear which data are

**Table 1** Unix environment variable controlling CUDA profiling

| Name: | Example | |
|---|---|---|
| CUDA_PROFILE | 1 | Switch on profiling |
| CUDA_PROFILE_CSV | 0 | Produce "comma separated values" suitable for importing into a spreadsheet or the CUDA visual profiler. With the value 0 a simple text file is produced. |
| CUDA_PROFILE_CONFIG | profile_r266a.txt | The name of a file containing instructions for the GPU profiler including which counters to enable. I suggest you start by copying CUDA_Profiler_3.0.txt from nVidia's web pages and then modifying it. |
| CUDA_PROFILE_LOG | profile_r266a.csv | The name of the profiler's output file. NB. the file will be overwritten if it already exists. |

important. Even if a counter has a five or six digit value, after it has been normalised by dividing by the instruction count it is clear which ratios are near zero and can be ignored.

Another useful measure is to calculate the number of "instructions" your kernel is executing per microsecond. The profiler is the only convenient route to these data. On a GTX 295, the profiler says a totally compute bound kernel will run in the region of 370 instructions per microsecond. Depending upon their "compute level" and because of the arcane way in which the profiler reports "instructions" other GPUs will each have their own value. (It is a useful exercise to construct your own compute bound kernel and see what figure your GPU gives.) Your application kernels will not reach the GPU's peak rate. If they are getting more than half the peak rate congratulate yourself and stop. I have had GTX 295 kernels as disastrously low as 5 instructions per microsecond.

## 5.2 *CUDA Timing Functions*

CUDA's timing functions can be used to time operations. They have the advantage of using the GPU's own high resolution clock but, as the following example shows, they tend to end up with voluminous code.

```
cutilCheckError(cutCreateTimer(&hTimer));

                    ⋮

cutilSafeCall( cudaThreadSynchronize() );
cutilCheckError( cutResetTimer(hTimer) );
cutilCheckError( cutStartTimer(hTimer) );
```

```
cutilSafeCall(
  cudaMemcpy(d_1D_in,In,In_size*sizeof(int),
             cudaMemcpyHostToDevice));

cutilSafeCall( cudaThreadSynchronize() );
cutilCheckError(cutStopTimer(hTimer));
const double gpuTimeUp = cutGetTimerValue(hTimer);
gpuTotal += gpuTimeUp;
```

As well as the reassurance of knowing what your code is doing, using the CUDA timing routines allows easy integration of timing information with the other data about your use of the GPU. However very similar timing information is available from the CUDA profiler without coding (Section 5.1). It is often convenient to create a CUDA timing data structure (`hTimer` in the above example) at the same time as you create your CUDA buffers (Section 6.1.3).

Notice some CUDA calls are asynchronous. Typically, this means, on the host they start a GPU operation and then return and allow the PC code to continue operation even though the GPU operation has only been started and will finish some time later. This allows 1) host PC and GPU operations to be overlapped and 2) the use of multiple GPUs on a single PC. However it does mean care is needed when timing operations on the PC, hence the heavy use of `cudaThreadSynchronize()` in the timing code. A common error is to omit calling `cudaThreadSynchronize()`. If it is not used `hTimer` typically gives the time taken to start an operation, e.g. the time taken to launch your kernel, rather than the time your kernel takes to run.

Except where multiple GPUs are to be used and assuming the GPU is doing the heavy computation, there is little advantage in allowing GPU and PC to operate asynchronously. This sort of parallelism is radically different from that provided by the CUDA and the GPU, it is just as error prone and hard to debug and typically offers only a modest performance advantage.

In production code you can use conditional compilation switches to disable `hTimer`. However, in practice (even when removing many `cudaThreadSynchronize()` calls) typically this will only make a marginal difference.

### *5.3 GPU Kernel Code Timing*

Although the GPU has on chip clocks, a useful approach is to add code to your kernel and see how much longer the kernel takes. This can be quite informative but needs to be done with care. Usually it is best to ensure the new code does not change subsequent operations in any way since their timing effects could totally cancel the timing effect of your new code.

Timing operation of the kernel from the PC is subject to noise from other activities on the PC. Random noise can be averaged out but it is better to ensure the timing effect that is being measured is much bigger than the noise. (E.g. perhaps do the additional operation a thousand times rather than just once.) When adding code you must remember that nvcc is an optimising compiler. In particular this means it will try to remove code that makes no difference to the kernel's outputs. To prevent nvcc optimising away the timing code we have just added, what is often done is to make the new code calculate a result and then use an "if" to ensure the result is discarded. Perhaps the if can depend upon one of the kernel's inputs, so that nvcc cannot easily reason about it, but we ensure that the if is always false. E.g. since `in_length` should never be negative, the code "`if(in_length<0) d_out=timing_info;`" will never be executed but nvcc does not know this and so cannot remove it and so it cannot remove the calculation of `timing_info` either.

This can be a useful way of confirming which parts of your kernel are expensive. However benefits can be disappointing. Kernels that are working well usually overlap reading from global memory with computation. So even large reductions in computation time can have little reduction in total time because the I/O time is unchanged. In the worse case, the more efficient coding simply increases the idle time waiting for data held in global memory to arrive.

Of course there is also always the dilution effect of Amdahl's law. In one example a function was made thirty times faster. However even the inefficient version of the function was responsible for only a small proportion of the total time. So vastly speeding it up made only an 11% change to the speed of the whole application.

# 6 GPU Debugging Techniques

## *6.1 Defensive Programming*

### 6.1.1 GPU Kernel Infinite Loops

The hardest problem to debug is probably when the kernel fails. Since CUDA GPUs do not have timeouts, this can mean the kernel never returns. It may lock the whole GPU. If you are using the same GPU to drive your computer's monitor, it will appear as if the whole computer has failed. It may require the computer to be restarted to reset the GPU. (Section 9.1 has some suggestions for reducing the impact of this.)

Notice not only is the result painful but you may get no indication of what has gone wrong or where. Further it is quite likely that it will happen again.

Given this is one of the worse bugs it is probably worth some defensive programming. A useful approach, particularly during development is to write a description of every kernel launch *before* it is started to a log file. (It may also be necessary to flush the log file before asking CUDA to start the kernel.) Conditional compilation switches could be used to remove it from production code. When a kernel fails, or is interrupted, the last thing in the log should give you an indication of where the error lies. I tend to write not just the kernel's name but also the thread grid dimensions, block size, number of bytes of shared memory requested and parameters to the kernel. In the case of arrays, I write the volume of the data in the array, rather than all it's values. This is probably unnecessary for most bugs but it is easier to be consistent and it is impossible to be sure in advance which information in the log will be useful.

```
printf("kernel_name<<<%d,%d,%d>>>(%d,%d,%d,<%d>,<%d>,<%d>:",
       grid_size, block_size, shared_size,
       height,width,len,
       len*sizeof(int),
       len*width*sizeof(unsigned int),
       len*sizeof(int));
printf("<%d>,<%d><%d>)\n", //outputs
       len*width*sizeof(unsigned int),
       len*width*sizeof(unsigned int),
       3*sizeof(int));

kernel_name<<<grid_size, block_size, shared_size>>>
(height,width,len,d_in,d_a,d_y,d_out1,d_out2,d_status);
cutilCheckMsg("kernel_name execution failed.\n");
```

Typically kernels have a main thread loop which allows you to change the block and/or grid size without recoding or recompiling but still ensures it steps through all of the input array. (See second example in this section.) Given the CUDA parallel processing architecture, it is seldom necessary to have other loops in kernel code. Similarly recursion is seldom used (in fact it is has only recently become possible). Thus it should not be too difficult to track all (potential) loops in your code and make absolutely sure that they terminate. A recent bug will show how this was used and proved very helpful.

```
int id   = -1;  //found it
int free = -1;  //free slot
int i    =  hash(value,Nvalue); //start search at i
int loop =  0;  //prevent looping forever
do {
  if(s_value[i]==value) {id  =i; break;}
  if(s_value[i]==    0) {free=i; break;}
  i++; if(i>=Nvalue) i=0; //Goto beginning of s_value
} while(loop++ < Nvalue);
if(id == -1 && free <0 ) Error(0x99960000,Nvalue);
}
```

The `do while` loop searches `s_value` for `value`. On successful exit `id` will indicate where it is. If it has not already been stored, `free` will say where it can be stored. hash() is only used to speed the search. If things were working as expected, the while loop could have been coded as while(1) However we know in advance the maximum number of times the loop should go round. (It is Nvalue, the size of the array s_value.) Therefore we can use while(loop++ < Nvalue) to force the loop to terminate, knowing it will catch indefinite loop errors but not abort the loop too soon. In fact the two break statements are the only legitimate ways of exiting the loop. An older programmer may have used goto, which might have simplified the last line.

The last line, checks the loop terminated as expected and if not reports an error. If, in some unexpected future run, we have more examples of value than we have space in s_value the error could arise legitimately. If we had not provided a check on loop++, this would cause the kernel to lock up the GPU and hence the monitor would freeze.

In an actual bug, hash() returned a very negative value. The search loop terminated and the problem was reported by via Error() on the last line.

The second example bug arose in the following loop structure which is based on CUDA's SDK examples:

```
int tid = MUL(blockDim.x, blockIdx.x) + threadIdx.x;
int threadN = MUL(blockDim.x, gridDim.x);
for(int i = tid; i < length; i += threadN) {
  ...
}
```

CUDA should provide legal values for blockDim.x, gridDim.x, threadIdx.x and so the loop *should* always terminate and so is commonly not protected. However in one kernel it was desired to dedicate different blocks of threads within the grid to different parts of the calculation and a bug was introduced when the second MUL was changed. This lead to threadN being set to zero and the kernel running until manually aborted. Of course, after the fact, it is also possible to add code to detect indefinite loop errors in this construct too. However, as errors here are not expected, it it seldom done.

### 6.1.2 CUDA Kernel Launch Failure

When launching a kernel always follow kernel_name<<<...>>> with cutil CheckMsg("kernel_name execution failed.\n"); This will ensure you know which was the first kernel to fail. Normally the string you supply to cutilCheckMsg() is fixed. However it need not be. If, for example, you start your kernel in a loop, you could use sprintf() to make the string you pass to cutil CheckMsg() include the loop index.

Since there is seldom a good reason for allowing the code to continue passed an error, you should wrap all host calls to CUDA routines with cutilSafeCall() or cutil CheckError(). See the examples in CUDA's SDK routines.

Sometimes the error message supplied by CUDA can be helpful but often it is very general. E.g. `cutilCheckMsg cudaThreadSynchronize error: kernel_name execution failed in file <kernel.cu>, line 1455 : unspecified launch failure`. This error message says there is an error somewhere. It is probably related to a particular kernel launch and the message tells you where in your source code to start looking. Sometimes starting your program via `cuda-memcheck --continue` can give additional information perhaps confirming the bug is an addressing error within the kernel.

The information you have written to the log file can sometimes be very helpful. For example did you tell CUDA to launch a kernel with zero threads per block? Was the grid size more than 65535? Or did you tell it to use more shared memory than the GPU has? Sometimes index array out of bounds errors inside the kernel can be reported as `unspecified launch failure`.

### 6.1.3 GPU Device Buffers

High end GPUs typically have a lot of high speed "graphics" memory. PCs with their lower performance typically have lower speed memory. Since it is cheaper, host computers typically have more memory than GPUs.

A good CUDA coding convention is to allocate a buffer in the PC's memory for each buffer in the GPU's global memory. The host and device buffers are of the same type and same size.

Given the high initial overhead on both starting kernels and transferring buffers, GPGPU applications tend to have a few large buffers. Even a complex application is unlikely to have more than a dozen PC/GPU buffer pairs.

It turns out that allocating CUDA device buffers has a very high overhead, so typically they and their shadow PC buffer are allocated once when the application starts and reused many times. I suggest you adopt a naming convention which makes it is obvious which buffers are on the CPU and which on the GPU and which shadows which.

Using `cudaMalloc` to create GPU global memory buffers:

```
cutilSafeCall( cudaMalloc(
     (void**)&d_buffer, buff_size*sizeof(int) ));
```

As with other C code, when debugging it is a very good idea to set all variables into a defined state before using them. In the case of GPU buffers this can be done with `cudaMemset()`:

```
cutilSafeCall( cudaMemset(
     d_buffer, 0, buff_size*sizeof(int) ));
```

`cudaMemset()` is fine for use whilst debugging. Often applications can be written which do not require large buffers to be cleared. However if yours does, it may be slightly more efficient to use GPU kernel code to initialise a large array in global memory, rather than to use `cudaMemset()`.

### 6.1.4 Host PC Buffers – Non-paged "Pinned Memory"

The host buffers on the PC can be created in the usual C or C++ ways however it is more efficient to ensure that they are locked into the PC's memory rather than being pageable. (This avoids the GPU driver copying the data twice.) Normally this effectively doubles the transfer speed to and from the GPU. However in one case, switching to non-paged memory gave a 27 fold speed up. `cudaMallocHost` provides a convenient way of allocating "pinned memory":

```
printf("Allocating non-paged host memory\n");
cutilSafeCall( cudaMallocHost(
    (void**)&Buffer, buff_size*sizeof(int) ));
```

Even though "pinned memory" is in host RAM, some versions of the GNU GDB debugger cannot access it. Instead it produces error messages confusingly similar to those it produces if you try and access the GPU's memory via GDB.

### 6.1.5 Debugging GPU Device Buffers

GPU device buffers are often huge, typically containing thousands or millions of data. Too many to check all individually. It is not always easy to construct small test examples which highlight particular bugs. Indeed the bug may only manifest itself with larger data sets.

Sometimes the GNU GDB debugger can deal with whole arrays. However the ability to interactively display arrays, even in an intelligible screen format, rapidly becomes less useful as the arrays get bigger. The CUDA programming style tends to mean pointers to buffers are passed around the code and (even without the problem of "pinned memory" mentioned in the previous section) GDB rapidly loses the sense of data as being an array and human access is only via pointers and offsets. Interactive access via pointers and offsets is tedious and hence error prone.

What has proved useful is creating a suite of host functions, one per data type, which simply dump an entire GPU buffer into a disk file in human readable format. (Depending upon your application, you may also want functions to load data from disk.) The files mean the whole of a buffer can be rapidly inspected by eye. They can also be subjected to semi-automatic sanity checks. Such checks might be informal or only true in particular circumstances. E.g. you might want to double check that there are exactly 273 non-zero elements in the buffer. It can be easier to apply such variable checks outside your application code.

Notice the following debug code does not use the host/GPU shadow but creates it's own dedicated buffer and reads from the GPU. The idea is to avoid cross talk between the debug code and the code being debugged. Also we avoid making assumptions about what we thought we had put into the GPU and instead read what is actually there.

```
if(debugging) {
  my_type* in = new my_type[size];
  cutilSafeCall(
    cudaMemcpy(in, d_in, size*sizeof(my_type),
               cudaMemcpyDeviceToHost) );
  print_my_type("In.txt",in,size);
  delete[] in;
}
```

Each of the print routines sends each datum to an output file one per line. The human readable format of each data item is as simple and as clear as possible.

```
void print_my_type(const char* fname,
                   const my_type buff[],
                   const int length) {
  FILE* ifd = open_(fname);
  for(int i=0;i<length;i++) {
    fprintf(ifd, "%8d %g\n",
            buff[i].timestamp,buff[i].pressure);
  }
  fclose(ifd);
}
```

The idea is to have a file for each GPU buffer. It may be that during a particular debug/test cycle not all of them will be needed.

When you have a working version of your application these files become valuable in their own right. The assumption is, since you know your application is working, then the contents of the GPU buffers and hence these files is also correct. Therefore when we produce a new version of the code (e.g. to tune it's performance or port it to different hardware) we can readily re-run the new code on the old input and use these files to confirm that the contents of the GPU buffers are the same as they were before.

The idea of `open_()` is to automatically give each file a name which depends on the version of the kernel we are running. `open_()` uses a `Version` macro containing the source file kernel.cu's version number: `#define Version "Revision: 1.266a "`. Thus GPU buffer `d_in` will be automatically saved in file `In.266a`

```
FILE* open_(const char* fin) {
  FILE* ifd;
  //replace fin type by Version
  char fname[80];
  char* p = strrchr(fin,'.');
  const int len = p-fin+1;   assert(len >0 && len <80);
  strncpy(fname,fin,len);
  char buf[80], buf2[80];
  strncpy(buf,Version,79);
  char* p2 = strrchr(buf,'.')+1;
  {const int len2 = p2-buf;  assert(len2>0 && len2<80);}
  strcpy(buf2,p2);
  char* p3 = strrchr(buf2,’ ’);
  {const int len3 = p3-buf2; assert(len3>0 && len3<80);}
  *p3 = ’\0’;
  strcpy(fname+len,buf2);

  ifd = fopen(fname, "w");
  if(ifd==NULL) {
    printf("Failed to fopen %s w\n",fname);
    exit(1);
  }
  printf("Printing to %s\n",fname);
  return ifd;
}
```

When either debugging or conducting regression tests there are at least two reasons why simple comparisons between two versions of a file might fail. 1) Your code has changed and the effect of the change on the GPU is being entirely correctly reflected by differences in the files. 2) The code is not deterministic but the details of it’s output (even when correct) depends upon the exact order in which parallel operations appear in the files. Thus running the program twice need not produce identical files (see Section 8.5). This makes the whole of testing and debugging much more complicated and so nondeterminism should be avoided. The increased possibility of creating a successful application may mean it is better to have deterministic code, even if it is slower.

If nondeterminism or potential future code changes mean that the order of data inside the GPU might change it is better to avoid saving line numbers, indexes, time stamps, etc., in the file. If blocks of data can legitimately move in the buffer, utilities like diff can often report this as a simple move of data about the file. Another approach is to sort the two files and then compare the sorted files. If data have simply been rearranged, the two sorted files will be identical.

It is now possible to generate your own debug text from inside your kernel using `printf()`, however my preference is still to use the “dump whole GPU buffer to disk file” approach. It is less intrusive to the code you are trying to debug and requires no change to the kernel code at all. Although, with very large files, it can have an impact on performance, the impact is readily isolated when inspecting either your own timing log or the CUDA profiler output. As mentioned above, it gives easy access to the whole of large data structures and typically integrates well with regression testing. With modest kernels, studying their source code, inputs and outputs is often sufficient to quickly locate problems. Perhaps as kernels grow in complexity, `printf()`’s ability to report kernel internals will be more important.

## 6.2 Debugging GPU Bioinspired Algorithms

Whilst some aspects of getting Bioinspired algorithms to work on GPUs will be specific to the algorithm many of the ideas I have described will also apply. However a particular class are the stochastic search algorithms, such as evolutionary computation. I have already mentioned the need to control their use of randomness (see Section 5.1, and also Section 7.1). Some algorithm specific techniques developed for serial versions can also be very useful when run on GPUs. For example in genetic programming [27, chpt. 13] it is recommended to test your implementation by seeding the population with one or more individuals of known fitness. E.g. create a GP seed individual which passes no tests or a "perfect" individual which passes all tests. Then verify that the fitness of these individuals, when calculated by your code on the GPU, is 0% and 100% respectively. (This is similar to the idea, described in Section 7.1, of testing by ensuring the GPU implementation gives the same results as a serial version of the algorithm.)

Another way of verifying your algorithm is to try it on a published "benchmark", e.g. 6-Mux [14] and ones max [29]. Your GPU code should give the same answer. With randomised algorithms like genetic algorithms (GAs) it will be necessary to do many runs and compare the mean number of fitness evaluations or other statistic to be sure that minor differences can really be put down to random chance fluctuations.

However this raises the awkward question of what to do if your GPU really does generate different answers. Unfortunately some benchmarks are not well described. This suggests you use well established simple benchmarks with published results from a range of authors. Also consider problems which you have worked on and have (debugged) serial code implementations. Can you compare your new GPU code against them? Of course for such comparisons to make sense, your GPU algorithm must be doing the same as the serial algorithm.

One oft repeated discovery is that GAs with distributed populations tend to do better than those with a single monolithic (panmictic) population. This is due the populations searching in different ways. For example, while it may make perfect sense for your GPU GA to contain several isolated populations each stored in isolated shared memory, we would not expect it to behave the same as a GA with the same combined population size, which every generation allowed complete mixing of all members of the population. These are different algorithms. One may work better than the other but it probably does not make sense to compare them when looking for bugs.

## 6.3 Your First GPU Kernel

The following way of debugging GPU kernels was suggested by Gernot Ziegler of nVidia. The idea is not to have the kernel do anything but simply prove to yourself that it can read it's inputs and send output to the right place.

When you come to debug more complex kernels, these steps may still be important. 1) Does the input data reach the kernel? This may be particularly important if the data were created by another kernel. 2) Does output leave the kernel? 3) Do the various threads put the data in the correct places? Are their values correct?

Lets start with a simple CUDA kernel which checks "does the GPU sends data to the right places?"

```
int tid = MUL(blockDim.x, blockIdx.x) + threadIdx.x;
int threadN = MUL(blockDim.x, gridDim.x);
for(unsigned int t = tid; t < LEN; t += threadN) {
  d_1D_out[t] = threadIdx.x;
}
```

You will need fair amount of code on the PC to support even this simple kernel. See the examples in the CUDA SDK sources directories. These directories include compilation command scripts. Remember to include code to check the kernel really is working. Once satisfied with your first kernel, inject a fault into it [22]. Did it fail in the way you expected? Did your error checking code catch the error, and handle it in an appropriate way? Did your revision control system (Section 7.3) allow you to recover your working version reliably and correctly?

Ok so now try both input and output. E.g. replace the contents of the loop with:

```
d_1D_out[t] = 1 + d_1D_in[t];
```

What values did you put in `d_1D_in`? Did you get the expected values in `d_1D_out`? Did you get the expected values in `d_1D_in`? How fast is it? How does the speed vary: if the arrays are bigger or smaller? if the arrays are types other than integer? what happens with different numbers of threads per block? (Remember Figure 1.) what happens if the grid size and dimensions are changed? what happens if adding one is replaced by a more demanding calculation? (Remember to check the answers the GPU gives.) What do you expect to happen if you run your kernel on a different GPU?

## *6.4 GPU Coding Style*

Often GPGPU applications contain only one or two kernels. Less than half a dozen is very typical. It is common to design them to be small (e.g. between 10 and 100 lines).

Earlier nVidia GPU's had quite limited numbers of registers, however current Fermi designs include 32 768 registers per multi-processor. The registers are used by every thread active on the multi-processor. Thus a kernel which used 100 registers could use at most 327 threads per block. With small purpose written kernels, the number of registers is no longer as big a factor as it used to be on earlier GPUs with fewer registers. However if large serial functions are converted into large kernels the number of registers could be an issue. The nvcc compiler has various options to allow fewer registers to be used and/or to "spill over" registers into local memory. However remember local memory is actually stored off-chip and even with caches it has the same performance impact as using global memory.

### 6.5 GPU __device__ functions

The CUDA C compiler, nvcc, efficiently supports functions on the GPU. Since nvcc inlines function calls, there is no overhead in calling them but the GPU code is not reduced by being able to use common subroutines to implement functionality needed in multiple places. Nonetheless nvcc implements them as full C functions and so one gets the normal development advantages of data scoping and variable arguments. Indeed there is no parameter passing overhead. Nonetheless one must always remember that the functions are to be run by many threads in parallel.

A coding problem, unique to parallel computing, is that the programmer must keep track of which threads are really going to execute the function. The following shows how this (and programmer error) created a bug.

Suppose a GPU function assembles an answer in GPU shared memory, it then wishes to send the answer to the host PC. It must first write it to global memory. In a kernel the following loop might be used.

```
for(int i=threadIdx.x; i<Nvalue; i+=blockDim.x) {
  d_out[i] = s_value[i]; //Bug
}
```

Notice how it spreads the work evenly amongst all the threads and allows the GPUs I/O hardware to efficiently bunch together large numbers of simultaneous writes into large low overhead blocks. Even access to the shared memory `s_value` avoids the overhead of bank conflicts. Unfortunately the code may be wrong.

Worse the error lies not in the code itself but in how it is used. Even worse the error may be very subtle, with almost all data correct and only incorrect every so often, depending on exactly what data the kernel is processing. Indeed if, e.g. for performance reasons, `d_out` is not reset between kernel invocations, it's last contents may be close to the values expected.

If instead of using a function, we had placed the above code inside the kernel itself we might have spotted the error immediately.

```
for(unsigned int t = tid; t < LEN; t += threadN){
  for(int i=threadIdx.x; i<Nvalue; i+=blockDim.x) {
    d_out[i] = s_value[i]; //Missing threads bug
  }
}
```

It starts to become clear that there is a relationship between the threads in the outer loop and those in the inner loop. The inner loop assumes all `blockDim.x` threads will run it. So does the outer one. However the problem arises, because once `t` reaches LEN the outer loop assumes it is done and effectively stops any remaining threads. Thus these threads are not available to the inner loop. This only happens in the last iteration of the outer loop, it only effects the highest numbered threads. At least it is deterministic but without studying the code and knowing the details of the parameters used to launch the kernel and the value LEN we do not know which threads will be affected. At the start of the kernel, both loops work well, however at the end some parts of `d_out` may not be updated and which ones depends on too many details.

Notice, to detect this error, it would be better to check the end of the buffer, rather than it's start. In fact the bug was picked up by noticing a regular pattern of zeros towards the end of the output file (generated using the debugging technique described in Sections 6.1.3–6.1.5).

This bug arises from parallel computing. In serial computing, once we have coded a subroutine and debugged it, we are now confident in it and only limited further checks are made. This is the case here. The code has been checked and when it is run it works. The problem arises because we think certain threads are going to run it but they do not. The code would have worked but it was never run.

To avoid the detailed consideration needed to ensure this bug does not happen, you should try to code `__device__` functions so as to avoid operations which need interaction between threads. This also has the advantage that `__syncthreads()` should not normally be needed in `__device__` functions.

## 6.6 nVidia GPU Shared Memory

GPU shared memory is rapid access read write on-chip memory available to blocks of kernel threads, see Figure 1. It gives CUDA it's only modifiable rapid access arrays. (Individual CUDA threads can have modifiable "local" arrays but until Fermi all local data was off chip and consequently slow. Fermi provides a cache which potentially makes read/write access to local arrays competitive with shared memory.) Shared memory can also be used as a very rapid way of passing data between computation threads in the same block. It cannot link threads from different blocks.

Shared memory is required for parallel computing "reduction" techniques (see SDK's `reduction_kernel.cu`). Whereby each thread calculates part of an answer but the whole answer is created by reducing these partial answers hierarchically into one (usually thread 0). It takes $\log_2 n$ steps to combine the answers of $n$ threads.

There is only a small amount of shared memory and it may be quickly be exhausted. CUDA's SDK has examples (e.g. `histogram`) where data are first stored in shared memory and then results from different blocks of threads are combined. SDK's matrix manipulation examples also make heavy use of shared memory.

In kernels where data are not processed independently shared memory can be a good place to store intermediate results. E.g. When scanning and removing duplicates from large arrays, multiple threads are needed to read the array rapidly but each thread needs to know which duplicates the others have found [20].

As with global, local and constant memory, there is a “best” way to arrange your threads when they access shared memory (which will of course be simultaneously). However unlike the other three types of memory the penalty for not using the best is slight and shared memory “bank conflicts” are seldom worth worrying about before the kernel is debugged. However as I have got a better understanding of how GPUs work, my kernels have used shared memory less.

It is often suggested that shared memory be used as a cache for your kernel. This can be a bit misleading. It is not worth using shared memory to buffer either input or output data whilst it is being read from or written to off chip memory. If you use `__syncthreads()` to ensure all data has arrived before you try and use them the GPU loses a large part of it’s ability to overlap I/O with computing and performance falls horribly. Each thread has a number of registers. Global data can be read/written directly to/from a thread register very efficiently without using shared memory.

### *6.7 Error Reporting*

The function Error(), mentioned in Section 6.1.1 (and in Section 8.9.2) was introduced into a kernel which was proving very hard going. It is designed to report the first error detected to the host PC, where code retrieves it and reports it to the log file. Given a parallel multithreaded environment, it need not always be clear which is the first error. The implementation of Error() does not try overly hard and the debugger must always be aware that events may be reported in an unexpected order. Even so Error() is probably more sophisticated than necessary for most kernels.

```
__device__ void Error(const int error,
                      const short int aux) {
  if(s_error==0) s_error = error | (aux & 0xffff);
}
```

In the main loop of the kernel we also have

```
if(s_error) {d_status[2] = s_error; break; }
```

Notice `d_status[2]` is shared between all the blocks of threads and so will suffer from “races”. We do not take special precautions about this since: it is code that should not normally be in use, the simpler it is the easier it will be to understand and the less likely it too will have bugs in. (Debugging debug code is especially annoying[1].) However `d_status` is an array of 32 bit values, so each 32-bit word will be self consistent. Often several blocks of threads will encounter errors and `d_status[2]` will contain the first error reported by the last block of threads. When using it to assist your debugging you may need to be aware that it was not necessarily the only error reported by your kernel. You will also need some code to transfer `d_status[2]` to the host PC and check it’s value:

[1] When you are in the swamp killing alligators, the thing to remember is that you are not supposed to be killing alligators; you are supposed to be draining the swamp.

```
cutilSafeCall(cudaMemset(d_status,0,3*sizeof(int) ));
                              ⋮

cutilSafeCall(
  cudaMemcpy(Status, d_status, 3*sizeof(int),
             cudaMemcpyDeviceToHost)
);
if(Status[2]) {
  printf("ERROR reported by kernel 0x%x\n",Status[2]);
  exit(99);
}
```

In the production code it is tempting to remove Error() or similar sanity checking code (such as assert in the host code). I suggest you do not remove it from the source code. In code that is in use, there will always be another bug and what you have already developed might help you or the next programmer find it. Again conditional compilation might be a good way to disable it. However in one complex kernel, commenting out "unneeded" sanity checking code saved only 6% of it's overall execution time.

## 7 Testing Parallel Software

Assume new or modified code is wrong. This is particularly important with stochastic bioinspired techniques. Guided by a fitness function, there are many occasions where evolution has worked around horrendous implementation bugs. From an application point of view, this is of course a strength. If the genetic algorithm came up with a good solution, we do not care the implementation was poor. Indeed it might be argued buggy GAs are considerably cheaper to implement than perfect ones. From a scientific point of view this is less satisfactory.

If we are researching an improved stochastic search operator (e.g. a new GA crossover operator) for a particular application domain, we want to be sure that any differences are really due to the crossover operator and not due to bugs in either our GA or in the GA we are comparing against. The fact that a good solution was found, does not mean the GA code we used did what we thought it did.

### *7.1 Comparison with a "Gold Standard"*

Many of nVidia's SDK examples, not only show how to code an example in CUDA but also include comparing the GPU's results with a traditional implementation of the example. Can you do the same? Do you have a convenient solution to your problem (which you are confident is correct)? Can you knock up a simple (even inefficient) conventional version? This need not even be written in C, perhaps python, gawk or spreadsheet, as long as it produces correct (but non-trivial) answers. (Nonetheless remember to use your revision control system, Section 7.3.)

It is much easier to compare results if your CUDA code produces identical results to your gold standard. Insist on it. Once you get into heavy coding it is easy to assume small differences are unimportant and as data volumes ramp up larger differences can be overlooked in a mass of minor ones.

With stochastic methods use (at least during testing) deterministic sources of random numbers (PRNGs), e.g. [16]. Keep a record of the seeds used in the log file. Perhaps use the same seeds with the GPU and your gold standard code. (Do not use these seeds during production runs).

With floating point numbers the GPU will produce different answers. Decide in advance how big a difference you expect. When comparing PC and GPU results, use an automated method which will only show you unexpected differences. Consider if you should include -0, NaN, etc., as different.

## *7.2 Regression Testing*

Be sparing in your inclusion and careful in the placement of: version numbers, date stamps and elapse times in output files. Even in correct code, these will be reported as different and you can quickly be swamped by uninteresting differences, which may (particularly if mixed with other data) conceal important differences.

## *7.3 Software Version Control*

You will create multiple version of your source code. At some point you will insert a fault into it and want to revert to an earlier version. You will want to be able to compare different versions. You should start using a convenient version control system when you start coding.

Having said that the best way to use it will depend on you. It is easy to delay saving a version whilst coding/debugging is going well and then find at the end of the day (usually when tired) that an error has been made and you do not want to throw away all the nice code written since the last time you checked kernel.cu into your revision control system (rcs) before the error was made. However you did not spot the error as it was made and either your editor will not allow you to undo the changes or you need to undo so many individual character changes that that it itself becomes tedious and error prone. On the other hand it is possible to check-in source code too often so the rcs history log becomes a sequence of meaningless messages of the type “changed function xxx: still not working”. My preference is for too often. After all saving a revision will take less time than compiling it.

# 8 GPU Bugs

The following sections described a few examples of real GPGPU bugs, how they were found and how they were fixed.

## *8.1 Not All Threads Available*

Another manifestation of the problem described in Section 6.5 occurred when a function was called inside conditional code within the main kernel.

```
if(data) {
  ... lookup data ...
  if(missing) save_data(data,...);
}
```

It is obvious from this code that only certain threads (those for which `data` is both non-zero and has not already been saved) will call save_data(). However this is not so clear when studying, as one is trained to do, save_data() in isolation.

Initially there were other problems with save_data() and this bug mearly added to the confusion. For performance reasons, save_data() was redesigned several times and eventually detailed knowledge of how it handled threads in different warps was used to implement it efficiently.

Large volumes of test data were passed through the kernel both to soak test it and to give reasonable estimates of how it will perform for real. The soak test gives some reassurance that the heavily inspected code really can cope with all combinations of simultaneous arrival of identical and non-identical data.

## *8.2 nVidia GPU Shared Memory Bug*

The optional third parameter in nvcc's <<< >>> CUDA kernel launch syntax allows you to specify the number of bytes of shared memory available to each block of threads in the kernel. The nVidia CUDA C programming guide says how to write your kernel. Unfortunately it is complicated and, as we shall see, error prone.

`kernel<<<grid_size,block_size,shared_size>>>(...)` effectively gives the kernel an anonymous array[2] which the kernel (with the compiler's help) has to convert into usable C variables. I have evolved the following (which is based on the CUDA C programming guide).

There is one shared array. (It appears that if you try and declare two, they will actually be placed on top of each other.) It is declared in your .cu file using `extern __shared__ unsigned int shared_array[];` Every shared

[2] Anyone else old enough to remember Fortran unnamed common blocks? They were also a bug waiting to happen.

variable is explicitly defined as an offset from the start of it. You should provide host based checks (cf. `shared_size`) that these do not run off the top of shared memory. CUDA will check at run time you have not asked for more shared memory than your GPU has.

For every kernel that uses shared memory, we define macros like set_shared. Each such macro is used in the scope of it's kernel and/or the kernel's `__device__` functions.

```
#define set_shared                                        \
  volatile int* xs_error =  (int*) &shared_array[0]; \
  volatile int* xs_ndata =  (int*) &shared_array[1]; \
  volatile unsigned int* s_data =  &shared_array[2]; \
  volatile int* s_ptr =     (int*) &s_data[Nvalue]

#define shared_size ((3+2*Nvalue)*sizeof(int))

#define s_error  xs_error[0]
#define s_ndata  xs_ndata[0]

                              ⋮

__device__ void Error(...) {
  set_shared;
  if(s_error==0) s_error = ...
}
```

The additional macros s_error and s_ndata allow the kernel code to treat them as scalars rather than arrays. Notice array s_ptr should lie after s_data and none of the data should overlap. The particular bug arose as a cut and paste error whereby instead of using starting s_ptr at the last plus one element of s_data (i.e. s_data[Nvalue]) another value was used. Nvalue is a const int set to 800. The wrongly used variable was set to 600. Hence a quarter of the two arrays overlapped. This meant the code worked on some small examples but failed horribly on others. The device buffers described in Sections 6.1.3–6.1.5 and regression testing were used whilst finding and fixing this bug. However knowing which parts of the source code had been recently changed lead quickly to the location of the problem.

## *8.3 nvcc C++ Compiler Volatile Keyword*

I tend to avoid exotic parts of programming languages and so had overlooked nvcc's use of volatile when declaring shared memory variables. volatile essentially turns off nvcc's optimisations whereby it uses registers rather than direct access to shared memory. Normally I would simply let the compiler get on with generating code but here was a bug in the making. Shared memory was deliberately used by multiple threads. When multiple threads of the same warp write to the same shared data, the hardware ensures one of them succeeds and the data from the others is discarded.

When nvcc optimises code which does not use volatile it may replace an access to shared memory by using a thread register. This lead my C code to think all the threads had succeeded in writing. Now that I realise what can happen, I use volatile on all shared memory declarations. The performance penalty of accessing shared memory rather than a register is small and I have not yet found an example where I am sure it is safe to allow the compiler to prevent inter-thread communication. After all I am mostly using shared memory to communicate between threads.

## *8.4 nVidia GPU Constant Memory*

I going to call this a bug because even though the correct answers were calculated: in supercomputing we don't just want the correct answers but we want them fast, and this wasn't.

At first sight constant memory (Figure 1) appears attractive. Often applications have important data that we know is not going to change. Sometimes it looks small enough that it will fit into 64Kbytes. Or perhaps it is sparse and we can compress it into 64K. Essentially it can be much faster than global memory but it is not really 64Kbytes but a 64K window onto a much smaller caching system [34]. One view is to use textures instead since these are cached. Another possibility might be to take advantage of Fermi's cache and assume it will have the kernel's (read only) data in it most of the time that it is needed.

Here is my view of how constant memory works. Each kernel has a 64Kbytes window onto the same patch of regular global memory. Only the host PC is allowed to update that window but it can do it multiple times. Each time a thread tries to read from constant memory, the read request works it's way up through a hierarchy of caches. I am sure the details will vary between GPU architectures but Wong *et al.* [34] suggests the closest and hence fastest cache has only space for 512 integers or floats. (They say the largest useful cache has space for 2048.) Hence, we might think, if each thread block uses somewhat less than 8 KB (ideally less than 2 KB) there is a reasonable chance constant memory will help. Now it might be that we manage our kernel so a different thread block reads a different 8 KB, so it may be we can actually efficiently use all 64 KB if we are lucky (or skillful) with the details of how we write our kernel's reading of `__constant__` data. (Have I put you off yet? It gets worse.)

The hardware restrictions mean only one word can be read at a time from the `__constant__` cache. So if you code your kernel so that all threads in a warp read the same datum at the same time all is well. If they read two data, even if both are in the `__constant__` cache, the hardware will stall some of the threads and the whole read will take twice as long. In the worse case, where each thread accesses it's own datum, the read takes 32 times as long. So while we have an advertised 64 KB, this is actually something like 512 words of real fast memory and then we can efficiently only read one of them!

The CUDA profiler turned out to be very useful. It can display the compute level 1.x GPU counter warp_serialise. In one case warp_serialise was huge, about 23 times the instruction count. This required the whole application to be redesigned. Essentially random access was replaced by a system where each block of threads uses only a limited part of the 64K and usually threads in the same warp read the same elements of the array at the same time. warp_serialise fell to an average of less then 1% across the kernel and the kernel at last started to run at a reasonable speed.

The following two code snippets declare and set constant memory. They are in the same .cu file and so are compiled in one go by nvcc.

```
__constant__
unsigned int Constant[15*1024]; //Leave 1kw free
```

The host PC code uses `cudaMemcpyToSymbol()` to initialise `Constant[]`. `cudaMemcpyToSymbol()` can also be used to change `Constant[]` between kernel executions. Placing it in a host function allows the GPU `Constant` array to be changed anywhere in the host PC code.

```
assert(0<matrix_size &&
       matrix_size<=15*1024*sizeof(unsigned int));
cutilSafeCall(
  cudaMemcpyToSymbol((const char*)Constant,matrixw,
    matrix_size, 0, cudaMemcpyHostToDevice) );
```

In principle the compiler can use the C `const` quantifier to recognise read only inputs to your kernel and access them via constant memory. In practise I have not seen it do this. At present it appears that only GPU data you explicitly denote with `__constant__` is accessed via constant memory.

nvcc uses `.const` in the ptx assembler it generates to indicate `__constant__` memory. (See the nvcc `-keep` command line option.) Note although the compiler generates human readable assembler, inspecting it is very rarely helpful.

### *8.5 Non-reproducible Parallel Bugs*

In industry it can be standard practise to ignore non reproducible bugs. They are hard to find and hard to fix. And besides there are plenty of well behaved bugs to fix. In parallel code the fact that it behaves differently in different circumstances can give you a clue that it suffers from some race condition. It may be that an asynchronous update problem has been in your code sometime but is only exposed by a change in the way it used. For example running on a different GPU or a change in load within the kernel or a change in the way it uses threads, particularly increasing the number of threads above 32.

## 8.6 Impossible Bugs

Sometimes is just impossible to see why something does not work. It may be this is an opportunity to re-read the relevant CUDA documentation, find examples which do work, or consult the various online discussion groups, e.g. the nVidia CUDA Programming and Development forum. However perhaps you should use your revision control system (Section 7.3) to rewind your source code back to some earlier stable version.

Is it absolutely essential you implement the feature in the buggy kernel code? If so, is the bug related to the parallel threads? Perhaps it would be sufficient to have a serial version?

The following example shows using thread zero to force what should have been done in parallel to be done in series. (Remember the warning in Section 6.5 that thread zero must actually execute your serial code.)

```
//ugly hack
if(threadIdx.x==0) {
  s_ndata = 0; //Number of non-zero elements in s_data
  for(int i=0; i<Nvalue; i++) {
    if(s_data[i]) s_ndata++;
  }
}
__syncthreads();
```

## 8.7 Difficult Code

Perhaps if you suspect something is going to be hard you should consider writing a prototype first. The idea is the prototype should the opposite of CUDA. It need not be fast, it should not be run in parallel and it should be easy to implement. I tend to use gawk scripts because they handle reading input files much better than C. But it needs to be something you are comfortable with programming. Hack about your prototype until you have worked out the transformation you want the kernel code to do and the algorithm whereby it should do it. Kernels do not take kindly to being hacked. It should be much easier to work through your ideas in simple serial host PC code.

The GPU buffer files described in Section 6.1.3 might be quite a useful source of test data for your prototype. Ensure at least the “final” version of your prototype and any scripts/command lines needed to run it are saved in your revision control system (Section 7.3) before you go back to coding your kernel.

## 8.8 CUDA Bugs

Very rarely you will come across bugs in nvcc. Old versions of nvcc are not going to be fixed. If you have the very newest nvcc, you can report the problem. In all cases you will have to work around the problem.

Some advocate the C++ Standard Template Library (STL), but C++ templates have caused compiler bugs in the past.

## 8.9 C Coding Bugs

### 8.9.1 Loop++

This was a logic error and not particularly related to CUDA or parallel computing. I had provided hash() to speed up searches. The monitoring code suggested a huge problem with many more hash clashes than searches. Inspecting the kernel code suggested the problem lay here:

```
int loop =  0;
do {
  if found .... break;
  else ... continue to search ...
} while(loop++ < large limit) //avoid infinite loop
if(loop) report long search
```

If hashing was working well, in almost all cases the `loop` should be exited before the `while` statement but in many cases `loop` was not zero and a hash clash was being reported. The wrong fix was applied. "Obviously" `loop++` had incremented loop from 0 to 1, so the last line should have been checking `if(loop>1)` not `if(loop)`. This was wrong (and did not resolve the problem). It turned out the hashing algorithm was flawed, resulting in a hash clash in many cases causing the `while` loop to be reached and `loop` to be correctly incremented and a hash clash to be correctly reported. Eventually hash() was improved and the number of hash clashes reported fell dramatically.

Part of the reason for the misdiagnosis was the delay between when I had first (correctly) written the loop and the availability of hash() and so the ability to test the loop. In the intervening period I had forgotten the logic of how `while(loop++` was expected to work. Better comments in the source code might have helped.

### 8.9.2 C Shift Operations and `unsigned int`

Given the dire warnings about the computational expense of division on GPUs and for other "efficiency" reasons the use of left shift << and right shift >> is common place. It is easy to overlook the warning in [13, p49] which says >> on an `int` can either fill with copies of the sign bit ("arithmetic shift") or with zeros ("logical shift") depending on the hardware. This gave rise to the bug mentioned in Section 6.1.1. For example when kernel local variable `int v` has the value 0x80000000 and it is right shifted 24 instead of getting 0x00000080 (128), `v >> 24` gives 0xffffff80 (-127). Once found, this is readily fixed by declaring `v` as `unsigned int`.

It is claimed that the CUDA optimising compiler, nvcc, will spot division by integer powers of two and replace them by the correct shift operation. So it is common to use `/32` rather than `>>5` and rely on nvcc to create efficient code.

Although `Error(0x99960000,Nvalue)` quickly trapped the error, it was actually localised by remembering that the nearby hash() function had been recently changed and then asking the rhetorical question "how could hash() generate unexpected values".

hash() is expected to return a value between 0 and `Nvalue-1`, so conditional code was added to report if hash() returned something outside this range. E.g. `if(i<0 || i>=Nvalue) Error(0x999a0000,i);` Once this confirmed hash() was misbehaving (probably producing negative values) `if.. Error` could be used to further localise the bug but fundamentally hash() is short enough for the unexpected source of negative integers to be traced to my wrong assumptions about `v >> 24` and the declaration of `v` to be corrected.

### 8.9.3 Defensive Coding and Conditional Compilation

Again this is a bug which should not have happened, nevertheless it gives an example of where defensive coding was helpful. I had changed my GP CUDA kernel so that it ran all possible test cases rather than just a sample. Thinking I would only want to use this in special cases I intended wrapping it in conditional compilation marks `#ifdef ALL20` unfortunately I placed the corresponding `#endif //ALL20` after the new code. Thus leaving the original code to be compiled regardless of whether `ALL20` was defined or not. This meant when `ALL20` was defined each individuals fitness was calculated using both all the tests and the original sample. It was thus quite possible to score more than 100%. Here the defensive coding came in.

When the GP had been ported to the GPU all values calculated by the GPU were regarded with suspicion. In particular there was an `assert` which checked for both negative fitness values and values above 100%. After the GP had been debugged "obviously" the check was no longer needed, however fortunately I had not removed it. When the "improved" kernel was run, the check was quickly triggered and the error reported. The bug was easily located using the revision control system (Section 7.3) to highlight code that I had recently changed.

### 8.9.4 Graphics Card Hardware Monitoring

When debugging is hard it is always tempting to look for hardware problems. The nvidia-smi program can be used. E.g. `nvidia-smi -a` will tell you which GPUs you have and their temperature. However this is seldom useful.

Sometimes, e.g. when not using X-11, the nVidia GPU driver can unload software when the GPU is not in use. It can take several seconds, particularly if you have multiple GPUs, to reload it. This can delay the start of some GPU tools. A hack to avoid the driver thinking the GPUs are not in use, is to run nvidia-smi continuously in the background. E.g: `nvidia-smi -l -i 10 > /dev/null &` keeps nvidia-smi looping every ten seconds but discards its output.

`lspci` can be useful during installation for confirming you have the GPUs you expected plugged into the computer you expected them to be in.

# 9 GPGPU Development Environment

Section 9.1 suggest ways of setting up your system to ease development. Sections 9.2 and 9.3 described compiling your code whilst Section 9.4 discusses common configuration problems.

## *9.1 Hardware Environment*

As mentioned in Section 6.1.1, the most disruptive problem to debug is probably when the kernel locks up your computer. There are a range of ways to set up your GPU programming system to ameliorate this:

- Test kernels on a dedicated computer.
- Have the test computer and GPU physically adjacent to your desk.
- Have multiple GPUs in the computer. E.g. a small cheap one that only drives the monitor and one or more GPU that are used for kernel development. It may be there is already a GPU on your PC's motherboard which was disabled when the development GPU was plugged into it. Perhaps it can be renabled?

  Make sure your CUDA application uses the GPU you want it to. It is probably sufficient to be able to specify which CUDA device your application will use via the command line.

  ```
  if(argc>1 && argv[1][0]) {
    const int dev = atoi(argv[1]);
    cutilSafeCall( cudaSetDevice( dev ));
  }
  else cudaSetDevice( cutGetMaxGflopsDeviceId() );
  ```

- If the PC you use to develop CUDA applications is on the network, arrange that another networked computer is nearby so that you can log in via the network (e.g. using ssh). While this may allow you to gain reassurance that it really is a GPU problem rather than anything else, in the event of a GPU lock up it may be that there is little you can do, other than reboot. However you should have the option of telling the operating system to shutdown in a more controlled fashion. Perhaps informing other users/applications before their resources are removed.

- Some computer rooms have facilities to allow remote reboot. This may be under software control or you may have to ring up the operator and ask them to do it for you. Make sure you tell them the right computer!
- Make sure all of your CUDA system restarts automatically on reboot. Remember to include all the "little" tweaks to the operating system and X-11 windows that were done when CUDA was installed. This is especially important if CUDA was installed by someone else or if any of the "tweaks" need the root system password to reapply them.
- With its default setting, X-11 times out your screen if it fails to respond in about 10 seconds. E.g. suppose your kernel sometimes takes 12 seconds. Every so often it will cause the GPU on which it is running not to respond to X windows fast enough. For someone who is using the screen, this appears the same as if the GPU had failed, even though the GPU may be ok. Since this only effects X-11, you may be able to recover without rebooting Linux. For example, use one of the methods mentioned above to log into the host PC and restart X. It is also possible to disable the X-11 timeout or change its default setting.

  If the GPU can be reserved for calculations only, it might make sense to configure X-11 to ignore the monitor connected to the compute only GPU. However this might effect non-GPU uses of X-11, e.g. ssh -Y.

## *9.2 Compiling CUDA C/C++ Programs*

You will need to compile your kernel with nVidia's CUDA compiler, nvcc. nvcc is also able to compile regular C and C++ code. nvcc host and GPU code can be linked with PC code compiled in the normal way. nvcc recognises many of the command line switches used by the GNU gcc compiler, such as setting conditional compilation switches (e.g. `-DUNIX`) and the debug flag `-g`). You will probably also need the GPU specific switch which tell the compiler to produce code for a particular nVidia GPU compute level (e.g. `-arch sm_20` for Fermi compute level 2.0). Check with the nvcc compiler documentation.

CUDA supports both 32 bit and 64 bit host PCs. You may need to double check you are linking the right libraries when you ask the linker to create your executable program.

## *9.3 CUDA SDK Makefile common.mk*

The CUDA SDK examples include compilation scripts, known as Makefile. Most of their complexity is common to all SDK examples and is kept in a common make file (known as `common.mk`). One approach is to organise your application so that it follows the same directory structure and file naming conventions as CUDA's SDK. This will allow you to use common.mk. However it is also possible to adapt one of the SDK Makefile for your own project.

A disadvantage of using common.mk is that it assumes particular locations for your object and executable files. By default, the GNU GDB debugger run within emacs, is not compatible with this and refuses to show your host sources inside an emacs window as you use step through (the host part) of your application. If so, it may be easier to compile and link in your usual fashion. (cuda-gdb and commercial debuggers, e.g. Parallel Nsight and Allinea DDT, are increasingly available and increasingly capable.)

## 9.4 CUDA Compilation and Linking Problems

We next describe some errors that are common when you first use CUDA or after upgrading it and suggest potential solutions.

If using Unix and SDK's common.mk a helpful option is to run make in verbose mode so that it tells you the commands it is running. This is enabled in Unix by setting the environment variable verbose. E.g. setenv verbose 1.

On some older CUDA systems the additional line, "NVCCFLAGS += -include=vararg-fix.h" in common.mk may be required to get your kernel to compile.

Error mkdir: cannot create directory '/opt/cuda/sdk': Read-only file system suggests a problem with ROOTDIR or some inconsistency between your Makefile and common.mk. Perhaps you need to try overriding ROOTDIR, e.g. ROOTDIR := /my_directory/cuda/sdk, where /my_directory... refers to the directory tree you are using for your application.

nvcc compilation error error: cutil_inline.h: No such file or directory suggests a problem with COMMONDIR or some inconsistency between common.mk and your Makefile. Perhaps try overriding ROOTDIR2, e.g. ROOTDIR2 := /usr/local/cuda/NVIDIA_GPU_Computing_SDK/C/tools. Of course the actual setting for ROOTDIR2 will depend on where exactly the files were placed when CUDA was installed.

nvcc compilation error error: cuda_runtime.h: No such file or directory. Again perhaps a problem with ROOTDIR2, however also check your system does really have a copy of cuda_runtime.h installed somewhere. It might also be a problem with CUDA_INSTALL_PATH. If so, you could try overriding it with something like CUDA_INSTALL_PATH := /usr/local/cuda-3.0

The Unix linker error /usr/bin/ld: cannot find -lcutil suggests a problem with LIBDIR or inconsistency between make files. This can occur when there are multiple versions of CUDA installed. Perhaps try overriding LIBDIR, e.g. by adding something like LIBDIR := /my_directory/cuda_3.1/cuda/

`NVIDIA_CUDA_SDK/lib`. However eventually it may be better to resolve the problem of multiple version of CUDA and/or create your own make file or compilation script or process.

The Unix linker error `ld: skipping incompatible /usr/local /cuda-3.0/lib/libcudart.so when searching for -lcudart` might be a 32 bit v 64 bit problem. The Unix `file` utility will tell you if `libcudart.so` contains 32 or 64 bit code. Perhaps you need to change `LIBDIR` with something like `LIBDIR := /usr/local/cuda/lib64`

If you get `error while loading shared libraries: libcudart .so.2: cannot open shared object file: No such file or directory` this suggests your `LD_LIBRARY_PATH` environment variable is incorrectly defined. `LD_LIBRARY_PATH` allows the Unix program starter to search for `libcudart.so.2` in multiple directories. These are separated by a ":". Assuming you have an existing `LD_LIBRARY_PATH` environment variable then an option is to append the directory holding `libcudart.so.2` E.g. `setenv LD_LIBRARY_PATH "$ LD_LIBRARY_PATH":/usr/opt/cuda/lib`.

## 10 Other Sources of Help with Parallel Software Development

### *10.1 nVidia*

nVidia has made available a host of documentation for CUDA and each of its components. Typically these are freely downloadable in PDF format.

A typical CUDA installation comes in three parts: GPU operating system drivers, CUDA toolkit and CUDA SDK. It is well worth installing the SDK directory tree when you install the first two. It contains more than 70 CUDA programming examples and GPGPU utilities, including their source code and in some case detailed documentation.

The SDK examples often both explain and give examples of tricky but highly efficient parallel computing approaches and are of course written for a GPU like yours. Examples include fast matrix multiply and calculating histograms in parallel. These examples show how to efficiently use shared memory in CUDA C.

### *10.2 nVidia Forums*

nVidia hosts an impressive array of discussion fora at forums.nvidia.com. There are perhaps too many for an individual and it is better to stick to the one closest to your interest. For GPGPU the CUDA Programming and Development forum has proved useful.

### *10.3 Other Venues*

There are many other Internet web pages in addition to those hosted by nVidia. For example, Simon Harding runs gpgpgpu.com specifically for combining genetic programming and GPUs whilst gpgpu.org is more generic. Whereas gpgpu.org does not deal specifically with bioinspired algorithms, there are a number of workshops and special events which do. For example, Computational Intelligence on Consumer Games and Graphics Hardware CIGPU, has run annually since 2008. Similarly the Workshop on Parallel Architectures and Bioinspired Algorithms WPABA has also run each year. With a winder remit than just GPUs, EvoPAR is set to become a track within the european evolutionary computing EvoApplications conference.

### *10.4 Alternative Approaches*

We have talked about CUDA C. Is CUDA C the right language to choose? C is notoriously difficult and other languages are being added (e.g. Fortran, Matlab, Mathematica and Python). Nevertheless we can be reasonably confident that in the near term C/C++ will remain both the most efficient high level language for GPU computing and the most advanced and best supported CUDA programming language. CUDA is and is expected to remain nVidia's best way into the GPGPU world. However you might want your application to run on other manufacturer's GPUs or even non-GPU parallel hardware. OpenCL has been proposed by a small group of companies (including nVidia, AMD, Intel, Apple and IBM) as a way of implementing parallel applications. In theory it offers the possibility of running code on both GPUs from different manufactures and other parallel architectures. Currently support is patchy in practice.

In 2007 Harding gave a nice summary GPGPU tools [10]. It is notable that many have already fallen out of use. The software side of GPU computing has proved less stable than the underlying GPU architectures.

## 11 Conclusions

Some physical devices, e.g. some types of disk drive, give some indication of being used (e.g. audible hum or clicks, change in appearance or shaking, or getting hot). However, as with most solid state electronic devices, GPUs give little physical indication of how much they are being used or how close to they are to their maximum performance. To get the best of GPGPU you must use software techniques to predict, design and monitor performance. Sections 4 and 5 described how high GPGPU performance can be obtained and measured in practise.

Although tools continue to improve debugging CUDA C remains hard. Section 6 and 7 gave practical ideas for debugging and testing, whilst Section 8 describes how they were used with real bugs. The last two sections give practical advice on setting up your CUDA development system, other sources of help and alternatives.

## *Computation Is Cheap. Data Is Expensive*

Perhaps slightly too strong but I have put it strongly to make the point. Wasting computing power does not come naturally. It is the opposite of what we were told as students. Nevertheless when on a GPU it can be more efficient to waste computation than move data.

It may be better to recalculate intermediate results than to store them. E.g. in some large matrix calculations. This is especially true if the intermediate results have to be saved on the host computer. On a GPU it often takes longer to move data than it does to calculate with it once it has arrived.

Since the GPU multiprocessors can only execute one instruction at a time, closely linked parallel threads which need to run different code have to run sequentially, not in parallel. Divergence represents idle compute resources. Effectively divergences is throwing away computation. But computation is cheap! It may be better to discard it than be unable to use a GPU at all.

The trend is for the cost of computation to fall faster than the cost of moving data. Thus the balance will continue to move in favour of more intensive calculations.

## *Debugging Is the Most Expensive Thing You Can Do*

Avoid writing new code. Do you really need new code? Can you reuse nVidia's examples? Can you use an existing library?

Does it makes sense to treat your application as a matrix manipulation problem.

Is there an existing solution written in a matrix manipulation language (e.g. Matlab) which will run on your GPU?

Is there an existing GPGPU solution? Perhaps it is available on the Internet via FTP?

## *The Future of Bioinspired Applications*

Life is parallel. Nature runs in parallel. Chemical molecules react when they meet. Antibodies neutralise antigens. Nerve cells fire at the same time. Ants follow trials. Bees swarm. Birds flock. Fish school. Populations mate and rear their young simultaneously. In many cases bioinspired algorithms are naturally parallel. Even embarrassingly parallel. Typically there is a good fit to parallel computing. This is especially true of low cost GPGPU computing.

Although a main stream break though in parallel computing has been forecast for at least 30 years [2] the 3 GHz ceiling has forced the hardware manufactures to generate affordable massively parallel computers and provide software support for them. Already there are many parallel bioinspired applications (Section 2) and with improving parallel development tools and cheap hardware, GPGPU (perhaps soon GPPPU) based applications have a great future.

**Acknowledgements.** I am grateful for the assistance of Gernot Ziegler of nVidia, Steve Worley, Sarnath Kannan and Erick Cantu-Paz. The T10P early engineering sample and C2050 Teslas were given by nVidia. Funded by EPSRC grant EP/G060525/2.

## References

1. Anderson, D.T., Luke III, R.H., Keller, J.M.: Speedup of fuzzy clustering through stream processing on graphics processing units. IEEE Transactions on Fuzzy Systems 16(4), 1101–1106 (2008), `http://dx.doi.org/10.1109/TFUZZ.2008.924203`
2. Arabnia, H.R., Oliver, M.A.: A transputer network for the arbitrary rotation of digitised images. The Computer Journal 30(5), 425–432 (1987), `http://comjnl.oxfordjournals.org/cgi/reprint/30/5/425.pdf`
3. Bakhoda, A., Yuan, G.L., Fung, W.W.L., Wong, H., Aamondt, T.M.: Analyzing CUDA workloads using a detailed GPU simulator. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston, MA, USA, April 26-28, pp. 163–174 (2009), `http://dx.doi.org/10.1109/ISPASS.2009.4919648`
4. Cantu-Paz, E., Goldberg, D.E.: Efficient parallel genetic algorithms: theory and practice. Computer Methods in Applied Mechanics and Engineering 186(2-4), 221–238 (2000), `http://dx.doi.org/10.1016/S0045-78259900385-0`
5. Ebner, M., Reinhardt, M., Albert, J.: Evolution of Vertex and Pixel Shaders. In: Keijzer, M., Tettamanzi, A.G.B., Collet, P., van Hemert, J., Tomassini, M. (eds.) EuroGP 2005. LNCS, vol. 3447, pp. 261–270. Springer, Heidelberg (2005), `http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/eurogp_EbnerRA05.html`
6. Fernando, R., Kilgard, M.J.: The Cg Tutorial. Addison-Wesley, nVidia (2003)
7. Fok, K.-L., Wong, T.-T., Wong, M.-L.: Evolutionary computing on consumer graphics hardware. IEEE Intelligent Systems 22(2), 69–78 (2007), `http://dx.doi.org/10.1109/MIS.2007.28`
8. Garland, M., Kirk, D.B.: Understanding throughput-oriented architectures. Communications of the ACM 53(11), 58–66 (2010)
9. Greene, C.S., Sinnott-Armstrong, N.A., Himmelstein, D.S., Park, P.J., Moore, J.H., Harris, B.T.: Multifactor dimensionality reduction for graphics processing units enables genome-wide testing of epistasis in sporadic ALS. Bioinformatics 26(5), 694–695 (2010), `http://dx.doi.org/10.1093/bioinformatics/btq009`
10. Harding, S., Banzhaf, W.: Fast Genetic Programming on GPUs. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 90–101. Springer, Heidelberg (2007), `http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/eurogp07_harding.html`
11. Harding, S.L., Banzhaf, W.: Distributed genetic programming on GPUs using CUDA. In: Hidalgo, I., Fernandez, F., Lanchares, J. (eds.) Workshop on Parallel Architectures and Bioinspired Algorithms, Raleigh, NC, USA, September 13, pp. 1–10. Universidad Complutense de Madrid (2009), `http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/hardinggpem2009.html`
12. Harvey, N., Luke, R., Keller, J.M., Anderson, D.: Speedup of fuzzy logic through stream processing on graphics processing units. In: Wang, J. (ed.) 2008 IEEE World Congress on Computational Intelligence, Hong Kong, June 1-6, pp. 3809–3815 (2008), `http://dx.doi.org/10.1109/CEC.2008.4631314`

13. Kernighan, B.W., Ritchie, D.M.: The C Programming Language, 2nd edn. Prentice-Hall, Englewood Cliffs (1988)
14. Koza, J.R.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT press (1992), http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/koza_book.html
15. Langdon, W.B.: Evolving GeneChip correlation predictors on parallel graphics hardware. In: Wang, J. (ed.) 2008 IEEE World Congress on Computational Intelligence, Hong Kong, June1-6, pp. 4152–4157 (2008), http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/langdon_2008_CIGPU2.html
16. Langdon, W.B.: A fast high quality pseudo random number generator for nVidia CUDA. In: Wilson, G. (ed.) CIGPU Workshop at GECCO, Montreal, July 8, pp. 2511–2513. ACM (2009), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/langdon_2009_CIGPU.pdf
17. Langdon, W.B.: A Many Threaded CUDA Interpreter for Genetic Programming. In: Esparcia-Alcázar, A.I., Ekárt, A., Silva, S., Dignum, S., Uyar, A.Ş. (eds.) EuroGP 2010. LNCS, vol. 6021, pp. 146–158. Springer, Heidelberg (2010), http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/langdon_2010_eurogp.html
18. Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nVidia template. In: Sobrevilla, P. (ed.) 2010 IEEE World Congress on Computational Intelligence, Barcelona, July 18-23, pp. 2376–2383 (2010), http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/langdon_2010_cigpu.html
19. Langdon, W.B., Harrison, A.P.: GP on SPMD parallel graphics hardware for mega bioinformatics data mining. Soft Computing 12(12), 1169–1183 (2008), http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/langdon_2008_SC.html
20. Langdon, W.B., Yoo, S., Harman, M.: Formal concept analysis on graphics hardware. In: Napoli, A., Vychodil, V. (eds.) The Eighth International Conference on Concept Lattices and Their Applications, Nancy, France, October 17-21, pp. 413–416. INRIA Nancy and LORIA (2011), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/langdon_2011_cla.pdf
21. Langdon, W.B., Banzhaf, W.: A SIMD Interpreter for Genetic Programming on GPU Graphics Cards. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 73–85. Springer, Heidelberg (2008), http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/langdon_2008_eurogp.html
22. Langdon, W.B., Harman, M., Jia, Y.: Efficient multi-objective higher order mutation testing with genetic programming. Journal of Systems and Software 83(12), 2416–2430 (2010), http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/langdon_2010_jss.html
23. Moler, C.: Matrix computation on distributed memory multiprocessors. In: Heath, M.T. (ed.) Proceedings of the First Conference on Hypercube Multiprocessors, Knoxville, Tennessee, USA, August 24-27, pp. 181–195. Society for Industrial and Applied Mathematics (1986), http://books.google.co.uk/books?id=QN8HNVwZEecC&printsec=frontcover&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false
24. Moore, G.E.: Cramming more components onto integrated circuits. Electronics 38(8), 114–117 (1965), ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf

25. Mussi, L., Cagnoni, S., Daolio, F.: GPU-based road sign detection using particle swarm optimization. In: Ninth International Conference on Intelligent Systems Design and Applications, ISDA 2009, Pisa, Italy, pp. 152–157. IEEE (2009), November 30-2 December, http://dx.doi.org/10.1109/ISDA.2009.88
26. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proceedings of the IEEE 96(5), 879–899 (2008), http://dx.doi.org/10.1109/JPROC.2008.917757; (invited paper)
27. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming (2008) (With contributions by Koza, J.R.), Published via http://lulu.com, freely available at http://www.gp-field-guide.org.uk
28. Prabhu, R.D.: SOMGPU: an unsupervised pattern classifier on graphical processing unit. In: Wang, J. (ed.) 2008 IEEE World Congress on Computational Intelligence, Hong Kong, June 1-6, pp. 1011–1018 (2008), http://dx.doi.org/10.1109/CEC.2008.4630920
29. Reeves, C.R., Rowe, J.E.: Genetic Algorithms–Principles and Perspectives: A Guide to GA Theory. Kluwer Academic Publishers (2003)
30. Ribeiro Filho, J.L., Treleaven, P.C.: Genetic-algorithm programming environments. Computer 27(6), 28 (1994), http://www.cs.bham.ac.uk/~wbl/biblio/gp-html/RibeiroFilho_1994_GPE.html
31. Rieffel, J., Saunders, F., Nadimpalli, S., Zhou, H., Hassoun, S., Rife, J., Trimmer, B.: Evolving soft robotic locomotion in PhysX. In: GECCO 2009: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference, Montreal, Québec, Canada, July 8-12, pp. 2499–2504. ACM (2009), http://dx.doi.org/10.1145/1570256.1570351
32. Sinnott-Armstrong, N.A., Granizo-Mackenzie, D., Moore, J.H.: High performance parallel disease detection: an artificial immune system for graphics processing units. In: GECCO 2010 GPUs for Genetic and Evolutionary Computation, Winning Entry (2011), http://www.gpgpgpu.com/gecco2010/2.pdf
33. Stender, J. (ed.): Parallel Genetic Algorithms: Theory and Applications. IOS press (1993)
34. Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., Moshovos, A.: Demystifying GPU microarchitecture through microbenchmarking. In: IEEE International Symposium on Performance Analysis of Systems Software (ISPASS 2010), White Plains, NY, USA, March 28-30, pp. 235–246 (2010), http://dx.doi.org/10.1109/ISPASS.2010.5452013
35. Yamamoto, L., Banzhaf, W., Collet, P.: Evolving Reaction-Diffusion Systems on GPU. In: Antunes, L., Pinto, H.S. (eds.) EPIA 2011. LNCS, vol. 7026, pp. 208–223. Springer, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-24769-9_16
36. Yudanov, D., Shaaban, M., Melton, R., Reznik, L.: GPU-based implementation of real-time system for spiking neural networks. In: Sobrevilla, P. (ed.) 2010 IEEE World Congress on Computational Intelligence, Barcelona, July 18-23, pp. 2143–2150 (2010), http://dx.doi.org/10.1109/IJCNN.2010.5596334
37. Zhu, W., Curry, J.: Parallel ant colony for nonlinear function optimization with graphics hardware acceleration. In: IEEE International Conference on Systems, Man and Cybernetics, SMC 2009, San Antonio, Texas, USA, October 11-14, pp. 1803–1808 (2009), http://dx.doi.org/10.1109/ICSMC.2009.5346870

# Optimizing Shape Design with Distributed Parallel Genetic Programming on GPUs

Simon Harding and W. Banzhaf

**Abstract.** Optimized shape design is used for such applications as wing design in aircraft, hull design in ships, and more generally rotor optimization in turbomachinery such as that of aircraft, ships, and wind turbines. We present work on optimized shape design using a technique from the area of Genetic Programming, self-modifying Cartesian Genetic Programming (SMCGP), to evolve shapes with specific criteria, such as minimized drag or maximized lift. This technique is well suited for a distributed parallel system to increase efficiency. Fitness evaluation of the genetic programming technique is accomplished through a custom implementation of a fluid dynamics solver running on graphics processing units (GPUs). Solving fluid dynamics systems is a computationally expensive task and requires optimization in order for the evolution to complete in a practical period of time. In this chapter, we shall describe both the SMCGP technique and the GPU fluid dynamics solver that together provide a robust and efficient shape design system.

## 1 Introduction

Optimized shape design (OSD) is a problem of optimal control theory, where the task is to find a shape that minimizes certain parameters while satisfying a set of constraints. In this chapter we describe an OSD technique that uses inspiration from biological evolution to design hydrodynamic shapes (such as wing surfaces) which meet certain criteria like the minimization of drag or the

Simon Harding
IDSIA (Istituto Dalle Molle di Studi sull'Intelligenza Artificiale), Switzerland
e-mail: simon@idsia.ch

Wolfgang Banzhaf
Memorial University of Newfoundland, Canada
e-mail: banzhaf@mun.ca

F.F. de Vega et al. (Eds.): Parallel Architectures & Bioinspired Algorithms, SCI 415, pp. 51–75.
springerlink.com

maximization of lift. The approach we shall consider is general and could be used to find good shapes for car bodies, aeroplane fuselages, boat hulls, etc. The technique uses a so-called distributed parallel evolutionary algorithm to optimize the solution, along with a general purpose parallel fluid dynamics solver to evaluate the shape parameters.

Evolutionary algorithms (EAs) have a long history of being used to generate designs for physical objects. In fact, one of the branches of this field, Evolutionary Strategies, started out with a problem for nozzle design [25, 27]. Generally, EAs mimic mechanisms of biological evolution (populations of solutions under mutation and recombination, using fitness evaluation to determine what is being promoted from one generation of solutions to the next) to solve optimization problems. In the area of design previous examples include electronic circuit design, furniture design, or the design of structural features of buildings, and aerodynamic shape design. Section 2 describes previous attempts to evolve aerodynamic shapes using similar approaches. Section 3 then introduces a genetic programming technique for the optimized shape design. Genetic Programming [14, 23] is another branch of EAs that has recently become more prominent due to its ability to adapt solutions to the complexity of a problem at hand.

Generally, the work flow of the method includes steps in which such designs need to be tested in a simulated environment, complete with models of physics. This will result in the assignment of solution quality to each of the individual solutions in the population, provided quality ("fitness" criteria, in EA speak) have been defined beforehand. Physical simulations, however, are notoriously expensive in terms of computation time - but with recent advances in Graphics Processing Units (GPUs) it is now possible to speed up these simulations and hence the fitness evaluation of solutions consisting of complicated objects within a complex physical system at relatively low cost.

Graphics processing units are a specific type of parallel many-core processing units. GPUs are cheap and ubiquitous, they are now present in almost all modern PCs and laptops to enhance performance. Originally designed for gaming and graphics processing, they have evolved into general purpose processing units with advantages for the solution of many types of scientific problems. Section 4 describes the hardware and software models of GPUs and their advantages for this type of problem.

Computational Fluid Dynamics (CFD) is an area of fluid mechanics that uses algorithms and numerical methods to solve fluid flow problems. Within CFD there are many different approaches, involving different solution methods. The choice of a solution method is largely dependent on the problem, on the context of the problem, and on what is required of a solution for later analysis (post-processing). Section 5 will describe a technique to simulate and evaluate general purpose fluid design environments using GPUs to increase efficiency.

## 2 Evolving Aerodynamic Shapes

Nature is full of examples of creatures that are adapted to operate with aerodynamic or hydrodynamic requirements. Wings, fins, streamlined bodies, textures to minimize turbulence, and feathers are all examples of this. Given the variety and efficiency of what nature produces it seems appropriate to use the same principles to solve man-made challenges.

Evolutionary algorithms have already been successfully applied to optimizing the design of objects that interact with a fluid environment. For example, in [22] genetic algorithms, another branch of EAs, are used to optimize the design of airfoils. Two-dimensional (2D) representations of airfoils were specified as a set of control points for B-Splines. The evolutionary algorithm adjusted these control points until a satisfactory arrangement was found. Fitness of candidate airfoils was determined by simulating the design and measuring variables such as pressure rise. Using a similar technique, nozzles for rocket engines have been optimized, too[3].

In [24], the authors applied a hybrid system of a genetic algorithm and a neural network to optimize the design of yacht keels. Here, the parameters for the keel design were optimized to reduce drag and maximize lift. In [4], airfoils were evolved, again using Bezier surfaces to define the sections. Recently, [1] built on this work employing a grid of computers to reduce the bottleneck of fitness evaluation.

Wing evolution has also been demonstrated using physical models, where the rotation of a number of connecting plates was altered and tested for lift in a wind tunnel [26]. Rechenberg has also used a similar method to examine other related systems such as the evolution of the wing tips of birds and nozzle designs.

Previous approaches have largely involved either optimizing parameters for known designs or using Bezier/Nurbs surfaces. This was well suited for genetic algorithms or evolutionary strategies, as it significantly reduced the search space. With such constrained representations, however, it is difficult to imagine how radically new designs could be produced or how these techniques could be expanded to evolve more complex, multi-component systems. In other words, as soon as more creativity in solutions is required, or one is prepared to test truly novel ideas, other techniques will need to be applied.

There have been several attempts to implement an evolutionary design of objects. The aim of the work described in this chapter is to allow for arbitrary structures to be evolved, and therefore the Bezier control point based representations mentioned above are not used. The requirements for this work include the ability to produce both 2D and 3D representations, single objects and non-connected designs, to produce vectorized objects that allow for distortion-free scaling and rotation, and the ability to produce curved/free form shapes. It is also envisaged that interesting designs would include concepts such as symmetry, repetition or repetition with variation. These requirements suggest that a genetic programming approach in the context of a developmental system would be appropriate [17, 2].

## 3 Self-modifying Cartesian Genetic Programming (SMCGP)

Self-modifying Cartesian Genetic Programming (SMCGP) is a developmental version of Genetic Programming. In brief, SMCGP is a way of evolving computer programs that can change their own structure (and hence behaviour) at runtime. This method has been used for numerous applications, such as evolving digital circuits [9, 10, 6], finding algorithms that approximate physical constants [11, 12], discovering learning algorithms [8] and regression and classification [7].

As the name suggests, SMCGP is based on the Cartesian Genetic Programming (CGP) technique. In CGP, programs are encoded in a partially connected feed forward graph. (see [18]). The genotype encodes this graph, with each node represented as a function and connections to other nodes that this function connects to. The representation has a number of interesting features. For instance, not all of the nodes of a solution representation ("the genotype") need to be connected to the output node of the program, so there are nodes in the representation that have no effect on the output, a feature known in GP as "neutrality". This has been shown to be very useful [20] for the evolutionary process. Also, because the genotype encodes a graph, there can be reuse of nodes (revisiting of nodes is allowed), which makes the representation distinct from a classically tree-based GP representation.

Although CGP has been used in other developmental systems [19, 15], the programs that those approaches produced were not themselves developmental. SMCGP, on the other hand, was designed as an attempt to bring development into CGP so that CGP could be used as a general purpose developmental GP system.

The SMCGP representation is similar to CGP in some ways, but has extensions that allow it to exhibit self-modifying features. SMCGP genotypes are a linear string of nodes. Each node connects to two other nodes by way of a relative address, which states how many nodes back to connect. To prevent cycles, nodes can only connect to other nodes in one direction. Relative addressing allows entire sections of the graph to be moved, duplicated, deleted, etc, without breaking the reference structure, whilst allowing some sort of modularity.

In overview, each node in an SMCGP graph contains a number of elements:

- The computation function, represented in the genotype as an integer;
- A list of (relative) connection addresses, again represented as integers;
- A set of parameters, represented by 3 floating point numbers.

As with CGP, the number of nodes in the genotype is typically kept constant throughout an evolutionary run. However, this means care has to be taken to ensure that the genotype is large enough to store a possibly complex target program. Any kind of adjustment to the complexity would then come from the turning on and off of node execution paths through this graph which we shall explain next.

### *3.1 Executing a SMCGP Individual*

SMCGP individuals are evaluated in a multi-step fashion, with the evolved program (the "phenotype") executed several times. An evolved program in SMCGP initially has the same structure as the genotype, which is supposed to represent it. The first step in producing the phenotype is to simply make a copy of the genotype and call it the initial phenotype. This graph is the 'working copy' of the program that will later be modified during further execution of nodes. Each time the program is executed, the phenotype graph is first run and then any self-modification operations encoded are invoked.

The graph is executed in the following manner: First, the node (or nodes) to use as output(s) are identified. This is done by parsing through the graph looking for nodes of type OUTPUT. Once a sufficient number of these nodes has been found, the various nodes that they connect to are identified using recursion. In case that there are not enough output nodes found in this way, the last $n$ nodes in the graph are rededicated as output nodes, where $n$ means the number of outputs required. If there are not even enough nodes to satisfy this condition, execution is aborted and the individual is discarded as lethal.

At this point, all the nodes that are used by the program have been identified and so their values can be calculated. For mathematical and binary functions, these operations are performed in the usual manner. However, SMCGP has a number of special functions (see Table 1) that allow for self-modification.

If a function is a self-modification function, then it may be activated. Binary functions are always activated, but numeric nodes are activated only if the first input is larger than the second input. The self-modification operation of an activated node is added to a list of pending operations - the 'ToDo' list. After execution, the self-modification functions on the ToDo list are applied to the current graph, up to a maximum number of self-modification operations which is a parameter of the system.

In turn, the self-modification functions usually require parameters, which are taken from the parameters part of the calling node. Many of the parameters are integers, so the parameters may need to be cast into integer numbers. For instance, parameters may be treated as relative addresses depending on the function. The program can now be iterated again, if necessary. It is important to note that modifications are only made to the phenotype, and not to the genotype.

In the current work, we extended SMCGP to allow for design generation. To do this, several changes to SMCGP are required. Extra functions are added to the function set that perform various drawing operations. To support this, each node in the genotype additionally encodes for a structure that can represent the parameters of these shape functions, the shape data type (SDT).

An SDT structure contains five vectors of four element each. One of these vectors is labeled as "source", another is labeled as "destination". Four-element vectors are used so that we can easily move to a 3D representation. In 3D geometry, 4 element vectors are very useful as they can represent rotations and transformations as quaternions. Elements are referred to as $x, y, z$ and $w$. The five vectors in the SDT represent entry and exit location, rotation, size and value. Value can be considered as a holder for some additional parameters.

These source and destination vectors are used when mathematical operations are applied upon two SDT structures. For example, to ADD two structures A and B, a copy of A is made and the destination vector in A is set to the sum of the source vectors in A and B. The structure contains 5 vectors and each vector encodes a different parameter that is used to specify a shape location, size, rotation and connectivity, as well as an additional parameter.

When a shape is drawn, it is drawn with respect to a current position (origin) and rotation. Further parameters specify the size of each dimension of the shape. Two of the parameters encode additional position information as to where to start drawing the object (entry position) and where the next origin should be (exit position). The entry and exit positions are relative to the origin and rotation.

Consider Figure 1. The origin and initial rotation are determined by the previous drawing operation, or by a predetermined position for the first shape. The shape is then drawn relative to the origin and entry positions. The subsequent origin will be the exit position of the last node, and the subsequent

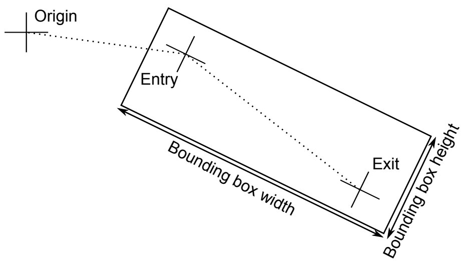


**Fig. 1** Shapes are defined relative to an initial origin, with Entry and Exit positions defined relative to the shape. The entry and exit positions, shape rotation and shape geometry are under evolutionary control, and can be modified by the SMCGP program at run time.

initial rotation vector will be the current rotation added to this shape's rotation vector. Entry, exit and rotation values are taken from the SDT structure.

The shape parameters are calculated from the value of the SDT passed to that node. Hence, they can be affected by computations performed by mathematical functions. This allows for more complex transformations to be performed.

To simplify the shape generation, only one shape function is allowed. This function, called the "Superformula", is able to generate a wide variety of shapes, including many that have a very biological feel [5]. Conveniently, the function also can be extended to 3D which will be useful in later work. In polar form the equation is:

$$r(\phi) = \left[ \left| \frac{\cos(\frac{m\phi}{4})}{a} \right|^{n_2} + \left| \frac{\sin(\frac{m\phi}{4})}{b} \right|^{n_3} \right]^{-\left(\frac{1}{n_1}\right)}$$

Each of the parameters $a$, $b$, $m$, $n_1$, $n_2$ and $n_3$ is under evolutionary control, defined by the values stored in the SDT. $a$, $b$ are taken from the $z$ and $w$ component of the size vector. The other four parameters are taken from the value vector.

Because the formula is written in polar coordinates, results needs to be converted to Cartesian coordinates ($p$ and $q$). In this transformation, the $x$ and $y$ values from the size vector are used to specify the radii of the transform:

$$p = x \sin(\phi)$$

$$q = y \cos(\phi)$$

The function set also contains other functions for manipulating the current origin and rotation. The MOVE command specifies a simple translation of the current origin. TRANSMC allows for the origin to be moved and scaled. A stack of origin and rotation values is also provided. The PUSHTRANSFORM and POPTRANSFORM perform operations on this stack.

Figure 2 shows the developmental steps of a simple object (a rough outline of an aeroplane). Each frame in the sequence shows the next time step in the developmental process. Here, all but the last time step add new shapes to the figure.

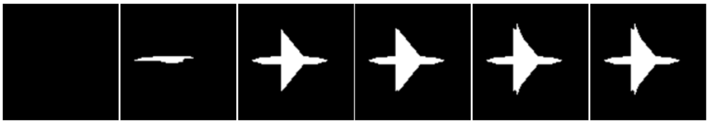


**Fig. 2** Developmental steps in drawing a simple object. Each frame represents one time step in the developmental process.

**Table 1** The SMCGP function set.

| Function | Description |
|---|---|
| MOVE | Move the origin |
| POLYGON | Draw a polygon |
| PUSHTRANSFORM | Push origin/rotation to stack |
| POPTRANSFORM | Pop origin/rotation from stack |
| TRANSMC | Translate and scale the current origin. |
| ADD, SUB, DIV, MUL | Perform the relevant mathematical operation on the source vectors |
| PRC | Executes a subgraph as a procedure |
| MOVESRCTODEST | Moves a vector from the source register to the output register |
| INDEX | Returns a STD that represents the current index of the node in the graph |
| CONST | Returns a STD that represents a set of evolved mathematical constants |
| OUTPUT | Labels this node as being the output, i.e. final connected node in the graph |
| SMDUP | Duplicates a set of nodes, inserts copy elsewhere in the graph |
| SMDEL | Deletes a set of nodes |
| SMDUPREV | Same as SMDUP, but reverses the order of the inserted nodes |

## 4 Graphics Processing Units (GPUs)

Graphics Processing Units (GPUs) have a many-core parallel architecture. They consist of a set of stream processors that execute programs (also called kernels) in parallel. GPUs were originally designed for graphics processing, so the stream processors are designed for small and fast operations (per stream processor) such as filtering a texture. A simple description of GPU programming and hardware models is given in this section. For more information about the both GPU architecture(s) and programming models readers should consult [21].

The programming model used for GPUs is built around a SIMT (single-instruction multiple-thread) architecture concept. SIMT is not the same as the traditional SIMD (single instruction multiple data) concept in that SIMD applies the same instruction to multiple pieces of data simultaneously, while SIMT executes the same thread (code block) simultaneously with a single instruction. A typical program execution on a GPU consists of a mapping of the threads (or kernels) to a two-level grid of a user-specified size (see Figure 3). The threads are mapped as a set of threads, grouped into blocks. The number of blocks in the grid is called the *grid size* and the number of threads per block is called the *block size*. Once the threads are mapped they are then enumerated and distributed to the available cores on the device. Scheduling of these kernels, as threads of the grid are terminated and new ones are executed, is performed automatically on the GPU itself.

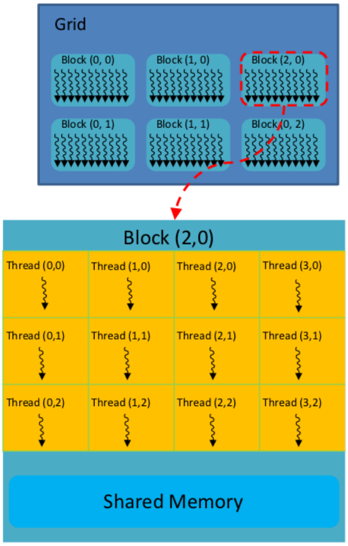


**Fig. 3** GPU Program Model

The hardware model of most GPUs, illustrated in Figure 4, are generally designed as an array of multi-threaded Streaming Multiprocessors (SMs). Each of these processors contain a set of Scalar Processor (SP) cores (currently all NVIDIA devices contain eight cores per SM), a multi threaded instruction unit, and a shared memory unit for that multiprocessor. Outside of the array of SMs there is at least one memory space, most importantly the main device memory, that is in use by all components of the GPU (other memory spaces are not relevant to this chapter). Device memory is the slowest on-card memory, while shared memory (per SM) is the fastest on-card, next to the registers of course but not far behind [21].

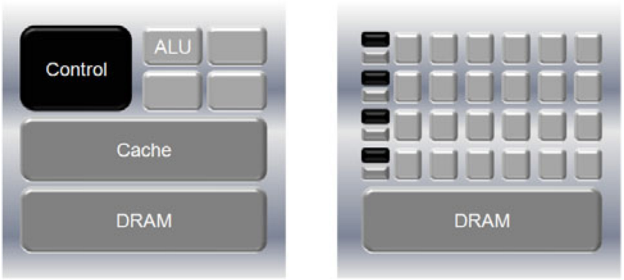


**Fig. 4** GPU Hardware Model

A general set of optimization rules for developing any type of algorithm for GPUs is:

**Memory Transfers.** Memory transfer to or from device memory is the slowest individual operation that can be performed on a GPU. For this reason memory transfers should be kept to a minimum for any algorithm developed for the GPU. An optimal design approach is to design your algorithm to perform all operations on the main data in device memory and transfer only the results back to host memory.

**Memory Coalescence.** Memory coalescence is the pattern of reads performed on the device memory. For example, if text is read one word at a time (each kernel reads one word, and assuming words are stored in memory in the order that they are written) to perform some operation on each word, then each kernel should read blocks of memory locations that are contiguous. In contrast, an algorithm that would have kernels read a set of random words from the text would cause many random read locations per kernel and therefore be inefficient. The former is an example of good memory coalescence which is ultimately a consequence of the architectural design choices made for GPUs optimized for a uniform memory access strategy.

**Domain Decomposition.** When developing a parallel algorithm care should be taken to decompose the domain in order to allow separate thread blocks to run on separate sub-domains of a discretized physical system. This will allow memory access and thread usage/scheduling for thread blocks on multiprocessors to be optimized. Keeping with the text reading example, one would decompose a text into separate paragraphs and map one word to one thread and one paragraph to one thread block so that each multiprocessor can process a paragraph at a time.

**Shared Memory.** Shared memory is much faster than device memory on GPUs. If an algorithm involves reads of the same subset of data for multiple kernels with a thread block, this data should be loaded into shared memory at the start of the thread block (which would require a thread sync call in order for all threads within the block to be processed up to the point where shared memory is loaded). Using the text reading example again, it would be the requirement that each thread would have to have read access to three words (the working word, the one prior it and the one directly following) which would cause an overlap in the read of surrounding words. The optimal approach would be to load all words in a paragraph into the shared memory of that thread block and then process the data.

**Multiprocessor Occupancy.** The GPU occupancy (CUDA Occupancy for NVIDIA GPUs) is a measure of kernel invocation that describes how well the kernels make use of the multiprocessor resources located on GPUs, such as allocated registers and shared memory. This concept is best described by NVIDIA [21] and is related to domain decomposition and algorithm design. Care must be taken to divide a sequential algorithm into operations that can be converted to kernel calls so that each kernel does not require too much limited resources (shared memory, registers, etc). Otherwise, the algorithm is not optimally designed and resources could be wasted. The overall goal is to maximize the multiprocessor occupancy measure.

## 5 Computational Fluid Dynamics (CFD) on Graphics Processing Units

The governing equations of a fluid system are at a minimum the continuity equation for mass and the Navier-Stokes equation, although others may be applied as required by the system in question, such as the equation of state, conservation of mass, conservation of energy, and/or boundary condition equations. The continuity equation for mass and the Navier-Stokes equation will be all we can discuss in this chapter, but the method can be easily extended to other equations using the same techniques.

The continuity equation is a description of the transport of mass under mass conservation,

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{1}$$

where $\rho$ is the density of the fluid, $t$ is the time, and $\mathbf{u}$ is the velocity vector. Since we are only concerned with incompressible fluid flow, the incompressible Navier-Stokes equation is relevant,

$$\rho \left( \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f} \tag{2}$$

where $p$ is the pressure, and $\mathbf{f}$ symbolizes external forces. Equations (1) and (2) comprise the required equations for solving incompressible transient (time dependent) fluid flow. It is important to understand that not all solutions to fluid flow are required to be transient, some simple flows have time independent solutions, or steady-state solutions. The steady-state equations are similar to the above equations but do not contain explicit time dependence.

The main issue in solving these equations is that they are coupled nonlinear differential equations. Solving these types of equations usually requires an iterative method that optimizes an approximation to the equation solution. Next, we will describe the iterative methods used to solve both the steady-state and transient fluid flow equations.

## 5.1 Method for Solving Fluid Equations

The iterative method used to solve transient fluid flow equations is called the PISO (Pressure Implicit Splitting of Operators) method. This method requires that the system be discretized, we take the example of a finite volume (FV) discretization here. The PISO (Pressure Implicit with Splitting of Operators) method is described in Algorithm 1.

From this algorithm, the most computationally expensive step for each iteration is solving the momentum and pressure correction equations, which are systems of linear equations. Another iterative method can be used to solve these systems of linear equations. It is called the successive over-relaxation (SOR) method and we will discuss it in the next section, specifically a version for GPUs (the SOR-GPU method).

## 5.2 Solving Fluid Equations on GPUs

The fluid flow simulation algorithm discussed in Section 5.1 requires a general set of operations:

- Construct coefficient matrices for systems of linear equations
- Solve systems of linear equations
- Apply corrections to flow fields
- Check convergence (residual sum).

In order to ensure speed optimization on GPUs it is best to keep all data in GPU memory with minimal swapping to host or main memory because host-to-device and device-to-host memory transfers are the slowest single operation on GPUs. The fluid simulation method discussed in this chapter keeps all relevant data in the GPU memory at all times. While this limits the size of the system we can simulate (to the amount of memory available on the GPU), it ensures optimal simulation speed. The following sections describe the design of these operations optimized for GPUs.

**Algorithm 1.** PISO algorithm

```
Initialize guesses for $p^*$, $u^*$, $v^*$.
repeat
    {STEP 1: Solve discretized momentum equations to get $u^*$, and $v^*$}
    $a_{i,j}u^* = \sum a_{nb}u^*_{nb} + \frac{1}{2}(p^*_{i-1,j} - p^*_{i+1,j})A_{i,j} + b_{i,j}$
    $a_{i,j}v^* = \sum a_{nb}v^*_{nb} + \frac{1}{2}(p^*_{i,j-1} - p^*_{i,j+1})A_{i,j} + b_{i,j}$

    {STEP 2: Solve pressure correction equation to get $p'$}
    $a_{i,j}p'_{i,j} = a_{i-1,j}p'_{i-1,j} + a_{i+1,j}p'_{i+,j} + a_{i,j-1}p'_{i,j-1} + a_{i,j+1}p'_{i,j+1} + b_{i,j}$

    {STEP 3: Correct pressure and velocities}
    $p_{i,j} = p^*_{i,j} + p'_{i,j}$
    $u_{i,j} = u^*_{i,j} + \frac{1}{2}d\_u_{i,j}(p'_{i-1,j} - p'_{i+1,j})$
    $v_{i,j} = v^*_{i,j} + \frac{1}{2}d\_v_{i,j}(p'_{i,j-1} - p'_{i,j-1})$

    $p^* = p$; $u^* = u$; $v^* = v$

    {STEP 4: Solve second pressure correction equation to get $p'$}
    $a_{i,j}p''_{i,j} = a_{i-1,j}p''_{i-1,j} + a_{i+1,j}p''_{i+,j} + a_{i,j-1}p''_{i,j-1} + a_{i,j+1}p''_{i,j+1} + b_{i,j}$

    {STEP 5: Correct pressure and velocities using second pressure correc-
    tion}
    $p_{i,j} = p^*_{i,j} + p''_{i,j}$
    $u_{i,j} = u^*_{i,j} + \frac{1}{2}d\_u_{i,j}(p''_{i-1,j} - p''_{i+1,j})$
    $v_{i,j} = v^*_{i,j} + \frac{1}{2}d\_v_{i,j}(p''_{i,j-1} - p''_{i,j-1})$

    $p^* = p$; $u^* = u$; $v^* = v$
until convergence
```

### 5.2.1 Construction of Coefficient Matrices

Constructing coefficient matrices for each system of linear equations is the first operation to be parallelized on the GPU architecture. This operation can be performed with a single GPU program, or kernel, for each type of equation, e.g each velocity component, pressure correction, second pressure correction. Since the matrix is sparse involving only coefficients for direct neighbor nodes memory on the device needs only to be allocated for neighboring coefficients, not for the full matrix. Access to field values at a local node and at direct neighbors is also required, and since global memory access on GPUs is their most important bottleneck, shared memory is used to store nodes per block in order to reduce the number of duplicate memory accesses.

### 5.2.2 Solving Systems of Linear Equations

The most important part of the implementation is the solution method used for solving systems of linear equations, since this operation is performed up to *2 + number of dimensions* times for each iteration. Normally (on a CPU), a type of preconditioned conjugate gradient (CG) method would the best choice for these linear solvers, but the CG method involves a matrix-vector multiplication and a vector-vector summation, which, compared to a linear solution method such as the Gauss-Seidel (GS) or successive over-relaxation (SOR) methods (with a single update per iteration), is much more expensive computationally on a GPU. The reason this is so much more expensive on a GPU is that GPUs do not handle random memory access very well, that is they are built and optimized for a uniform memory access strategy (coalesced memory access). For this reason, the SOR method is used for all linear solvers discussed in this chapter.

**Successive Over-Relaxation Method on GPUs.** The successive over-relaxation (SOR) method is an iterative method for solving linear systems of equations. It involves a single update per node in the system. The general algorithm is described in Algorithm 2, a full description of the method and the terms can be found in [16]. For the purpose of this chapter, we note that the algorithm is composed of an outer iteration loop for all nodes in the system with a single update to each node.

**Algorithm 2.** Successive over-relaxation algorithm

```
for iter=0:maxiter do
  for i = 2:(m+1) do
    for j = 2:(n+1) do
      u(i,j) = $\omega((a\_W(i,j)u(i-1,j)+a\_E(i,j)u(i+1,j)+a\_S(i,j)u(i,j-1) + a\_N(i,j)u(i,j+1) + b(i,j))/a\_P(i,j)) + (1-\omega)u(i,j)$
    end for
  end for
  if convergence then
    break
  end if
end for
```

The GPU implementation of the Gauss-Seidel (GS) or successive over-relaxation method (both methods are very similar and the terms will be used interchangeably in the rest of this chapter) is a type of domain decomposition of the numerical fluid system. The implementation is not a straightforward domain decomposition, however, it involves making two passes over the system per iteration, although each node is only updated once per iteration.

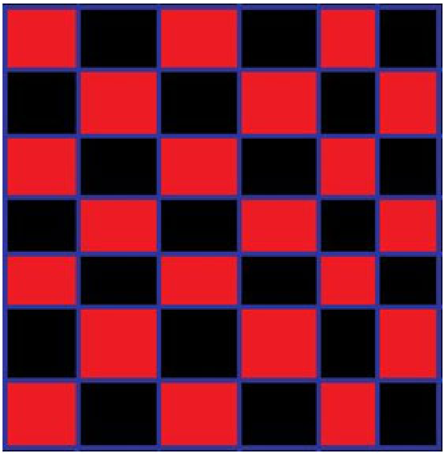


**Fig. 5** Red Black Nodes

Initially the method "colors" each node in the system by two alternating colors so that no node has neighboring nodes of the same color, such as in Figure 5. This coloring of nodes (two colors for a uniform two dimensional mesh) is why this parallel technique for the GS is also known as the red-black or the checkerboard method. Once each node is assigned a virtual color, we continue as we would in the sequential version of the method, with for one change: at each iteration there are two passes over the nodes, the first pass updates one set of colored nodes (the red nodes) and the second pass updates the second set of colored nodes (black nodes). Then we iterate as normal until convergence is reached. So far the parallel algorithm may look something like (in sequential form for now) Algorithm 3.

**Algorithm 3.** Parallel (Red-Black) Gauss-Seidel algorithm

```
for iter=0:maxiter do
   for i = all RED nodes do
      update u(i)
   end for
   for i = all BLACK nodes do
      update u(i)
   end for
   if convergence then
      break
   end if
end for
```

The advantage of this algorithm, in a parallel sense, is that all RED nodes can be updated simultaneously and all BLACK nodes can be updated simultaneously since from the sequential Algorithm 2 we know that only neighboring nodes are read during each node update. Since neighboring nodes will definitely not be updated at the same time (because of the different coloring), this allows us to perform updates on all nodes of the same color simultaneously.

Now that we have the general idea of the algorithm, we can move on to a more custom implementation for GPUs. First of all, we can map each node to a single thread. With the implementation up to this point, for a two dimensional system, five reads and one write per node update are required. The write is to the node that is mapped to the thread (the local node) and the reads are from the local node and its direct neighbors, as indicated by the white dots in Figure 6.

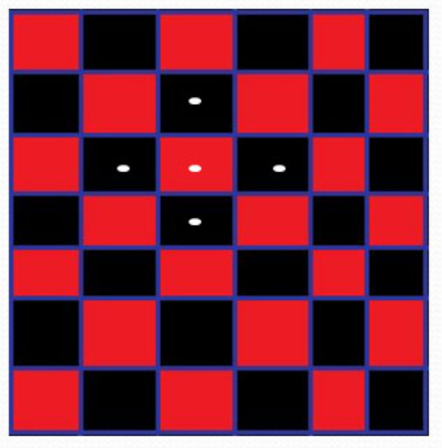


**Fig. 6** Red Black nodes with local and neighboring nodes

As of the algorithm developed so far, we use global GPU memory for the five reads per node update, which requires many duplicate reads per update since all neighboring nodes of a single local node are being read at least one more time and up to four more times per half iteration (per single color update pass). If we recall Section 4, the GPU programming model uses a set of blocks, where each block contains a set of threads, and each block has access to more efficient memory (called shared memory in the section above). If we make use of this shared memory per block we can remove nearly all of these duplicate reads by loading all nodes in a block into shared memory before we do the update. The set of nodes required for a block to update all of its associated threads (from the running example) are indicated by white lines in Figure 7. If we load all of these nodes into shared memory, including the ghost layer which is the layer of non local nodes (nodes that do not need to be updated by this current block) that surround the edges of the block,

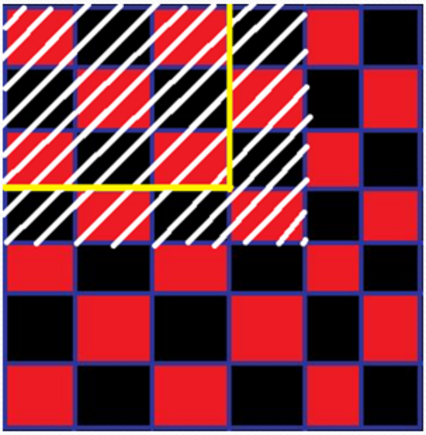


**Fig. 7** Red Black nodes that must be read per block (block is highlighted in yellow)

we can reduce the number of reads by a factor of almost four along with the memory access time for these reads since shared memory is much more efficient.

The pseudo code for the kernel (see Section 4) that is executed for each thread would then look something like Algorithm 4. Lines 2 to 15 load all local nodes and the ghost layer into shared memory. Line 17 uses the *__syncthreads()* function, which causes all threads in the block to wait at this location in the code until all of them have reached that point, this way all required data is loaded into shared memory before we start to do any updates (reads and writes) using this data.

### 5.2.3 Applying Corrections to Flow Fields

The application of the corrections to the flow fields is simply a kernel that applies the correction to each node. Since these corrections require only access to local field values at each node (as opposed to field values at neighbor nodes), a simple update per kernel is most efficient.

### 5.2.4 Convergence Check

Convergence of the PISO method can be determined in many different ways, depending on the application of the method. The most popular methods are a check of the velocity residual sum against a tolerance, or a check of the norm of pressure correction against a tolerance.

For both residual sum and norm calculations on the GPU we must perform a sum. This may seem simple but to efficiently do this on a GPU a little work is required. To do an efficient sum of a large vector on the GPU we do a parallel sum reduction. The method used in this work is defined in [28], and uses a tree based approach within each thread block, as illustrated in Figure 8. This algorithm works by assigning a uniform and contiguous subset of the vector to each thread block, each thread block then performs the sum of its associated subset and stores the result in the first memory location of its subset (denoted by the child node in the figure). It recursively does this until only one value is left (moved down the tree in the figure), which is the sum of the original vector. The time complexity of this technique is $O(N/\#Blocks + logN)$, vs $O(N)$ if we were to use a simple loop for summation.

**Algorithm 4.** GPU Gauss-Seidel algorithm

```
{Load local node into shared memory}
u_shared[s_i][s_j] = u[ij];
{check if on edge node, if yes then load ghost layer}
if threadIdx.x == 0 then
   u_shared[s_i-1][s_j] = u[i-1][j];
end if
if threadIdx.x == BLOCK_SIZE_X-1 then
   u_shared[s_i+1][s_j] = u[i+1][j];
end if
if threadIdx.y == 0 then
   u_shared[s_i][s_j-1] = u[i][j-1];
end if
if threadIdx.y == BLOCK_SIZE_Y-1 then
   u_shared[s_i][s_j+1] = u[i][j+1];
end if
{wait for all threads in block to finish loading shared memory}
__syncthreads();
for i,j = all RED or BLACK nodes only do
   update u[i][j]
end for
if convergence then
   break
end if
```

Advantages of this technique are not only that the majority of the computations are performed on the GPU but that the vector itself never needs to leave the GPU (which is preferred since all other calculations for the PISO method are on the GPU). Further, only one value needs to be copied from GPU memory to host memory. As we have noted in Section 4, copying from device to host memory is one of the most serious bottlenecks in any GPU implementation.

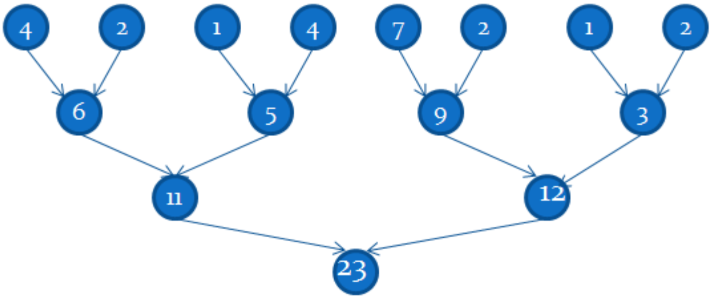


**Fig. 8** Parallel sum reduction using tree based approach within each thread block

## 6 Optimized Shape Design with SMCGP and CFD-GPU

Optimized shape design is the optimization of shapes in order to minimize and/or maximize specific parameters of the shape. The technique described throughout this chapter is an optimization of a shape to minimize drag and maximize lift within a fluid. The results of this experiment can be easily predicted to develop a shape that is more aerodynamically "smooth", such as that of an airfoil or hydrofoil at some optimal angle of attack.

The technique used to drive the optimization is genetic programming as described in Section 3. The fluid simulation technique described in Section 5 is used to evaluate fitness. Since the GP method could potentially require millions of evaluations in order to evolve an optimal solution we have to use this fluid simulation technique on a parallel architecture to increase performance. This is required to achieve an optimal solution in a practical period of time.

To tackle the optimized shape design problem we expect that about a million evaluations are required. Table 2 illustrates run time estimates for GP to converge with a $1024 \times 512$ discretized fluid system. As shown in this table, it is evident that to perform this shape optimization on a single CPU is very impractical (as it would require 10 years). But it requires only 10 days on a cluster of 50 (average) GPUs.[1]

Figure 9 illustrates a simple result of this technique for a partial evolution. This figure shows only a very small subset of the shapes as the evolution progresses in order to illustrate the effectiveness of the technique.

It is interesting to observe changes in the design of shapes during evolution. For example, Figure 10 shows one experimental run. In the initial population, a simple triangle is found, and this shape forms the basis for further evolution. At first evolution modifies the parameters of this shape, making it

[1] For example a nVidia GeFore 9800 GT, with 120 cores and 1 GB of memory produces about 336 GFLOPS, is average at the time of this writing.

**Table 2** Optimized Shape Design Problem: Estimated Times

| Hardware | GP Convergence Time |
|---|---|
| 1 CPU | 10 years |
| 50 CPU Cluster | 70 days |
| 1 GPU | 1.4 years |
| 50 GPU Cluster | 10 days |

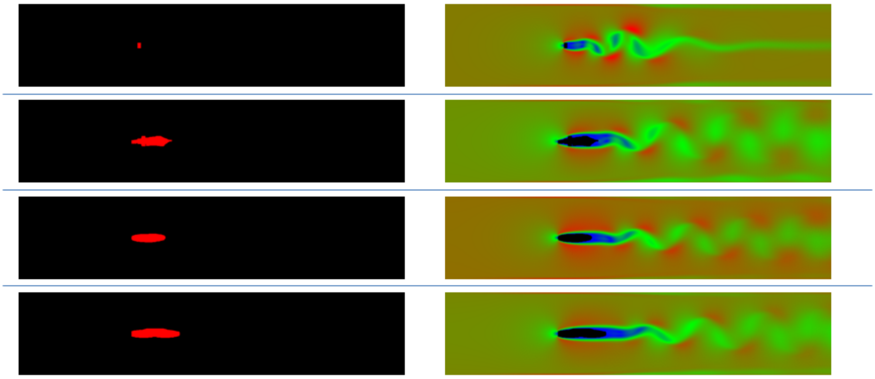


**Fig. 9** Shape evolution for drag minimization

more angled at the front to reduce drag. Next evolution introduces a small spike on the top of the shape, which will alter the flow over the top of the surface. Eventually, this spike is smoothed into a small lump, that will have less drag. This shape then further changes into larger, rounder shape which encompasses the entire front of the triangle further reducing drag. Finally, the evolution approximates a shape very similar to the familiar shape of a wing cross-section.

## *6.1 Measuring Fitness Using CFD*

The fitness function uses the CFD simulation to obtain drag and lift coefficient for a design. The fitness score is the absolute lift minus the absolute drag. Hence, fitness scores above zero represent objects with more lift than drag, and therefore indicate that a lifting body has been evolved. Here we also look for shapes that have a minimal size. In order to enforce this, we insert a block into the environment that the shapes must form around. The block has zero lift and a large drag. To get a good fitness score, this block must somehow be incorporated into the design.

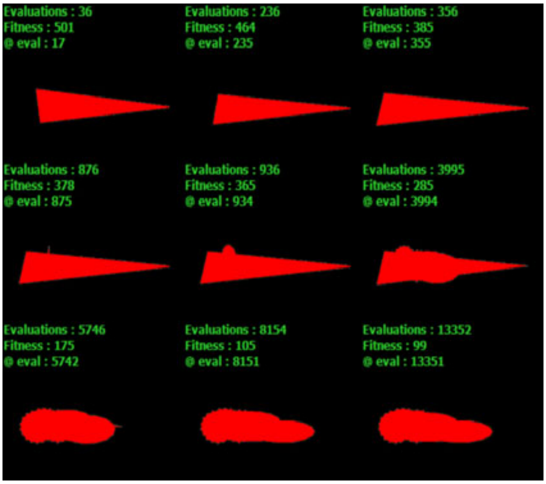


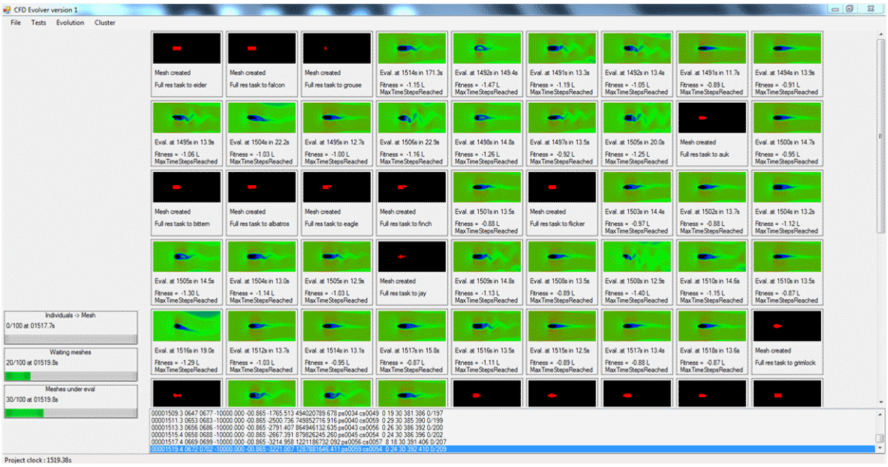
**Fig. 10** Sequence showing the best shape at different times within an evolutionary experiment.

For the fitness function to work correctly, shape designs that will not simulate correctly are discarded. In particular, very small and very large structures are not tested. We also currently discard non-contiguous shapes which, if we were to construct the shape, would mean it could be fashioned from a single piece.

## *6.2 Evolutionary Algorithm*

The actual evolutionary algorithm used was also parallel, and distributed in nature. As the evaluation time of individuals would be different (dependent not only on the convergence properties of the simulation, but also on the computing hardware used), to work efficiently, the algorithm also has to work in an asynchronous and non-blocking way, so that all available computing resources are always helping with the search. The evolutionary algorithm is based on [13], as this was found to work efficiently in an asynchronous environment. A central population of individuals is stored on the root computer, and when a client node finishes processing an individual it is returned to this population. When a client requires a new individual to process, individuals are selected from this population and crossover/mutation applied to produce a new individual for evaluation. As this shared population increases in time as evaluated individuals are added, it periodically needs to be reduced in size. In [13] this dynamic of a variable population size was found to be beneficial to evolution. Figure 11 shows an example of the algorithm running.

The most significant parameters are shown in Table 3.



Mesh created
Full res task to eider

Mesh created
Full res task to falcon

Mesh created
Full res task to grouse

Eval. at 1514s in 171.3s
Fitness = -1.15 L
MaxTimeStepsReached

Eval. at 1492s in 149.4s
Fitness = -1.47 L
MaxTimeStepsReached

Eval. at 1491s in 13.3s
Fitness = -1.19 L
MaxTimeStepsReached

Eval. at 1492s in 13.4s
Fitness = -1.05 L
MaxTimeStepsReached

Eval. at 1491s in 11.7s
Fitness = -0.89 L
MaxTimeStepsReached

Eval. at 1494s in 13.9s
Fitness = -0.91 L
MaxTimeStepsReached

Eval. at 1495s in 13.9s
Fitness = -1.06 L
MaxTimeStepsReached

Eval. at 1504s in 22.2s
Fitness = -1.03 L
MaxTimeStepsReached

Eval. at 1495s in 12.7s
Fitness = -1.00 L
MaxTimeStepsReached

Eval. at 1506s in 22.9s
Fitness = -1.16 L
MaxTimeStepsReached

Eval. at 1498s in 14.8s
Fitness = -1.26 L
MaxTimeStepsReached

Eval. at 1497s in 13.5s
Fitness = -0.92 L
MaxTimeStepsReached

Eval. at 1505s in 20.0s
Fitness = -1.25 L
MaxTimeStepsReached

Mesh created
Full res task to auk

Eval. at 1500s in 14.7s
Fitness = -0.95 L
MaxTimeStepsReached

Mesh created
Full res task to bittern

Mesh created
Full res task to albatros

Mesh created
Full res task to eagle

Mesh created
Full res task to finch

Eval. at 1501s in 13.5s
Fitness = -0.88 L
MaxTimeStepsReached

Mesh created
Full res task to flicker

Eval. at 1503s in 14.4s
Fitness = -0.97 L
MaxTimeStepsReached

Eval. at 1502s in 13.7s
Fitness = -0.88 L
MaxTimeStepsReached

Eval. at 1504s in 13.2s
Fitness = -1.12 L
MaxTimeStepsReached

Eval. at 1505s in 14.5s
Fitness = -1.30 L
MaxTimeStepsReached

Eval. at 1504s in 13.0s
Fitness = -1.14 L
MaxTimeStepsReached

Eval. at 1505s in 12.9s
Fitness = -1.03 L
MaxTimeStepsReached

Mesh created
Full res task to jay

Eval. at 1509s in 14.8s
Fitness = -1.13 L
MaxTimeStepsReached

Eval. at 1508s in 13.5s
Fitness = -0.89 L
MaxTimeStepsReached

Eval. at 1508s in 12.9s
Fitness = -1.40 L
MaxTimeStepsReached

Eval. at 1510s in 14.6s
Fitness = -1.15 L
MaxTimeStepsReached

Eval. at 1510s in 13.5s
Fitness = -0.87 L
MaxTimeStepsReached

Eval. at 1516s in 19.0s
Fitness = -1.29 L
MaxTimeStepsReached

Eval. at 1512s in 13.7s
Fitness = -1.03 L
MaxTimeStepsReached

Eval. at 1514s in 13.1s
Fitness = -0.95 L
MaxTimeStepsReached

Eval. at 1517s in 15.8s
Fitness = -0.87 L
MaxTimeStepsReached

Eval. at 1516s in 13.5s
Fitness = -1.11 L
MaxTimeStepsReached

Eval. at 1515s in 12.5s
Fitness = -0.89 L
MaxTimeStepsReached

Eval. at 1517s in 13.4s
Fitness = -0.88 L
MaxTimeStepsReached

Eval. at 1518s in 13.6s
Fitness = -0.87 L
MaxTimeStepsReached

Mesh created
Full res task to grimlock

Individuals -> Mesh
0/100 at 01517.7s

Waiting meshes
20/100 at 01519.8s

Meshes under eval
30/100 at 01519.8s

```
00001509.3 0647 0677 -10000.000 -00.865 -1765.513 494020789.678 ps0034 cs0049  0 19 30 381 386 0/197
00001511.3 0653 0683 -10000.000 -00.865 -2500.736 749852716.916 ps0040 cs0059  0 29 30 385 390 0/199
00001513.3 0656 0686 -10000.000 -00.865 -2791.407 864946132.635 ps0043 cs0056  0 26 30 386 392 0/200
00001515.4 0658 0688 -10000.000 -00.865 -2667.391 879826245.260 ps0045 cs0054  0 24 30 386 396 0/202
00001517.4 0669 0699 -10000.000 -00.865 -3214.958 1221186732.092 ps0056 cs0057  8 18 30 391 406 0/207
00001519.4 0672 0702 -10000.000 -00.865 -3221.007 1287881646.411 ps0059 cs0054  0 24 30 392 410 0/209
```

Project clock : 1519.38s

**Fig. 11** Screen shot of the shape evolver. The small tiles show either an individual that is currently being processed, or the result of a simulation for an individual.

**Table 3** Parameters of the evolutionary algorithm.

| Parameter | Value |
|---|---|
| Initial genotype length | 20 nodes |
| Mutation rate | 0.05 |
| Initial Population Size | 50 |
| Frequency of population resizes | 60 seconds |

## 7 Conclusions

This chapter described a technique inspired by natural evolution and applied to a shape optimization design problem. The evolutionary technique itself was developed as a parallel algorithm for a distributed system, along with the fitness evaluation on a GPU parallel architecture. A focus on efficient algorithm design is necessary since fluid simulations are a very computationally expensive task, while the evolutionary algorithm discussed would require on the order of millions of evaluations of fluid simulations.

The fluid dynamics solver described in this chapter applied a popular iterative method for solving the pressure-coupled governing fluid flow equations adapted to the GPU to allow for faster evaluation times. The adaptation to the GPU required several parallel optimization steps many of which are general purpose optimizations that can be extended to other problems to be solved on GPUs.

The optimized shape design method described in this chapter also fulfills the requirement of having the capability to produce fully general shapes, where there are minimal constraints on the design. This minimal constraint design results from the SMCGP method allowing any combination of shapes to produce the final result. This requirement is an advantage in design optimization since it allows for new designs that may not have been expected, and may not have been possible with an otherwise more constrained approach.

**Acknowledgements.** The authors would also like to thank Dr. Taras Kowaliw for suggesting the use of the 'super function' for shape encoding. W.B. acknowledges funding by NSERC under the Discovery Grant Program, RGPIN 283304-07. S.H. was supported by a postdoctoral grant from the Atlantic Canada Computational Excellence Network (ACENET).

## References

1. Asouti, V.G., Giannakoglou, K.C.: Aerodynamic optimization using a parallel asynchronous evolutionary algorithm controlled by strongly interacting demes. Engineering Optimization 41(3), 241 (2009)
2. Banzhaf, W., Miller, J.: The challenge of complexity. In: Menon, A. (ed.) Frontiers of Evolutionary Computation, pp. 243–260. Springer (2004)
3. Billings, D.: PDE Nozzle Optimization Using a Genetic Algorithm. Technical report. Marshall Space Flight Center (2000)
4. Giannakoglou, K., Papadimitriou, D., Kampolis, I.: Aerodynamic shape design using evolutionary algorithms and new gradient-assisted metamodels. Computer Methods in Applied Mechanics and Engineering 195(44-47), 6312–6329 (2006)
5. Gielis, J.: A generic geometric transformation that unifies a wide range of natural and abstract shapes. Am. J. Bot. 90(3), 333–338 (2003)
6. Harding, S., Banzhaf, W., Miller, J.F.: A survey of self modifying cartesian genetic programming. In: Riolo, R., McConaghy, T., Vladislavleva, E. (eds.) Genetic Programming Theory and Practice VIII. Genetic and Evolutionary Computation, 20-22 May, ch. 6, vol. 8, pp. 91–107. Springer, Ann Arbor (2010)
7. Harding, S., Miller, J., Banzhaf, W.: Self Modifying Cartesian Genetic Programming: Fibonacci, Squares, Regression and Summing. In: Vanneschi, L., Gustafson, S., Moraglio, A., De Falco, I., Ebner, M. (eds.) EuroGP 2009. LNCS, vol. 5481, pp. 133–144. Springer, Heidelberg (2009)
8. Harding, S., Miller, J.F., Banzhaf, W.: Evolution, development and learning with self modifying cartesian genetic programming. In: GECCO 2009: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, pp. 699–706. ACM Press, New York (2009)
9. Harding, S., Miller, J.F., Banzhaf, W.: Self modifying cartesian genetic programming: Parity. In: Tyrrell, A. (ed.) 2009 IEEE Congress on Evolutionary Computation, Trondheim, Norway, May 18-21, pp. 285–292. IEEE Computational Intelligence Society, IEEE Press (2009)

10. Harding, S., Miller, J.F., Banzhaf, W.: Developments in cartesian genetic programming: self-modifying CGP. Genetic Programming and Evolvable Machines 11(3/4), 397–439 (2010); Tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines
11. Harding, S., Miller, J.F., Banzhaf, W.: Self modifying cartesian genetic programming: finding algorithms that calculate pi and e to arbitrary precision. In: Branke, J., Pelikan, M., Alba, E., Arnold, D.V., Bongard, J., Brabazon, A., Branke, J., Butz, M.V., Clune, J., Cohen, M., Deb, K., Engelbrecht, A.P., Krasnogor, N., Miller, J.F., O'Neill, M., Sastry, K., Thierens, D., van Hemert, J., Vanneschi, L., Witt, C. (eds.) GECCO 2010: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, Portland, Oregon, USA, 7-11 July, pp. 579–586. ACM (2010)
12. Harding, S., Miller, J.F., Banzhaf, W.: SMCGP2: finding algorithms that approximate numerical constants using quaternions and complex numbers. In: Krasnogor, N., Lanzi, P.L., Engelbrecht, A., Pelta, D., Gershenson, C., Squillero, G., Freitas, A., Ritchie, M., Preuss, M., Gagne, C., Ong, Y.S., Raidl, G., Gallager, M., Lozano, J., Coello-Coello, C., Silva, D.L., Hansen, N., Meyer-Nieberg, S., Smith, J., Eiben, G., Bernado-Mansilla, E., Browne, W., Spector, L., Yu, T., Clune, J., Hornby, G., Wong, M.-L., Collet, P., Gustafson, S., Watson, J.-P., Sipper, M., Poulding, S., Ochoa, G., Schoenauer, M., Witt, C., Auger, A. (eds.) GECCO 2011: Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation, Dublin, Ireland, July 12-16, pp. 197–198. ACM (2011)
13. Hu, T., Harding, S., Banzhaf, W.: Variable population size and evolution acceleration: a case study with a parallel evolutionary algorithm. Genetic Programming and Evolvable Machines 11(2), 205–225 (2010)
14. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
15. Kumar, S., Bentley, P.: On Growth, Form and Computers. Academic Press (2003)
16. LeVeque, R.J.: Finite Difference Methods for Ordinary and Partial Differential Equations. In: Society for Industry and Applied Mathematics Proceedings (2007)
17. Miller, J., Banzhaf, W.: Evolving the program for a cell: from french flags to boolean circuits. In: Kumar, S., Bentley, P. (eds.) On Growth, Form and Computers, pp. 278–301. Academic Press, London (2003)
18. Miller, J.F.: An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In: Proceedings of the 1999 Genetic and Evolutionary Computation Conference (GECCO), Orlando, Florida, pp. 1135–1142. Morgan Kaufmann (1999)
19. Miller, J.F.: Evolving Developmental Programs for Adaptation, Morphogenesis, and Self-Repair. In: Banzhaf, W., Ziegler, J., Christaller, T., Dittrich, P., Kim, J.T. (eds.) ECAL 2003. LNCS (LNAI), vol. 2801, pp. 256–265. Springer, Heidelberg (2003)
20. Miller, J.F., Smith, S.L.: Redundancy and computational efficiency in cartesian genetic programming. IEEE Transactions on Evoluationary Computation 10, 167–174 (2006)
21. nVidia: CUDA Programming Guide. nVidia Corporation, Version 2.3 (2009)

22. Obayashi, S., Tsukahara, T., Nakamura, T.: Multiobjective genetic algorithm applied to aerodynamic design of cascade airfoils. IEEE Transactions on Industrial Electronics 47(1), 211–216 (2000)
23. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming (2008) (With contributions by Koza, J.R.), Published via, `http://lulu.com`, freely Available at, `http://www.gp-field-guide.org.uk`
24. Poloni, C., Giurgevich, A., Onesti, L., Pediroda, V.: Hybridization of a multiobjective genetic algorithm, a neural network and a classical optimizer for a complex design problem in fluid dynamics. Computer Methods in Applied Mechanics and Engineering 186(2-4), 403–420 (2000)
25. Rechenberg, I.: Evolutionsstrategie: optimierung technischer systeme nach prinzipien der biologischen evolution. Frommann-Holzboog (1973)
26. Rechenberg, I.: Case studies in evolutionary experimentation and computation. Computer Methods in Applied Mechanics and Engineering 186(2-4), 125–140 (2000)
27. Schwefel, H.-P.: Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie. Mit einer vergleichenden Einführung in die Hill-Climbing- und Zufallsstrategien. Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik 60 (1977)
28. Sengupta, Harris, Garland: Efficient parallel scan algorithms for gpus. nVidia Technical Report NVR-2008-003. nVidia Corporation (2008)

# Characterizing Fault-Tolerance in Evolutionary Algorithms

Daniel Lombraña González, Juan Luis Jiménez Laredo,
Francisco Fernández de Vega, and Juan Julián Merelo Guervós

**Abstract.** This chapter presents a study of the fault-tolerant nature of some of the best known Evolutionary Algorithms, namely Genetic Algorithms (GAs) and Genetic Programming (GP), on a real-world Desktop Grid System. We study the situation when no fault-tolerance mechanisms is employed. The results show that when parallel GAs and GPs are run on non-reliable distributed infrastructures -thus suffering degradation of available hardware- they can achieve results of a similar quality when compared with a failure-free platform in three of the six scenarios under study. Additionally, we show that increasing the initial population size is a successful method to provide resilience to system failures in five of the scenarios. Such results suggest that Parallel GAs and GPs are inherently and naturally fault-tolerant.

**Keywords:** genetic programming, genetic algorithms, evolutionary algorithms, fault tolerance.

## 1 Introduction

Genetic Algorithms (GAs) and Genetic Programming (GP) are well known representatives of Evolutionary Algorithms (EAs), frequently used to solve optimization

Daniel Lombraña González
Citizen Cyberscience Centre, CERN, Switzerland
e-mail: teleyinex@gmail.com

Juan Luis Jiménez Laredo · Juan Julián Merelo Guervós
University of Granada
e-mail: {juanlu,jmerelo}@geneura.ugr.es

Francisco Fernández de Vega
Centro Universitario de Mérida, Universidad de Extremadura. Sta. Teresa Jornet,
38. 06800 Mérida (Badajoz), Spain
e-mail: fcofdez@unex.es

F.F. de Vega et al. (Eds.): Parallel Architectures & Bioinspired Algorithms, SCI 415, pp. 77–99.
springerlink.com

problems. Both require a large amount of computing resources when the problem faced is complex. The more complex the problem, the larger the computing requirements. This fact leads to a sometimes prohibitively long time to solution that happens, for example, when tackling real-world problems. In order to reduce the execution time of EAs, researchers have applied parallel and distributed programming techniques during the last decades.

There are two main advantages in exploiting the inherent parallelism of EAs: (i) the computing load is distributed among different processors, which improves the execution time, and (ii) the algorithm itself may suffer of structural changes allowing to outperform the sequential counterpart (see for instance [51]).

Parallel algorithms, and thus parallel GAs and GP, must be run on platforms that consists of multiple computing elements or *processors*. Although supercomputers can be employed, usually commodity clusters and distributed systems are used instead, due to both good performance and cheaper prices. One of the most popular distributed systems nowadays are the Desktop Grid Systems (DGSs). The term "desktop grid" is used to refer to distributed networks of heterogeneous single systems that contribute idle processor cycles for computing.

Perhaps the most well known desktop grid system is the Berkeley Open Infrastructure for Network Computing (BOINC) [4], which supports among other projects the successful Einstein@Home [29] . DGSs are also known as *volunteer grids* because they aggregate the computing resources (commodity computers from offices or homes) that volunteers worldwide willingly donate to different research projects (such as Einstein@Home).

One of the most important features of DGSs is that they provide large-scale parallel computing capabilities, only for specific types of applications –bag of tasks mainly–, at a very low cost. Therefore DGSs can provide parallel computing capabilities for running demanding parallel applications, which is frequently the case of EAs. A good example of the combination of PEAs and DGSs is the Milky-Way@Home project [13].

But with large scale comes a higher likelihood that processors suffer a failure [44], interrupting the execution of the algorithm or crashing the whole system (in this chapter we use the term "failure" and do not make the subtle distinction between "failure" and "fault", which is not necessary for our purpose). Such an issue is characteristic of DGSs: computers join the system, contribute some resources and leave it afterwards causing a collective effect known as churn [46]. Churn is an inherent property of DGSs and has to be taken into account in the design of applications, as these interruptions (computer powered off, busy CPU, etc.) are interpreted by the application as a failure.

To cope with failures, researchers have studied and developed different mechanisms to circumvent the failures or restore the system once a failure occurs. These techniques are known as *Fault-Tolerance mechanisms* and enforce that an application behave in a well-defined manner when a failure occurs [20]. Nevertheless, not many efforts have been applied to study the fault tolerance features of PEAs in general, and of PGAs and PGP in particular.

In previous works [36, 27] we firstly analyzed the fault-tolerance nature of Parallel Genetic Programming (PGP) under several simplified assumptions. These initial results suggested that PGP exhibits a fault-tolerant behavior by default, encouraging to go a step further and run PGP on large-scale computing infrastructures that are subject to failures without requiring the employment of any fault-tolerance mechanism. This work was lately improved [22, 23, 25] by studying the fault-tolerance nature of PGP and PGAs using real data from one of the most high churn distributed systems: the Desktop Grids. The results again showed that PGP and PGAs can cope with failures without using any fault-tolerance mechanism, concluding that PGP and PGAs are fault tolerant by nature since it implements by default the fault-tolerance mechanism called *graceful degradation* [21].

This chapter is a summary of the main results obtained for PGAs and PGP regarding the study of fault-tolerance and their intrinsic fault-tolerant nature. To this aim, we have chosen a fine-grained master-worker model of parallelization [51]. A server, “the master”, runs the main algorithm and hosts the whole population. The server is in charge of sending non-evaluated individuals to workers in order to obtain their fitness values. This approach is effective because one of the most time-consuming steps of GAs or GP is the evaluation –fitness computation– phase. The master waits until all individuals in generation $n$ are evaluated before going to the next generation $n+1$ and run the genetic operations.

We assume that the system only suffers from omission failures [21]:

- the master sends $N$ individuals with $N > 0$ to a worker, and the worker never receives them, e.g., due to network transmission problems; or
- the master sends $N$ individuals with $N > 0$ to a worker, the worker receives them but never returns them. This can occur because the worker crashes or the returned individuals are lost during the transmission.

In order to study the behavior of PGAs and PGP under the previous assumptions, we are going to simulate the failures using real-world traces of host availability from three DGSs. We have chosen Desktop Grid availability data because these systems exhibit large amounts of failures, and thus if it is possible to run inside them PGAs or PGP without using any fault-tolerance mechanism, PGAs and PGP will be able to exploit any parallel or distributed systems to its maximums.

The rest of the chapter is organized as follows. Section 2 reviews related work; section 3 describes main fault tolerance techniques. Section 4 presents the setup of the different scenarios and experiments; section 5 shows the obtained results and their analysis; and, finally, section 6 concludes the chapter with a discussion of the results and future directions.

## 2 Background and Related Work

When using EAs to solve real-world problems researchers and practitioners often face prohibitively long times-to-solution on a single computer. For instance, Trujillo *et al.* required more than 24 hours to solve a computer vision problem [53], and times-to-solution can be much longer, measured in weeks or even months. Consequently, several researchers have studied the application of parallel computing to Spatially Structured EAs in order to shorten times-to-solution [17, 51, 9]. Such PEAs have been used for decades, for instance, on the Transputer platform [6], or, more recently, via software frameworks such as Beagle [19], grid based tools like Paradiseo [41], or BOINC-based EA frameworks for execution on DGSs [38].

Failures in a distributed system can be local, affecting only a single processor, or they can be communication failures, affecting a large number of participating processors. Such failures can disrupt a running application, for instance imposing the application to be restarted from scratch. As distributed computing platforms become larger and/or lower-cost through the use of less reliable or non-dedicated hardware, failures occur with higher probability [43, 45, 54]. Failures are, in fact, the common case in DGSs. For this reason, fault-tolerant techniques are necessary so that parallel applications in general, and in our case PEAs, can benefit from large-scale distributed computing platforms. Failures can be alleviated, and in some cases completely circumvented, using techniques such as checkpointing [15], redundancy [26], long-term-memory [28], specific solutions to message-passing [2] or rejuvenation frameworks [48]. It is necessary to embed the techniques in the application and the algorithms. While some of these techniques may be straightforward to implement (e.g., failure detection or restart from scratch), the most common ones typically lead to an increase in software complexity. Regardless, fault tolerance techniques always requires extra computing resources and/or time.

Currently, available PEA frameworks employ fault tolerant mechanisms to tolerate failures in distributed systems such as DGSs. For instance ECJ [40], ParadisEO [8], DREAM [7] or Distributed Beagle [19]. These frameworks have distinct features (programming language, parallelism models, etc.) that may be considered in combination with DGSs, and provide different techniques to cope with failures:

- ECJ [40] is a Java framework that employs a master-worker scheme to run PEAs using TCP/IP sockets. When a remote worker fails, ECJ handles this failure by rescheduling and restarting the computation to another available worker.
- ParadisEO [8] is a C++ framework for running a master-worker model using MPI [18], PVM [47], or POSIX threads. Initially, ParadisEO did not provide any fault-tolerance. Later on, developers implemented a new version on top of the Condor-PVM resource manager [42] in order to provide a checkpointing feature [15]. This framework, however, is not the best choice for DGSs because these systems are: (i) loosely coupled and (ii) workers may be behind proxies, firewalls, etc. making it difficult to deploy a ParadisEO system.
- DREAM [7] is a Java Peer-to-Peer (P2P) framework for PEAs that provides a fault-tolerance mechanism called *long-term-memory* [28]. This framework is designed specifically for P2P systems. As a result, it cannot be compared directly with our work since we focus on a master-worker architecture on DGSs.
- Distributed BEAGLE [19] is a C++ framework that implements the master-worker model using TCP/IP sockets as ECJ. Fault-tolerance is provided via a simple time-out mechanism: a computation is re-sent to one or more new

available workers if this computation has not been completed by its assigned worker after a specified deadline.

While these PEA frameworks provide fault-tolerant features, the relationship between fault tolerance and specific features of PEAs has not been studied.

So far, EA researchers have not employed massively DGSs. Nevertheless, there are several projects using DGSs like the MilkyWay@Home project [13] which uses GAs to create an accurate 3D model of the Milky way, a ported version of LilGP [11](a framework for GP [14]) to one of the most employed DGSs, BOINC [4], or the *custom execution environment* facility proposed and implemented by Lombraña et. al. in [37, 24] for BOINC.

Other EA researchers have focused their attention on P2P systems [35], which are very similar to DGSs because the computing elements are also desktop computers in its majority. However these systems are different because there is not a central server as in DGSs.

In all the described proposals –to the best of our knowledge– none of them have specifically addressed the problem of failures within PGAs or PGP. Nevertheless, some of those solutions internally employ some fault-tolerance mechanisms. In this sense, only Laredo et al. have analyze the resilience to failures of a parallel Genetic Algorithm in [34], following the Weibull degradation of a P2P system (failures are the host-churn behavior of these systems as well as DGSs) proposed by Stutzbach and Rejaie in [46]. Therefore, PGAs or PGP have not been analyzed before under real host availability traces (a.k.a. host-churn). Hence, this chapter assesses fault tolerance in PGAs and PGP using host-churn data collected in three real-world DGSs [30]. Therefore, the key contribution of this chapter is the full characterization of PGAs and PGP from the point of view of fault-tolerance with the aim of studying if PGAs can be run in parallel or distributed systems without using any fault-tolerance mechanism.

## 3 Fault Tolerance

*Fault tolerance* can be defined as the ability of a system to behave in a well-defined manner once a failure occurs. In this chapter we only take into account failures at the process level. A complete description of failures in distributed systems is beyond the scope of our discussion. In this section, we describe different failure models as well as different techniques to circumvent failures.

### *3.1 Failure Models*

According to Ghosh [21], failures can be classified as follows: crash, omission, transient, Byzantine, software, temporal, or security failures. However, in practice, any system may experience a failure due to the following reasons [21]: (i) *Transient failures*: the system state can be corrupted in an unpredictable way; (ii) *Topology* *changes*: the system topology changes at runtime when a host crashes, or a new host is added; and (iii) *Environmental changes*: the environment – external variables that should only be read – may change without notice. Once a failure has occurred, a mechanism is required to bring back the system into a valid state. There are four major types of such fault tolerance mechanisms: masking tolerance, non-masking tolerance, fail-safe tolerance, and graceful degradation [21].

To discuss fault-tolerance in the context of PEAs, we first need to specify the way in which the GP or GA application is parallelized. Parallelism has been traditionally applied to GP and GAs at two possible levels: the individual level or the population level [51, 9, 50, 3]. At the individual level, it is common to use a master-worker scheme, while at the population level, a.k.a. the "island model", different schemes can be employed (ring, multi-dimensional grids, etc.).

In light of previous studies [51, 50] and taking into account the specific parallel features of DGSs [31, 30], we focus on parallelization at the individual level. In fact, DGSs are loosely-coupled platforms with volatile resources, and therefore ideally suited to and widely used for embarrassingly parallel master-worker applications. Furthermore parallelization at the individual level is popular in practice because it is easy to implement and does not require any modification of the evolutionary algorithm [9, 50, 3].

The server, or "master", is in charge of running the main algorithm and manages the whole population. It sends non evaluated individuals to different processes, the "workers," that are running on hosts in the distributed system. This model is effective as the most expensive and time-consuming operation of the application is typically the individual evaluation phase. The master waits until all individuals in generation $n$ are evaluated before generating individuals for generation $n+1$. In this scenario, the following failures may occur:

- *A crash failure* – The master crashes and the whole execution fails. This is the worst case.
- *An omission failure* – One or more workers do not receive the individuals to be evaluated, or the master does not receive the evaluated individuals.
- *A transient failure* – A power surge or lighting affects the master or worker program, stopping or affecting the execution.
- *A software failure* – The code has a bug and the execution is stopped either on the master or on the worker(s).

We make the following assumptions: (i) we consider all the possible failures that can occur during the transmission and reception of individuals between the master and each worker, but we assume that all software is bug-free and that there are no transient failures; (ii) the master is always in a safe state and there is no need for master fault tolerance (unlike for the workers, which are untrusted computing processes). This second assumption is justified because the master is under a single organization/person's control, and, besides, known fault tolerance techniques (e.g., primary backup [26]) could easily be used to tolerate master failures.

Our system only suffers from omission failures: (i) the master sends $N > 0$ individuals to a worker, and the worker never receives them (e.g., due to network transmission problems); or (ii) the master sends $N > 0$ individuals to a worker, the worker receives them but never returns them (e.g., due to a worker crash or to network transmission problems).

## 3.2 Fault-Tolerant and Non-Fault-Tolerant Strategies

Since our objective in this work is to study the implicit fault-tolerant nature of the PEA paradigm, we need to perform comparison with the use of a reasonable and explicit fault-tolerant strategy. In the master-worker scheme, four typical approaches can be applied to cope with failures:

1. Restart the computation from scratch on another host after a failure.
2. Checkpoint locally (with some overhead) and restart the computation on the same host from the latest checkpoint after a failure.
3. Checkpoint on a checkpointing server (with more overhead) and move to another host after a failure, restarting the computation from the last checkpoint.
4. Use task replication by sending the same individual to two or more hosts, each of them performing either 1, 2, or perhaps even 3 above. The hope is that one of the replicas will finish early, possibly without any failure.

Based on the analysis in Section 2 of existing PEA frameworks that are relevant in the context of DGSs, namely ECJ and Distributed Beagle, the common technique to cope with failures is the first one: re-send lost individuals after detecting the failure. The advantage of this technique is that it is low overhead, very simple to implement, and reasonably effective. More specifically, its modus-operandi is as follows:

1. After assigning individuals to workers, the master waits at most $T$ time-units per generation. If all individuals have been computed by workers before $T$ time-units have elapsed, then the master computes fitness values, updates the population, and proceeds with the next generation.
2. If after $T$ time-units some individuals have not been evaluated, then the master assumes that workers have failed or are simply so slow that they may not be useful to the application. In this case:

    a. individuals that have not been evaluated are resent for evaluation to available workers, and the master waits for another $T$ time-units for these individuals to be evaluated.
    b. If there are not enough available workers to evaluate all unevaluated individuals, then the master proceeds in multiple phases of duration $T$. For instance, if after the initial period of $T$ time-units there remain 5 unevaluated individuals and there remain only 2 available workers, the master will use $\lceil \frac{5}{2} \rceil = 3$ phases (assuming that all future individual evaluations are successful).

This method provides a simple fault-tolerant mechanism for handling worker failures as well as slow workers, which is a common problem in DGSs due to high levels of host heterogeneity [4, 21, 5]. For the sake of simplicity, we make the assumption that individuals that are lost and resent for evaluation to new workers are always evaluated successfully. This is unrealistic since future failures could lead to many phases of resends. However, this assumption represents a best-case scenario for the fault-tolerant strategy. The difference between the failure-free and the failure-prone case is the extra time due to resending individuals. In the failure-free case, with $G$ generations, the execution time should be $T_{executiontime} = G \times T$, while in a failure-prone case it will be higher.

By contrast with this fault-tolerant mechanism, we propose a simple non-fault-tolerant approach that consists in ignoring lost individuals, considering their loss just a kind of dynamic population feature [52, 16, 39, 32]. In this approach the master does not attempt to detect failures and no fault tolerance technique is used. The master waits a time $T$ per generation, and proceeds to the next generation with the available individuals at that time, likely losing individuals at each generation. The hope is that the loss of individuals is not (significantly) detrimental to the achieved results, while the overhead of resending lost individuals for recomputation is not incurred.

## 4 Experimental Methodology

All the experiments presented in this chapter are based on simulations. Simulations allow us to perform a statistically significant number of experiments in a wide range of realistic scenarios. Furthermore, our experiments are repeatable, via "reproduction" of host availability trace data collected from real-world DG platforms [30], so that fair comparisons between simulated experiments are possible.

### *4.1 Experiments and Failure Model*

We perform experiments for a GA and a GP well-known problems. The GP problem is the even parity 5 (EP5) which tries to build a program capable of calculating the parity of a set of 5 bits. In the case of the GA problem, we use a 3-trap instance [1] which is a piecewise-linear function defined on unitation (the number of ones in a binary string).

In every case, two kind of experiments are carried out:

1. for the failure-free case (i.e. assuming no worker failures occur)
2. reproducing and simulating failure traces from real-world DGSs.

In the failure free case the available number of computing nodes is kept steady throughout the execution, while in the second case the number of nodes vary along the generations.

The simulation of host availability in the DG is performed based on three real-world traces of host availability that were measured and reported in [30]: *ucb*, *entrfin*, and *xwtr*. These traces are time-stamped observations of the host availability in three DGSs. The *ucb* trace was collected for 85 hosts in a graduate student lab in the EE/CS Department at UC Berkeley for about 1.5 months. The *entrfin* trace was collected for 275 hosts at the San Diego Supercomputer Center for about 1 month. The *xwtr* trace was collected for 100 hosts at the Université Paris-Sud for about 1 month. See [30] for full details on the measurement methods and the traces, and Table 1 for a summary of its main features.

**Table 1** Features of Desktop Grid Traces

| Trace | Hosts | Time in months | Place |
|---|---|---|---|
| *entrfin* | 275 | 1 | SD Supercomputer Center |
| *ucb* | 85 | 1.5 | UC Berkeley |
| *xwtr* | 100 | 1 | Université Paris-Sud |

Figure 1 shows an example of available data from the *ucb* trace: the number of available hosts in the platform during 24 hours time. The figure depicts the typical churn phenomenon, with available hosts becoming unavailable and later becoming available again. Experiments were performed over such 24-hour segments.

In addition, we use two different scenarios when simulating host failures based on trace data. In the first scenario a stringent assumption is used: hosts that become unavailable never become available again (i.e. the system degrades). An example is shown in Figure 1, as the curve "trace without return." In such an scenario, we have selected as a starting point policy the moment in which a largest number of hosts are available. In the second scenario hosts can become available again after a failure and reused by the application. This phenomenon is called "churn," and is inherent to real-world DG systems. In this case, application execution starts at an arbitrary time in the segment. Note that in the first "no churn scenario," population size (i.e., the number of individuals) becomes progressively smaller as the application makes progress, while population size may fluctuates in the "churn scenario."

## 4.2 Distribution of Individuals to Workers

At the onset of each generation the master sends an equal numbers of individuals to each worker. This is because the master assumes homogeneous workers and thus strives for perfect load-balancing. We call this number $I$. Whenever a worker does not return the evaluated individuals within a time interval $T$, then those $I$ individuals are considered lost. In the fault-tolerant approach in Section 3.2, such individuals are simply re-sent to other workers. In our non-fault-tolerant approach, these individuals are simply lost and do not participate in the subsequent generations.

Note that for our non-fault-tolerant approach the execution time per generation in the failure-free and the failure-prone case are identical: with $P$ individuals to be evaluated at a given generation and $W$ workers, the master sends $I = P/W$ individuals to each worker. When a worker fails $I$ individuals are lost. Given that these individuals are discarded for the next generation and that the initial population size is never exceeded by new extra individuals, the remaining workers will continue evaluating $I$ individuals each, regardless of the number of failures or newly available hosts.

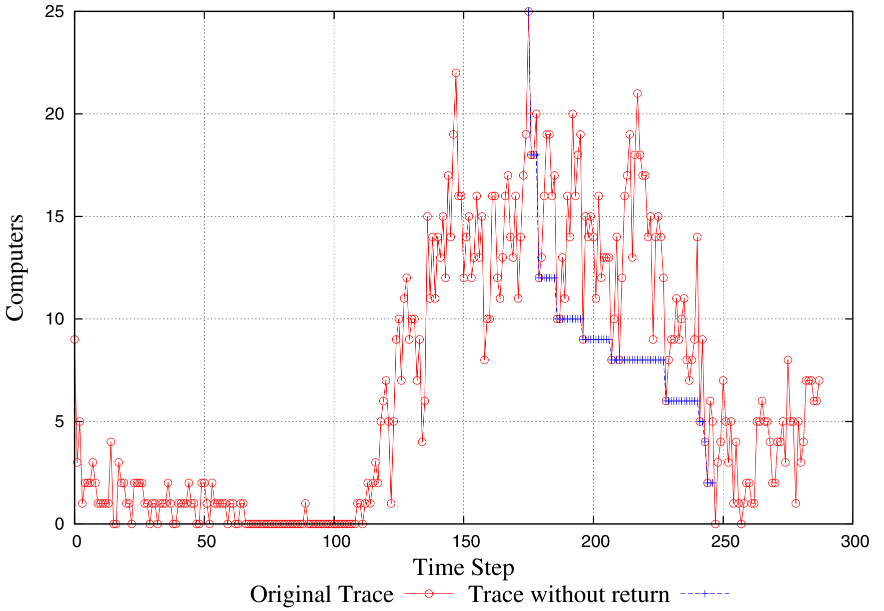


**Fig. 1** Host availability for 1 day of the *ucb* trace.

Regardless of the approach in use, if there is host churn then the population size can be increased at run-time due to the newly available hosts. We impose the restriction that the master never overcomes a pre-specified population size. This may leave some workers idle whenever a large number of workers become available. In such a case, it would be interesting to re-adjust the number of individuals $I$ sent to each worker so as to utilize all the available workers. We leave such load-balancing study outside the scope of this work and maintain $I$ constant.

In the churn scenario, one important question is: what work is assigned to newly available workers? When a new worker appears, the master simply creates $I$ new random individuals and increases the population size accordingly (provided it remains below the initial population size). These new individuals are sent to the new worker. Note that whenever there are no available workers at all, the master loses all its individuals except the best one thanks to the elitism parameter. Then, the master proceeds to the next generation by waiting a time $T$ for newly available workers.

## 4.3 Experimental Procedure

We have performed a statistical analysis of our results based on 100 trials for each experiment, accounting for the fact that different individuals can be lost depending on which individuals were assigned to which hosts. We have analyzed the normality of the results using the Kolgomorov-Smirnov and Shapiro-Wilk tests, finding out that all results are non-normal. Therefore, to compare two samples, the failure-free case with each trace (with and without churn), we used the Wilcoxon test. Table 4 and 8 present the Wilcoxon analysis of the data. The following sections discuss these results in detail.

# 5 Experimental Results

## *5.1 GP: Even Parity 5*

For the GP problem, fitness is measured as the error in the obtained solution, with zero meaning that a perfect solution has been found. All the GP parameters, including population sizes, are Koza-I/II standard [33]. See Table 2 for all details.

**Table 2** Parameters of selected problems.

| | EP5 |
|---|---|
| Population | 4000 |
| Generations | 51 |
| Elitism | Yes |
| Crossover Probability | 0.90 |
| Reproduction Probability | 0.10 |
| Selection: Tournament | 7 |
| Max Depth in Cross | 17 |
| Max Depth in Mutation | 17 |
| ADFs | Yes |

Even if the required time for fitness evaluation for the problems at hand is short, we simulate larger evaluation times representative of difficult real-world problems (so that 51 generations, the maximum, correspond to approximately 5 hours of computation in a platform without any failures).

### 5.1.1 EP5: Results without Churn

In this section we consider the scenario in which hosts never become available again (no churn). Figure 2 shows the evolution of the number of individuals in each generation for the EP5 problem when simulated over two 24-hour periods, denoted by *Day 1* and *Day 2*, randomly selected out of each of three of our traces, *entrfin*, *ucb*, and *xwtr*, for a total of 6 experiments.

Table 3 shows a summary of the obtained fitness for the EP5 problems and of the fraction of lost individuals by the end of application execution. The first row of the table shows fitness values assuming a failure-free case. The fraction of lost individuals depends strongly on the trace and on the day. For instance, the *Day 1*

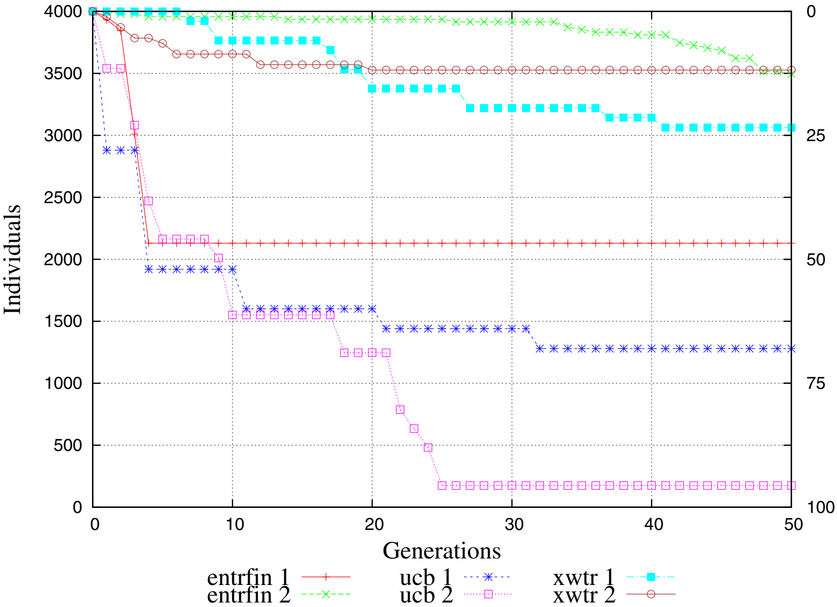


**Fig. 2** Population size vs. generation.

period of the *entrfin* trace exhibits on its 10 first generations a severe loss of individuals (almost half); the *ucb* trace on its *Day 2* period loses almost the entire population after 25 generations (96.15% loss); and the *xwtr* exhibits more moderate losses, with overall 23.52% and 12.08% loss after 51 generations for *Day 1* and *Day 2*, respectively.

The obtained fitness in the failure-free case is 2.56, and it ranges from 2.44 to 5.13 for the failure-prone cases (see Table 4 for statistical significance of results). The quality of the fitness depends on host losses in each trace. The *entrfin* and *ucb* traces present the most severe losses. The *ucb* trace exhibits 68% losses for *Day 1* and 96.15% for *Day 2*. Therefore, the obtained fitness in these two cases are the worst ones relatively to the failure-free fitness. The *entrfin* trace exhibits 48.02% and 13.04% host losses for *Day 1* and *Day 2*, respectively. As with the previous trace, when losses are too high, as in *Day 1*, the quality of the solution is significantly worse than that in the failure-free case; when losses are lower, as in *Day 2*, the obtained fitness is not significantly far from the failure-free case. Similarly, the *xwtr* trace with losses under 25% leads to a fitness that it is not significantly different from the failure-free case.

We conclude that, for the EP5 problem, it is possible to tolerate a gradual loss of up to 25% of the individuals without sacrificing solution quality. This is the case without using any fault tolerance technique. However, if the loss of individuals is too large, above 50%, then solution quality is significantly diminished. Since real-world DGSs do exhibit such high failure rates when running PGP applications, we attempt

**Table 3** Obtained fitness for EP5

| | | EP5 |
|---|---|---|
| Trace | Loss(%) | Fitness |
| Error free | 0.00 | 2.56 |
| *entrfin* (Day 1) | 48.02 | 3.58 |
| *entrfin* (*Day* 2) | 13.04 | 2.44 |
| *ucb* (Day 1) | 68.00 | 3.98 |
| *ucb* (*Day* 2) | 96.15 | 5.13 |
| *xwtr* (Day 1) | 23.52 | 2.78 |
| *xwtr* (*Day* 2) | 12.08 | 2.61 |

**Table 4** EP5 fitness comparison between failure-prone and failure-free cases using Wilcoxon test (*Day 1 and 2*) – "not significantly different" means fitness quality comparable to the failure-free case.

**Error Free fitness = 2.56**

***Results without Host Churn***

| | Trace | Fitness | Wilcoxon Test | Significantly different? | | Fitness | Wilcoxon Test | Significantly different? |
|---|---|---|---|---|---|---|---|---|
| Day 1 | *entrfin* | 3.58 | W = 6726, p-value = 1.843e-05 | yes | Day 2 | **2.44** | **W = 4778.5, p-value = 0.5815** | **no** |
| | *entrfin* 10% | 3.52 | W = 6685, p-value = 2.707e-05 | yes | | **2.65** | **W = 5201.5, p-value = 0.6167** | **no** |
| | *entrfin* 20% | 3.01 | W = 5760, p-value = 0.05956 | yes | | **2.29** | **W = 4571, p-value = 0.2863** | **no** |
| | *entrfin* 30% | 3.13 | W = 5942.5, p-value = 0.01941 | yes | | **2.36** | **W = 4732.5, p-value = 0.505** | **no** |
| | *entrfin* 40% | **2.80** | **W = 5355, p-value = 0.3773** | **no** | | 2.01 | W = 4098, p-value = 0.02458 | yes |
| | *entrfin* 50% | **2.85** | **W = 5620, p-value = 0.1233** | **no** | | 1.92 | W = 3994.5, p-value = 0.01213 | yes |
| | *ucb* | 3.98 | W = 7274, p-value = 1.789e-08 | yes | | 5.13 | W = 8735.5, p-value < 2.2e-16 | yes |
| | *ucb* 10% | 3.75 | W = 6927.5, p-value = 1.799e-06 | yes | | 5.21 | W = 8735.5, p-value < 2.2e-16 | yes |
| | *ucb* 20% | 3.61 | W = 6769, p-value = 1.123e-05 | yes | | 4.68 | W = 8266.5, p-value = 6.661e-16 | yes |
| | *ucb* 30% | 3.33 | W = 6390, p-value = 0.0005542 | yes | | 4.50 | W = 8152, p-value = 6.439e-15 | yes |
| | *ucb* 40% | 3.35 | W = 6408, p-value = 0.000464 | yes | | 4.71 | W = 8325.5, p-value = 2.220e-16 | yes |
| | *ucb* 50% | 3.17 | W = 6080, p-value = 0.007298 | yes | | 4.47 | W = 8024.5, p-value = 6.95e-14 | yes |
| | *xwtr* | **2.78** | **W = 5509, p-value = 0.2043** | **no** | | **2.61** | **W = 5238.5, p-value = 0.5524** | **no** |
| | *xwtr* 10% | **2.40** | **W = 4762, p-value = 0.5532** | **no** | | **2.66** | **W = 5215.5, p-value = 0.5927** | **no** |
| | *xwtr* 20% | **2.32** | **W = 4643.5, p-value = 0.3753** | **no** | | **2.42** | **W = 4686.5, p-value = 0.4364** | **no** |
| | *xwtr* 30% | **2.46** | **W = 4802, p-value = 0.6221** | **no** | | **2.33** | **W = 4611.5, p-value = 0.3336** | **no** |
| | *xwtr* 40% | **2.15** | **W = 4363, p-value = 0.1121** | **no** | | 1.96 | W = 4033.5, p-value = 0.01574 | yes |
| | *xwtr* 50% | **2.13** | **W = 4296.5, p-value = 0.08027** | **no** | | 2.24 | **W = 4511, p-value = 0.2226** | **no** |

***Results with Host Churn***

| | Trace | Fitness | Wilcoxon Test | Significantly different? | | Fitness | Wilcoxon Test | Significantly different? |
|---|---|---|---|---|---|---|---|---|
| | *entrfin* | **2.86** | W = **5513.5, p-value=0.2012** | **no** | | **2.75** | **W = 5404.5, p-value = 0.3142** | **no** |
| | *ucb* | 8.87 | W = 9997, p-value = 2.2e-16 | yes | | 5.89 | W = 9645, p-value < 2.2e-16 | yes |
| | *xwtr* | **2.56** | **W = 4940, p-value = 08823** | **no** | | **2.52** | **W = 5035, p-value = 0.9315** | **no** |

to remedy this problem. Our simple idea is to increase the initial population size (in our case by 10, 20, 30, 40, or 50%). The goal is to compensate for lost individuals by starting with a larger population.

Increasing population likely also affects the fitness in the failure-free case. We simulated the EP5 problem in the failure-free case with a population size increased by 10, 20, 30, 40 and 50%. Results are shown in Figure 3, which plots the evolution of fitness versus the "computing effort." The computing effort is defined as the total number of evaluated individuals nodes so far (we must bear in mind that GP individuals are variable size trees), i.e., from generation 1 to generation $G$, as described in [16]. We have fixed a maximum computing effort which corresponds to 51 generations and the population size introduced by Koza [33], which is employed in this work. Figure 3 shows that population sizes $M > 4,000$ for a similar effort obtain worse fitness values when compared with the original $M = 4{,}000$ population size. Thus, for static populations, increasing the population size is not a good option, provided a judicious population size is chosen to begin with. Nevertheless, we content that such population increase could be effective in a failure-prone case.

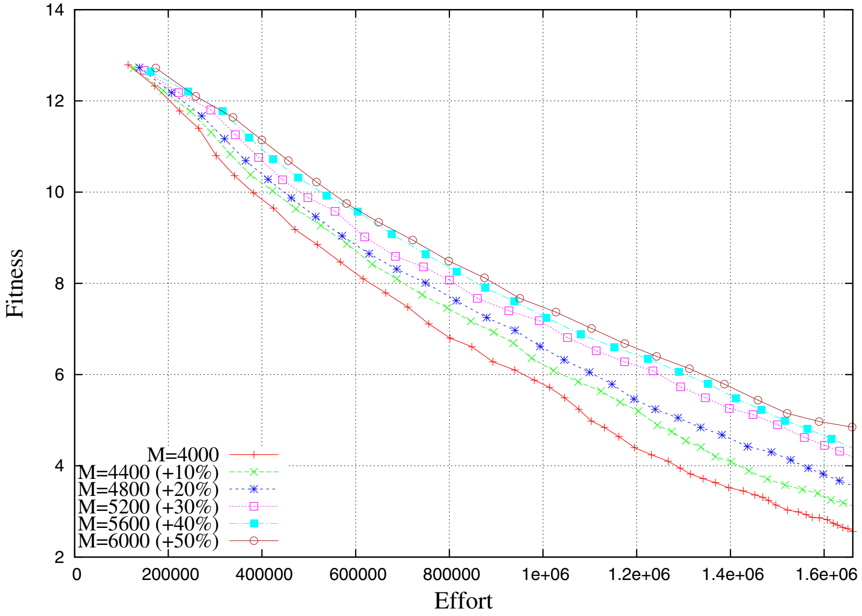


**Fig. 3** Fitness vs. Effort with increased population for failure-free experiments

Table 5 shows results for the increased initial population size, based on simulations for the *Day 1* and *Day 2* periods of all three traces. Overall, increasing the initial population size is an effective solution to tolerate failures while preserving (and even improving!) solution quality. For instance, for the *Day 1* period of the *entrfin* trace, with host losses at 48.02%, starting with 50% extra individuals ensures solution quality on par with the failure-free case. Similar results are obtained for the *entrfin* and *xwtr* two periods. Furthermore, for the *Day 2* period of traces *entrfin* and *xwtr*, adding 40% or 50% extra individuals results in obtaining solutions of better quality than in the failure-free case. However, the increase of the initial population is not enough for the *ucb* trace as its losses are as high as 96.15% and 68% for *Day 1* and *Day 2*, respectively. Note that in these difficult cases the fault-tolerant approach does not succeed at all.

From these results we conclude that increasing the initial population size is effective to maintain fitness quality at the level of that in the failure-free case. The fraction by which the population is increased is directly correlated to the host loss rate. If an estimation of this rate is known, for instance based on historical trends, then the initial population size can be chosen accordingly. Also, one must keep in

**Table 5** EP5 fitness with increased population

| Error Free fitness = 2.56 | | | | | | |
|---|---|---|---|---|---|---|
| Traces | *entrfin* | *ucb* | *xwtr* | *entrfin* | *ucb* | *xwtr* |
| +0% | 3.58 | 3.98 | 2.78 | 2.44 | 5.13 | 2.61 |
| +10% | 3.52 | 3.75 | 2.40 | 2.65 | 5.21 | 2.66 |
| +20% | 3.01 | 3.61 | 2.32 | 2.29 | 4.68 | 2.42 |
| +30% | 3.13 | 3.33 | 2.46 | 2.36 | 4.50 | 2.33 |
| +40% | 2.80 | 3.35 | 2.15 | 2.01 | 4.71 | 1.96 |
| +50% | 2.85 | 3.17 | 2.13 | 1.92 | 4.47 | 2.24 |

mind that an increased population size implies longer execution time for each generation since more individuals must be evaluated.

### 5.1.2 EP5: Results with churn

In this section we present results for the case in which hosts can become available again after becoming unavailable, leading to churn. Recall from the discussion at the beginning of Section 4.2 that the population size is capped at 4,000 individuals (according to Table 2) and that each worker is assigned *I* individuals. Such individuals are randomly generated by the master when assigned to a newly available worker.

**Table 6** Obtained fitness for EP5 with host churn

| Trace | Hosts | | | | | Fitness |
|---|---|---|---|---|---|---|
| | Min. | Median | Mean | Max. | Var. ($s^2$) | EP5 |
| Error free | - | - | - | - | - | 2.56 |
| *entrfin* (*Day 1*) | 92 | 160 | 157.50 | 177 | 179.33 | 2.86 |
| *entrfin* (*Day 2*) | 180 | 181 | 181.30 | 183 | 0.75 | 2.75 |
| *ucb* (*Day 1*) | 0 | 1 | 1.51 | 9 | 2.21 | 8.87 |
| *ucb* (*Day 2*) | 0 | 2 | 2.57 | 7 | 4.29 | 5.89 |
| *xwtr* (*Day 1*) | 28 | 29 | 28.92 | 29 | 0.07 | 2.56 |
| *xwtr* (*Day 2*) | 86 | 86 | 86 | 86 | 0 | 2.52 |

Table 6 shows the obtained fitness for the EP5 problem on all traces. It also shows the host churn represented by the minimum, median, mean, maximum, and variance of the number of available hosts during application execution. Among all the traces, the *ucb* trace is the worst possible scenario as it has very few available hosts. This prevents the master from sending individuals to workers, both in *Day 1* and *Day 2*, leading to poor fitness values. For the *entrfin* and *xwtr* traces, both for *Day 1* and *Day 2*, the obtained fitness value is comparable to that in the failure-free case (see Table 4 for statistical significance).

If the variance of the number of available hosts for a trace is zero, then the trace is equivalent to the failure-free case, as the hosts do not experience any failure. The obtained fitness should then be similar to that in the failure-free case.

The *xwtr* trace, *Day 2*, exhibits such zero variance, and indeed the obtained fitness value is similar to that in the failure-free case (see Table 6). The variance of the *xwtr* trace, *Day 1*, is low at 0.07, and the obtained fitness is again on par with that in the failure-free case. The *entrfin* trace, *Day 1*, exhibits the largest variance. Nevertheless, the obtained fitness is better than that of its counterpart in the non-churn scenario, and similar to that in the failure-free case. This shows that re-acquiring hosts is, expectedly, beneficial. Finally the *ucb* trace leads to the worst fitness values despite its low variability (see Table 6). The reason is a low maximum number of available hosts (9 and 7 for *Day 1* and *Day 2*, respectively), and many periods during which no hosts were available at all (in which case the master loses the entire population except for the best individual). As a result, it is very difficult to obtain solutions comparable to those in the failure-free case.

## 5.2 GA: 3-Trap Function

According to [12], 3-trap lies on the region between the deceptive 4-trap and the non-deceptive 2-trap having, therefore, intermediate population size requirements that Thierens estimates in 3000 for the instance under study in [49]. A trap function is a piecewise-linear function defined on unitation (the number of ones in a binary string). There are two distinct regions in the search space, one leading to a global optimum and the other one to the local optimum (see Eq. 1). In general, a trap function is defined by the following equation:

$$trap(u(\overrightarrow{x})) = \begin{cases} \frac{a}{z}(z - u(\overrightarrow{x})), & \text{if} \quad u(\overrightarrow{x}) \leq z \\ \frac{b}{l-z}(u(\overrightarrow{x}) - z), & \text{otherwise} \end{cases} \tag{1}$$

where $u(\overrightarrow{x})$ is the unitation function, $a$ is the local optimum, $b$ is the global optimum, $l$ is the problem size and $z$ is a slope-change location separating the attraction basin of the two optima.

For the following experiments, 3-trap was designed with the following parameter values: $a = l-1$, $b = l$, and $z = l-1$. Tests were performed by juxtaposing $m = 10$ trap functions in binary strings of length $L = 30$ and summing the fitness of each sub-function to obtain the total fitness. All settings are summarized in Table 7.

In order to analyze the results with confidence, data has been statistically analyzed (each experiment has been run 100 times). Firstly, we analyzed the normality of the data using the Kolgomorov-Smirnov and Shapiro-Wilk tests [10], obtaining as a result that all data are non-normal. Thus, to compare two samples, the error-free case with each trace, we used the Wilcoxon test (Table 8 shows the Wilcoxon analysis of the data).

Figure 2 shows, for the worst-case scenario, how the population decreases as failures occur in the system. As explained before, two different 24-hours periods

**Table 7** Parameters of the experiments

| **Trap instance** | |
|---|---|
| Size of sub-function ($k$) | 3 |
| Number of sub-functions ($m$) | 10 |
| Individual length ($L$) | 30 |
| **GA settings** | |
| GA | GGA |
| Population size | 3000 |
| Selection of Parents | Binary Tournament |
| Recombination | Uniform crossover, $p_c = 1.0$ |
| Mutation | Bit-Flip mutation, $p_m = \frac{1}{L}$ |

**Table 8** 3-Trap fitness comparison between error-prone and error-free cases using Wilcoxon test (*Day 1 and 2*) – "not significantly different" means fitness quality comparable to the error-free case.

**Error Free fitness = 23.56**

***Results without Host Churn***

| | Trace | Fitness | Wilcoxon Test | Significantly different? | | Fitness | Wilcoxon Test | Significantly different? |
|---|---|---|---|---|---|---|---|---|
| Day 1 | Entrfin | 23.3 | W = 6093, p-value = 0.002688 | yes | Day 2 | **23.57** | **W = 4979.5, p-value = 0.9546** | **no** |
| | Entrfin 10% | **23.47** | **W = 5408.5, p-value = 0.2535** | **no** | | **23.69** | **W = 4397.5, p-value = 0.07682** | **no** |
| | Entrfin 20% | **23.48** | **W = 5360, p-value = 0.3137** | **no** | | **23.67** | **W = 4522.5, p-value = 0.1645** | **no** |
| | Entrfin 30% | **23.49** | **W = 5283.5, p-value = 0.4271** | **no** | | **23.70** | **W = 4405, p-value = 0.08086** | **no** |
| | Entrfin 40% | **23.57** | **W = 4923.5, p-value = 0.8286** | **no** | | **23.69** | **W = 4453.5, p-value = 0.11** | **no** |
| | Entrfin 50% | **23.59** | **W = 4910.5, p-value = 0.7994** | **no** | | 23.75 | W = 4162.5, p-value = 0.01234 | yes |
| | Ucb | 23.22 | W = 6453, p-value = 6.877e-05 | yes | | 23.09 | W = 6672.5, p-value = 7.486e-06 | yes |
| | Ucb 10% | 23.27 | W = 6098.5, p-value = 0.002753 | yes | | 23.12 | W = 6826, p-value = 6.647e-07 | yes |
| | Ucb 20% | 23.37 | W = 5837.5, p-value = 0.02051 | yes | | 23.14 | W = 6654, p-value = 7.223e-06 | yes |
| | Ucb 30% | **23.40** | **W = 5664, p-value = 0.06588** | **no** | | 23.26 | W = 6371, p-value = 0.0001507 | yes |
| | Ucb 40% | **23.51** | **W = 5186.5, p-value = 0.6004** | **no** | | 23.37 | W = 5893.5, p-value = 0.01316 | yes |
| | Ucb 50% | **23.42** | **W = 5623, p-value = 0.08335** | **no** | | 23.32 | W = 6108, p-value = 0.002166 | yes |
| | Xwtr | **23.56** | **W = 5056, p-value = 0.8748** | **no** | | **23.60** | **W = 4806, p-value = 0.5791** | **no** |
| | Xwtr 10% | **23.57** | **W = 4923.5, p-value = 0.8286** | **no** | | **23.62** | **W = 4765, p-value = 0.5002** | **no** |
| | Xwtr 20% | **23.68** | **W = 4474, p-value = 0.1245** | **no** | | **23.69** | **W = 4453.5, p-value = 0.11** | **no** |
| | Xwtr 30% | 23.73 | W = 4259.5, p-value = 0.02812 | yes | | **23.60** | **W = 4806, p-value = 0.5791** | **no** |
| | Xwtr 40% | **23.68** | **W = 4502, p-value = 0.1466** | **no** | | **23.63** | **W = 4688.5, p-value = 0.3695** | **no** |
| | Xwtr 50% | **23.71** | **W = 4356.5, p-value = 0.05817** | **no** | | 23.77 | W = 4065.5, p-value = 0.004877 | yes |
| | ***Results with Host Churn*** | | | | | | | |
| | Entrfin | **23.52** | W = **W = 5222, p-value = 0.5322** | **no** | | **23.58** | **W = 4931, p-value = 0.8452** | **no** |
| | Ucb | 21.31 | W = 9708.5, p-value < 2.2e-16 | yes | | 23.03 | W = 7038.5, p-value = 4.588e-08 | yes |
| | Xwtr | **23.64** | **W = 4640, p-value = 0.2982** | **no** | | **23.7** | **W = 4405, p-value = 0.08086** | **no** |

randomly selected are shown, denoted by Day 1 and Day 2, for the three employed traces of the experiments. Thus, a total of 6 different experiments, one per trace and day period, were run with the 3-Trap function problem.

Table 8 shows a summary of the obtained results for the experiments. From all the traces, the *ucb* has obtained the worst fitness 23.22 and 23.09 (respectively for both periods Day 1 and Day 2). The reason is that this trace in the first day loses a 64% of the population and in the second day it loses more or less the whole population 95.83% (see Figure 2). Consequently it is very difficult for the algorithm to obtain a solution with a similar quality to the error-free scenario.

The second worst case of all the experiments is the *entrfin* trace for the first period (Day 1). This trace loses more or less half of the population in the first 5 generations (see Figure 2), making really difficult to obtain a good solution even though the population size is steady the rest of the generations. Thus, the obtained fitness for this period is not comparable to the error-free case.

Finally, the *xwtr* trace in both periods obtains solutions with similar quality to the error-free environment (23.56 and 23.6 respectively for each day). In both periods, the *xwtr* trace does not lose more than a 20% for Day 1 and 12% for the second day. Consequently, we conclude that for the 3-Trap function problem, it is possible to tolerate a gradual loss of up to 20% of the individuals without sacrificing solution quality and more importantly without using any fault-tolerance mechanism. Nevertheless, if the loss of individuals is too high, above the 45%, the solution quality is significantly diminished. Since real-world DGSs experience such large amount of failures, we attempt to address this problem. Our simple idea is to increase the initial population size (a 10%, 20%, 30%, 40% and 50%) and run the same simulations using the same traces. The aim is to compensate the loss of the system by providing more individuals at the first generation.

Table 8 shows the obtained results for Day 1 and Day 2 periods of the three traces with the increased population. For the *entrfin* trace, the first period (Day 1) with a loss rate of 45.3%, a 10% extra individuals is enough to obtain solutions of similar quality to the error-free case. In the second period, Day 2, the trace obtains similar solutions to the error-free case and when adding an extra 50% the obtained solution is even better than in the error-free case.

For the *ucb* trace, the first period (Day 1) increasing a 30% the size of the population is sufficient to obtain solutions with similar quality to the error-free case. The second period, Day 2, even though an extra 50% of individuals is added at the first generation it is not enough to cope with the high loss rate of this period: 95.83%.

Finally, the *xwtr* trace for both periods obtains solutions with similar quality to the error-free case and in some cases it improves it. For this trace, the increased population would have not been necessary because the PGA tolerates, without any extra individual, the rate loss of both periods.

It is important to remark that by adding more individuals to the initial population, we are increasing the computation time since more individuals have to be evaluated per generation. Nevertheless, this extra time is similar to the extra time that would be required by standard fault tolerance mechanisms (e.g. failure detection and re-send lost individuals for fitness evaluation). Thus, we conclude that increasing the population size, accordingly to the failure rate, is enough to improve the PGA quality of solutions when the failure rate is known.

Up to now, we have only considered the worst-case scenario: lost resources never become available again. Nevertheless, real-world DG systems does not behave like this assumption, and thus we are going to use the traces with the possibility of re-acquiring the lost resources (see Figure 1). Next section analyzes the results obtained when re-acquiring lost resources is a possibility.

### 5.2.1 3-Trap:Results with Churn

When using the full churn traces of the three DGSs (*entrfin*, *ucb* and *xwtr*) an important question arises: what work is assigned to the new available workers? We have assumed that when workers become available again the master node creates *I* new random individuals and increases the size of the population accordingly. Thus, the size of the population can be changed dynamically as individuals are added and removed along generations. In this scenario it could happen that new workers nodes appear during the execution of the algorithm increasing the population over its optimum size. Hence the master node is not allowed to create more individuals than the optimum population size leaving several workers idle. In order to avoid idle workers, it would be interesting to adjust the number of *I* individuals to evaluate accordingly to the number of available hosts. Nevertheless, we leave such load-balancing study for a future work.

On the other hand, due to the loss of resources, the population can be emptied because all the workers have disappeared. If this situation occurs, the server node proceeds to the next generation by waiting the specified time *T* (based on the required time per generation in the failure-free environment) for new workers.

Table 8 shows the obtained results for the three traces with the host-churn phenomena (*entrfin*, *ucb* and *xwtr*) and the previous corresponding two periods: Day 1 and Day 2. We used the same periods as in the worst-case scenario, but now choosing a random point in the 24-hours period as the starting point for the algorithm. Table 9 shows the obtained fitness of the 3-Trap function problem and the host churn of each trace represented by the minimum, median, mean, maximum, and variance of the number of available worker nodes.

**Table 9** Obtained fitness for 3-Trap function with host churn

| Trace | Hosts | | | | | Fitness |
|---|---|---|---|---|---|---|
| | Min. | Median | Mean | Max. | Var. ($s^2$) | 3-Trap |
| Error free | - | - | - | - | - | 23.56 |
| entrfin (*Day 1*) | 92 | 161.5 | 156.8 | 177 | 305.59 | 23.52 |
| entrfin (*Day 2*) | 180 | 181 | 180.9 | 182 | 0.6 | 23.58 |
| ucb (*Day 1*) | 0 | 2 | 1.9 | 9 | 3.12 | 21.31 |
| ucb (*Day 2*) | 0 | 4 | 3.7 | 7 | 2.7 | 23.03 |
| xwtr (*Day 1*) | 28 | 29 | 28.87 | 29 | 0.11 | 23.64 |
| xwtr (*Day 2*) | 86 | 86 | 86 | 86 | 0 | 23.70 |

If the variance of the number of available hosts is zero, then the execution is obviously the same as in the error-free case because the number of hosts is steady along generations. In this case, the obtained fitness should be similar to the error-free case. This situation is present within the second period (Day 2) of the *xwtr* trace (variance equal to zero) and thus the obtained fitness is similar to the error-free case (see Table 8). The other period of the *xwtr* trace has also a very small variance, 0.11, resulting in a similar solution quality in comparison with the error-free scenario. The *entrfin* trace for both periods obtains solutions of similar quality to the error-free environment, even though the large variance observed in the Day 1 period ($s^2 = 305.59$). Despite the large variance, the number of available hosts is high in comparison with the other traces, so the PGA tolerates better the failures and provides solutions of similar quality to the error-free case. Finally, the *ucb* trace obtains the worst results due to in both periods the minimum number of available hosts is zero. Consequently, the population is emptied, making very difficult to obtain solutions of similar quality to the error-free environment.

### *5.3 Summary of Results*

Based on two standard applications, EP5 and 3-trap, we have shown that PGP and PGA applications based on the master-worker model running on DGSs that exhibit host failures can achieve solution qualities close to those in the failure-free case, without resorting to any fault tolerance technique. Two scenarios were tested: (i) the scenario in which lost hosts never come back but in which one starts with a large number of hosts; and (ii) the scenario in which hosts can re-appear during application execution. For scenario (i) we found that there is an approximately linear degradation of solution quality as host losses increase. This degradation can be alleviated by increasing initial population size. For scenario (ii) the degradation varies during application execution as the number of workers fluctuates. The main observation is that in both cases we have *graceful degradation*.

## 6 Conclusions

In this chapter we have analyzed the behavior of a parallel approach to Genetic Programming and Genetic Algorithms when executed in a distributed platform with high failure rate. The aim is characterizing the inherent fault tolerance capabilities of the evolutionary computation paradigm. To that end, we have used two well-known problems and for the first time in this context (to the best of our knowledge) we have used host availability traces collected on real-world Desktop Grid platforms.

Our main conclusion is that, whenever executed in parallel, either GP and GA provide a fault tolerant mechanism known as *graceful degradation*.

We have also presented a simple method for tolerating faults in especially challenging scenarios with high host losses, which consists of increasing the initial population size.

To the best of our knowledge, this is the first time that PGP and PGAs are characterized from a fault-tolerance perspective. We contend that our conclusions can be extended to other Parallel Evolutionary Algorithms via similar experimental validation.

**Acknowledgements.** This work was supported by Luxembourg FNR GreenIT Project (C09/IS/05), Spanish Ministry of Science Projects TIN2008-05941 and TIN2011-28627-C04, Andalusian Regional Project P08-TIC-03903 and the Citizen Cyberscience Centre.

## References

1. Ackley, D.H.: A connectionist machine for genetic hillclimbing. Kluwer Academic Publishers, Norwell (1987)
2. Agbaria, A., Friedman, R.: Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In: HPDC 1999: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing, vol. 31, IEEE Computer Society, Washington, DC (1999)
3. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Transactions on Evolutionary Computation 6(5), 443–462 (2002)
4. Anderson, D.P.: Boinc: a system for public-resource computing and storage. In: Proceedings of Fifth IEEE/ACM International Workshop on Grid Computing, 2004, pp. 4–10 (2004)
5. Anderson, D.P., Fedak, G.: The Computational and Storage Potential of Volunteer Computing. In: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid, CCGRID 2006 (2006)
6. Andre, D., Koza, J.R.: Parallel genetic programming: a scalable implementation using the transputer network architecture, pp. 317–337 (1996)
7. Arenas, M., Collet, P., Eiben, A.E., Jelasity, M., Merelo, J.J., Paechter, B., Preuß, M., Schoenauer, M.: A Framework for Distributed Evolutionary Algorithms. In: Guervós, J.J.M., Adamidis, P.A., Beyer, H.-G., Fernández-Villacañas, J.-L., Schwefel, H.-P. (eds.) PPSN 2002. LNCS, vol. 2439, pp. 665–675. Springer, Heidelberg (2002)
8. Cahon, S., Melab, N., Talbi, E.G.: Building with paradisEO reusable parallel and distributed evolutionary algorithms. Parallel Computing 30(5-6), 677–697 (2004)
9. Cantu-Paz, E.: A survey of parallel genetic algorithms. Calculateurs Paralleles, Reseaux et Systems Repartis 10(2), 141–171 (1998)
10. Crawley, M.J.: In: Statistics, An Introduction using R. Wiley (2007)
11. Francisco Chávez de la, O., Guisado, J.L., Lombraña, D., Fernández, F.: Una herramienta de programación genética paralela que aprovecha recursos públicos de computación. In: V Congreso Español sobre Metaheuŕsticas, Algoritmos Evolutivos y Bioinspirados, Tenerife, Spain, vol. 1, pp. 167–173 (February 2007)
12. Deb, K., Goldberg, D.E.: Analyzing deception in trap functions. In: Darrell Whitley, L. (ed.) FOGA, pp. 93–108. Morgan Kaufmann (1992)
13. Desell, T., Szymanski, B., Varela, C.: An asynchronous hybrid genetic-simplex search for modeling the Milky Way galaxy using volunteer computing. In: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, pp. 921–928. ACM (2008)
14. Douglas Zongker Dr. Bill Punch. Lil-gp.: `http://garage.cse.msu.edu/software/lil-gp/index.html`
15. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys (CSUR) 34(3), 375–408 (2002)
16. Vanneschi, L., Fernández, F., Tomassini, M.: Saving computational effort in genetic programming by means of plagues. In: The 2003 Congress on Evolutionary Computation, CEC 2003 (2003)
17. Fernandez, F., Spezzano, G., Tomassini, M., Vanneschi, L.: Parallel genetic programming. In: Alba, E. (ed.) Parallel Metaheuristics, Parallel and Distributed Computing, ch. 6, pp. 127–153. Wiley-Interscience, Hoboken (2005)
18. Message Passing Interface Forum. Mpi: a message-passing interface standard. International Journal Supercomput. Applic. 8(3-4), 165–414 (1994)

19. Gagné, C., Parizeau, M., Dubreuil, M.: Distributed beagle: An environment for parallel and distributed evolutionary computations. In: Proc. of the 17th Annual International Symposium on High Performance Computing Systems and Applications (HPCS 2003), May 11-14, pp. 201–208 (2003)
20. Gartner, F.C.: Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Computing Surveys 31(1), 1–26 (1999)
21. Ghosh, S.: Distributed systems: an algorithmic approach. Chapman & Hall/CRC (2006)
22. Gonzáez, D.L., de Vega, F.F., Casanova, H.: Characterizing fault tolerance in genetic programming. In: Workshop on Bio-Inspired Algorithms for Distributed Systems, Barcelona, Spain, pp. 1–10 (June 2009)
23. González, D.L., de Vega, F.F., Casanova, H.: Characterizing fault tolerance in genetic programming. Future Generation Computer Systems 26(6), 847–856 (2010)
24. González, D.L., de Vega, F.F., Trujillo, L., Olague, G., Araujo, L., Castillo, P., Merelo, J.J., Sharman, K.: Increasing gp computing power for free via desktop grid computing and virtualization. In: Proceedings of the 17th Euromicro Conference on Parallel, Distributed and Network-Based Processing, Weimar, Germany, pp. 419–423 (February 2009)
25. González, D.L., Laredo, J.L.J., de Vega, F.F., Guervós, J.J.M.: Characterizing fault-tolerance of genetic algorithms in desktop grid systems. In: 10th European Conference on Evolutionary Computation in Combinatorial Optimization, Istanbul, Turkey, pp. 131–142 (April 2010)
26. Guerraoui, R., Schiper, A.: Software-Based Replication for Fault Tolerance. IEEE Computer 30(4), 68–74 (1997)
27. Hidalgo, I., Fernández, F., Lanchares, J., Lombraña, D.: Is the island model fault tolerant? In: Genetic and Evolutionary Computation Conference, London, England, vol. 2, p. 1519 (July 2007)
28. Jelasity, M., Preuß, M., van Steen, M., Paechter, B.: Maintaining connectivity in a scalable and robust distributed environment. In: Bal, H.E., Löhr, K.-P., Reinefeld, A. (eds.) Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2002), 2nd GP2PC Workshop, Berlin, Germany, pp. 389–394. IEEE Computer Society (2002)
29. Knispel, B., Allen, B., Cordes, J.M., Deneva, J.S., Anderson, D., Aulbert, C., Bhat, N.D.R., Bock, O., Bogdanov, S., Brazier, A., et al.: Pulsar Discovery by Global Volunteer Computing. Science 329(5997), 1305 (2010)
30. Kondo, D., Fedak, G., Cappello, F., Chien, A.A., Casanova, H.: Characterizing resource availability in enterprise desktop grids, vol. 23, pp. 888–903. Elsevier (2007)
31. Kondo, D., Taufer, M., Brooks, C., Casanova, H., Chien, A.: Characterizing and evaluating desktop grids: An empirical study. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2004). Citeseer (2004)
32. Kouchakpour, P., Zaknich, A., Bräunl, T.: Dynamic population variation in genetic programming. Information Sciences 179(8), 1078–1091 (2009)
33. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
34. Laredo, J.L.J., Castillo, P.A., Mora, A.M., Merelo, J.J., Fernandes, C.: Merelo, and Carlos Fernandes. Resilience to churn of a peer-to-peer evolutionary algorithm. Int. J. High Performance Systems Architecture 1(4), 260–268 (2008)
35. Laredo, J.L.J., Eiben, A.E., van Steen, M., Merelo, J.J.: On the Run-Time Dynamics of a Peer-to-Peer Evolutionary Algorithm. In: Rudolph, G., Jansen, T., Lucas, S., Poloni, C., Beume, N. (eds.) PPSN 2008. LNCS, vol. 5199, pp. 236–245. Springer, Heidelberg (2008)

36. Lombraña, D., Fernández, F.: Analyzing fault tolerance on parallel genetic programming by means of dynamic-size populations. In: Congress on Evolutionary Computation, Singapore, vol. 1, pp. 4392–4398 (2007)
37. Lombraña, D., Fernández, F., Trujillo, L., Olague, G., Cárdenas, M., Araujo, L., Castillo, P., Sharman, K., Silva, A.: Interpreted applications within boinc infrastructure. In: Ibergrid 2008, Porto, Portugal, pp. 261–272 (May 2008)
38. Lombraña, D., Fernández, F., Trujillo, L., Olague, G., Segal, B.: Customizable execution environments with virtual desktop grid computing. In: Parallel and Distributed Computing and Systems, PDCS (2007)
39. Luke, S., Balan, G.C., Panait, L.: Population Implosion in Genetic Programming. In: Cantú-Paz, E., Foster, J.A., Deb, K., Davis, D., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Standish, R., Kendall, G., Wilson, S., Harman, M., Wegener, J., Dasgupta, D., Potter, M.A., Schultz, A., Dowsland, K., Jonoska, N., Miller, J. (eds.) GECCO 2003. LNCS, vol. 2724, pp. 1729–1739. Springer, Heidelberg (2003)
40. Luke, S., Panait, L., Balan, G., Paus, S., Skolicki, Z., Popovici, E., Harrison, J., Bassett, J., Hubley, R., Chircop, A.: Ecj a java-based evolutionary computation research system (2007), http://cs.gmu.edu/~eclab/projects/ecj/
41. Melab, N., Cahon, S., Talbi, E.-G.: Grid computing for parallel bioinspired algorithms. J. Parallel Distrib. Comput. 66(8), 1052–1061 (2006)
42. Pruyne, J., Livny, M.: Interfacing Condor and PVM to harness the cycles of workstation clusters. Future Generations Computer Systems FGCS 12(1), 67–85 (1996)
43. Reed, D.A., Lu, C., Mendes, C.L.: Reliability challenges in large systems. Future Generation Computer Systems 22(3), 293–302 (2006)
44. Schroeder, B., Gibson, G.A.: A Large-Scale Study of Failures in High-Performance Computing Systems. In: Proc. of the International Conference on Dependable Systems, pp. 249–258 (2006)
45. Shooman, M.L.: Reliability of computer systems and networks: fault tolerance, analysis and design. Wiley Interscience (2002)
46. Stutzbach, D., Rejaie, R.: Understanding churn in peer-to-peer networks. In: Proceedings of the 6th ACM SIGCOMM on Internet Measurement (IMC 2006), pp. 189–202. ACM Press, New York (2006)
47. Sunderam, V.S.: Pvm: A framework for parallel distributed computing. Concurrency: Practice and Experience 2, 315–339 (1990)
48. Tai, A.T., Tso, K.S.: A performability-oriented software rejuvenation framework for distributed applications. In: DSN 2005: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN 2005), pp. 570–579. IEEE Computer Society, Washington, DC (2005)
49. Thierens, D.: Scalability problems of simple genetic algorithms. Evolutionary Computation 7(4), 331–352 (1999)
50. Tomassini, M.: Parallel and distributed evolutionary algorithms: A review. In: Neittaanmäki, P., Miettinen, K., Mäkelä, M., Periaux, J. (eds.) Evolutionary Algorithms in Engineering and Computer Science, p. 113, 133. J. Wiley and Sons, Chichester (1999)
51. Tomassini, M.: Spatially Structured Evolutionary Algorithms. Springer (2005)
52. Tomassini, M., Vanneschi, L., Cuendet, J., Fernandez, F.: A new technique for dynamic size populations in genetic programming. In: Congress on Evolutionary Computation, CEC 2004, vol. 1 (2004)
53. Trujillo, L., Olague, G.: Automated Design of Image Operators that Detect Interest Points, vol. 16, pp. 483–507. MIT Press (2008)
54. Vargas, E.: High availability fundamentals. Sun Blueprints (2000)

# Comparison of Frameworks for Parallel Multiobjective Evolutionary Optimization in Dynamic Problems

Mario Cámara, Julio Ortega, and Francisco de Toro

**Abstract.** In this chapter some alternatives are discussed to take advantage of parallel computers in dynamic multi-objective optimization problems (DMO) using evolutionary algorithms. In DMO problems, the objective functions, the constraints, and hence, also the solutions, can change over time and usually demand to be solved online. Thus, high performance computing approaches, such as parallel processing, should be applied to these problems to meet the quality requirements within the given time constraints. Taking this into account, we describe two generic parallel frameworks for multi-objective evolutionary algorithms. These frameworks are used to compare the parallel processing performance of some multi-objective optimization evolutionary algorithms: our previously proposed algorithms, SFGA and SFGA2, in conjunction with SPEA2 and NSGA-II. We also propose a model to explain the benefits of parallel processing in multi-objective problems and the speedup results observed in our experiments.

**Keywords:** dynamic multiobjective optimization (DMO), parallel processing, parallel evolutionary algorithms

## 1 Introduction

Present evolution towards multicore processors have fueled the relevance, usefulness, and interest of parallel implementations of applications that have to fulfill time constraints, and/or require a high amount of computing workload to be

Mario Cámara · Julio Ortega
Department of Computer Architecture and Technology, University of Granada, Spain
e-mail: {mcamara,jortega}@ugr.es

Francisco de Toro
Department of Signal Theory, Telematics and Communications, University of Granada, Spain
e-mail: ftoro@ugr.es

F.F. de Vega et al. (Eds.): Parallel Architectures & Bioinspired Algorithms, SCI 415, pp. 101–123.
springerlink.com

satisfactorily solved. Dynamic optimization problems, which appear in many different real-world applications with socio-economic relevance, are examples of such class of problems. These problems are dynamic because they show changes over time in the parameters on which the cost functions depend and/or in the restrictions that their solutions must meet. For example, in some scheduling problems, such as those appearing in parallel computing servers or in the semiconductor industry [1], the available resources and the volume of tasks to be allocated could vary over time [2], or the voltage should be regulated during operation [3]. In the control of an industrial plant the conditions change due to the aging of the plant, to random intrinsic effects, etc. The shifting of optima with time is an important issue in this kind of problems. An approach to solve dynamic optimization problems uses an on-line optimization procedure that considers the dynamic problem as stationary for a given period of time in which the optimization algorithm should be able to find the optimal (or near-optimal) solution. Thus, if it is assumed that the optimization problem remains constant or almost constant within the time $t$, it should be verified that $T_{opt} < t$, where $T_{opt}$ is the time needed by the optimization algorithm to obtain an enough accurate optimal solution. This way, the faster is the optimization algorithm, the wider is the set of dynamic optimization problems where it can be applied.

On the other hand, there are many optimization problems whose solutions must optimize several objectives that are in conflict. Examples of these problems have been reported in applications such as dynamic scheduling, and inventory management [4,5]. In this context, the concept of optimum must be redefined, because instead of providing only one optimal solution, the procedures applied to these multi-objective optimization problems should obtain a set of non-dominated solutions, known as Pareto optimal solutions [6], from which a decision agent (be human or not) will choose the most convenient solution in the current circumstances. Roughly speaking, these solutions are optimal in the sense that in the corresponding

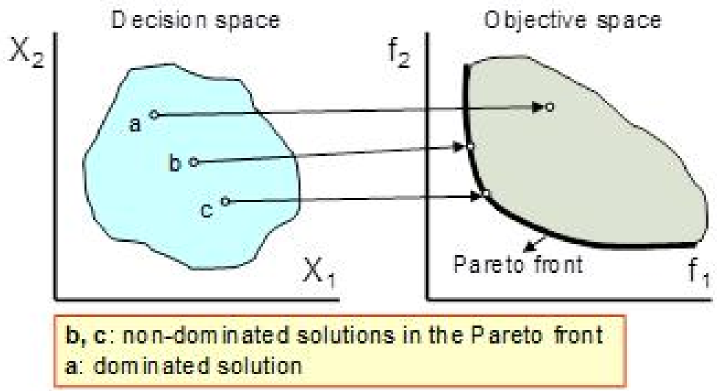


**Fig. 1** Decision space, Objective space, and Pareto front in a multiobjective optimization problem with two objectives, f1 and f2, to be minimized

hyper-area known as *Pareto set*, or *Pareto front*, there is not any solution worse than any of the other ones when all the objectives are taken into account (see Fig. 1). In the dynamic multi-objective optimization problems the objective functions and the set of variables which define the search or decision space may change over time.

In this chapter, Sect. 2 defines and describes some of the main topics involved in the evolutionary dynamic multi-objective optimization problems. Then, Sect. 3 deals with the parallel processing issues of evolutionary multi-objective optimization. The parallel processing frameworks proposed in this chapter are described in Sect. 4 and 5, while the corresponding results obtained from experiments on these parallel frameworks are presented and discussed in Sect. 6. Finally, the conclusions are given in Sect. 7.

## 2 Evolutionary Dynamic Multi-objective Optimization

A dynamic multi-objective optimization (DMO) problem [7,8,9] can be defined as the problem of finding a vector of decision variables, that satisfies a restriction set and optimises a function vector whose scalar values represent objectives that change with time. Thus, it has to be found a decision variable vector $x^*(t)=\{x_1{}^*(t), x_2{}^*(t),\ldots,x_n{}^*(t)\}$ that satisfies a given restriction set $\{g(x,t)=0; h(x,t)=0\}$, and optimises the function vector: $f(x,t)=\{f_i(x,t):1=i=k\}$, where $k$ is the number of objectives, and $t$ represents the time or the dynamic nature of the problem.

In a dynamic multi-objective optimization problem, we can define $S_p(t)$ and $F_p(t)$ as the sets of Pareto optimal solutions at time $t$, respectively in the decision and objective spaces. A classification of DMO problems depending on whether the sets $S_p(t)$ and $F_p(t)$ change with time or not, is presented in [8].

To tackle DMO problems, we will consider the use of evolutionary algorithms. Evolutionary algorithms have been widely applied to multi-objective optimization, bringing a different point of view on the resolution of these problems with respect to the classic methods previously proposed. They can give a very good approximation to the Pareto set and to reveal the properties of the optimal solutions [10].

In an evolutionary algorithm a set (population) of candidate solutions (individuals) to the problem is transformed through generations (iterations) by applying the so-called genetic operators (selection, mutation, crossover, etc.) resembling the species natural evolution process. In an evolutionary algorithm, a trade off is required between exploration and exploitation of the search space. Thus, the characteristics of the genetic operators must be set in order to find a balance between the search for solutions in new areas of the space and the convergence towards better solutions in the surroundings of the already found ones.

Because of this, diversity and uniform distribution are required in the solutions in order to provide an accurate description of the Pareto set. Moreover, in dynamic optimization problems, the population of the evolutionary algorithm must react to changes as fast as possible. Some of the main topics and techniques that should be addressed in evolutionary DMO are the following ones [9]:

1. *Diversity after changes.* As soon as a change is detected, diversity in the population should be increased in order to make easier the evolution towards a new optimum. If the mutation probability is too high, the situation is similar to a re-start of the algorithm and no advantage is obtained from the already found solutions. There are some alternatives to deal with this issue: *hypermutation* [11], a sudden increment in the mutation probability after a change of the conditions, and *variable local search* [12], where mutation probability is gradually increased.
2. *Diversity along the runtime.* It tries to avoid convergence through the execution of the algorithm so that the population could better adapt itself to changes. Here the alternatives are: to insert random migrant solutions in the population in each generation; the *thermodynamic genetic algorithms* [13]; and the use of *niching techniques* [14] for preserving diversity like sharing or crowding.
3. *Memory based techniques.* The evolutionary algorithm uses a memory that keeps information about what has happened in previous generations [15,16]. This approach is mainly useful when the problem shows conditions that have appeared before.
4. *Multi-population techniques*. The population is divided into subpopulations that hold information about different regions of the decision or objective spaces [17, 18]. The idea behind this approach is to evolve different optimal solutions in each independent population. This alternative will be reviewed with more detail in the development of parallel procedures for DMO [26-32].

The availability of parallel computers, even with nodes based on multi-core microprocessors, makes parallel processing an attractive approach to accelerate the reaction to the changes in the Pareto front that dynamic problems determine. In the next section, we consider the issues on parallel processing for evolutionary multi-objective optimization [19].

## 3 Parallel Evolutionary Multi-objective Optimization

Parallel processing is useful to efficiently solve dynamic optimization problems with evolutionary algorithms [20,21], not only by improving the quality of the solutions found but also by speeding up the execution times. Two decomposition alternatives are usually implemented in parallel algorithms: functional decomposition and data decomposition.

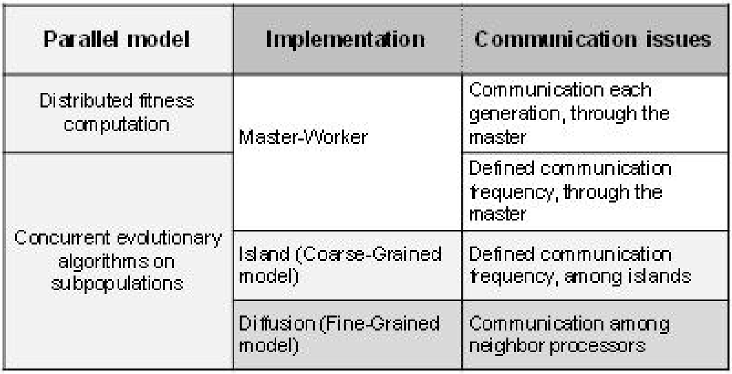
The functional decomposition techniques identify tasks that may be run separately in a concurrent way. The data decomposition techniques divide the sequential algorithm into tasks that are executed on different data (i.e. the individuals of the population). Moreover, hybrids methods are also possible.

In this paper, data decomposition has been applied as we consider this alternative more attractive and useful in many different problems, such as evolutionary computation. In an evolutionary algorithm, the evaluation of the objective function and the application of genetic operators to the individuals of the population can be independently done. As it will be seen, this allows data parallelization approaches that do not modify the convergence behaviour of the sequential algorithm. In what follows, sections 3.1 to 3.3 deal with the main issues on the parallelization of evolutionary algorithms and introduce a model to evaluate the speedup that could be achieved by different parallel models.

## *3.1 The Design Space of Data Decomposition Approaches*

Two main possible models can be considered to parallelize evolutionary algorithms in the context of the data decomposition approach (see Fig. 2): (1) the distribution of fitness computation; and (2) the concurrent execution of evolutionary algorithms over multiple subpopulations. In the first case, the individuals of the population are distributed among several processes (usually called workers) that compute the fitness values of the individuals in the corresponding subpopulation. Once each worker process completes the fitness evaluation of its subpopulation, it sends this information to another process that executes the evolutionary iteration by applying the corresponding operators to the individuals of the whole population. Then, the population is distributed again among the worker processes to obtain the new values of the fitness. It is clear that this alternative does not modify the convergence characteristics of the corresponding sequential algorithm although, to complete a given iteration, communication is required between the process that applies the evolutionary operators to the population and the processes that evaluate the fitness values of the subpopulations.

In the second alternative, several processes concurrently execute iterations of the corresponding evolutionary algorithm (including the fitness evaluation step) on different subpopulations. In this case, the convergence behavior could be different in sequential and parallel versions of the algorithms as the different subpopulations usually evolve concurrently and only exchange some information about their individuals after completing some iterations. The number of individuals in the



| Parallel model | Implementation | Communication issues |
|---|---|---|
| Distributed fitness computation | Master-Worker | Communication each generation, through the master |
| Concurrent evolutionary algorithms on subpopulations | | Defined communication frequency, through the master |
| | Island (Coarse-Grained model) | Defined communication frequency, among islands |
| | Diffusion (Fine-Grained model) | Communication among neighbor processors |

**Fig. 2** Data decomposition parallel alternatives for evolutionary algorithms

subpopulations (the grain size of the parallel procedure), and the characteristics of the communication (communication frequency, synchronous/asynchronous communication, characteristics of the data communicated, etc.) are important issues of these parallel procedures.

The previously described two parallelization alternatives can be implemented in several ways. The first approach, as it only distributes the fitness evaluation of the individuals is usually implemented by using a master-worker approximation in which the process that distributes the population and executes the rest of steps in the evolutionary iterations is executed by a processor while the other processors (the workers) evaluate the fitness of individuals in the different subpopulations. Thus, in a given iteration, the master has to communicate with the workers to distribute the population and receive the fitness values.

The second parallelization alternative (concurrent execution of evolutionary algorithms over multiple subpopulations) can be also implemented by using a master-worker scheme. The master distributes the subpopulation among the workers that communicate among themselves through the master. Nevertheless, this implementation implies that the master processor could become a bottleneck that reduces the efficiency of the parallel procedure. Thus, this parallel alternative is usually implemented by assigning each subpopulation to a different processor and communicating this processor with the others when required. According to the size of the grain (number of individuals in the subpopulations), two alternatives can be found, the island parallel approach (coarse grain) and the cellular or diffusion approach (fine grain). There are also hybrid models that include different implementation alternatives. For example, each island can be implemented by using a master-worker alternative.

The master-worker paradigm usually provides reasonable levels of speedup (sometimes even super-linear ones, as we will see in Sect. 6), but as it has been said it could suffer from a bottleneck in the communication and processing costs at the master process. Due to this circumstance, researchers have tried to develop a fully distributed algorithm for multi-objective optimization.

Unfortunately this task is not as easy to achieve as it could be in other kind of optimization algorithms. The reason is that evolutionary optimization behaves better when the underlying algorithm is searching in a set of tentative solutions at the same time. But due to the fact that in a fully distributed algorithm the processes should work independently, an issue arises on how to redistribute the search space among the processes. This is considered in the following section 3.2.

## *3.2 Search Space Distribution in Evolutionary Parallel Procedures*

An important issue in the parallel model of concurrent execution of evolutionary algorithms over multiple populations is the distribution of individuals among those different subpopulations. Basically, there are two options:

1. Every process uses the whole search space. This option is similar to run the sequential algorithm a number of times equal to the number of workers, although with less individuals in the population.
2. Every process explores a different part of the search space.

The second option is the ideal one, because it would allow the best use of the resources by avoiding that more than one process searches on the same area. However, it is also very hard to develop a working procedure that enforces that each process searches only on a specifically limited and independent area. There is also a hybrid approach where every process tries to focus on an area while overlapping between processes is allowed but somehow discouraged. This second approach is fairly possible when a cellular algorithm is employed, because every process is working on only one solution at any time. But even in this case it is difficult to restrict the search area of every process. Some researchers have tried to find an elegant and practical way to deal with that mixed approach where processes focus on some part of the search space but some overlapping may occur. Nevertheless, the overlap should be low.

The distribution of solutions among the subpopulations that define the different islands can be obtained in several ways. For example it is possible to divide the objective space by assigning the individuals in a given zone of the Pareto front to the same island. Another alternative consists on assigning individuals in different partitions of the search or decision space to different islands. In all these approaches, the local information is included into the global optimization procedure by dividing the search space into areas where the corresponding evolutionary algorithm is applied by each processor. Procedures such as [26,27,28], and our frameworks pdMOEA and pdMOEA+ described below (Sect.s 4 and 5), divide the objective space, while [29] divides the decision space.

In the papers [26,28], the authors propose an island model where the worker processes (the islands) can search in the whole decision space, although each worker is limited to a different part of the Pareto front. This restriction in the objective space is done implicitly by using a guided domination technique. The main drawback of this approach is that to work properly it requires to know the shape of the multi-objective problem that is to be solved.

Another proposal that provides a MOEA able to search at different space areas is given in [27]. This procedure divides geometrically the objective space by cone separation, and each part of the objective space is assigned to a different subpopulation in a processor. The problem behind this approach is that it is not good for objective spaces with more than two dimensions.

In [30], the authors use a similar approach to the pdMOEA framework of Sect. 4. They use a clustering algorithm where after a number of generations the MOEA combines the sub-populations and clusters them with the K-means algorithm, commonly used in many applications [31,32]. Then, the new sub-populations are partitioned and sent to the worker processes. Nevertheless, in [30] some global computation must be done for clustering and subpopulation redistributing.

Finally, in [29] it is proposed to distribute the decision space among the different processes by locating to each process an adaptive sphere including points of the corresponding decision space volume. The main difference of this framework with respect to the previously described procedures (included our procedures pdMOEA and pdMOEA+), is that in [29] the partition takes place in the decision space instead of the objective space. In addition, it uses particle swarm optimization algorithms to guide the spheres in the search space instead of using algorithms such as K-means to cluster the solutions.

As it has been said earlier, we propose a framework that allowed the processes to work wholly independent from a central master process. This approach can be considered a hybrid of the proposals described in [29,30]. The reason to use a cluster algorithm is that we feel that in this way we could respond to any shape that a problem could have, including discontinuous fronts. This is so important that some researchers even relied on the knowledge of the Pareto front [26]. Our proposed distributed algorithm, that has been called pdMOEA+, locates a centroid to every process. The centroids are intended to keep themselves far enough and try to approach the Pareto front.

### *3.3 A Simple Model to Estimate the Speedups*

The selection of individuals and the diversity maintenance operations in a evolutionary multiobjective algorithm require comparisons that imply the whole population or a big part of it. This means that data parallelization by the concurrent execution of evolutionary algorithms on subpopulations, especially in the case where there is not any mechanism to share information about the fitness of the individuals between the processes, modifies the behavior of the algorithm with regard to the sequential version. Most of the time, it is difficult to predict the behavior of this kind of parallelization and must be evaluated for each particular implementation. So, the initial population is divided into subpopulations associated to different search spaces which are evolved separately. Sometimes, individuals can be exchanged between the subpopulations (migration). This kind of parallelization could improve the diversity of the population during the algorithm convergence and lead to algorithms with better performance than the sequential versions. So, together with advantages from the bigger availability of memory and CPU, the evidences of bigger efficiency and diversity in the population justify the use of parallelism in the field of evolutionary algorithms.

After the evaluation of the objective functions, the algorithms with Pareto front-based selection usually calculate dominance and the corresponding distances as part of the mechanism for keeping diversity. This mechanism is implemented in each case, as a previous step to assign the fitness value to each individual and to select the parents. The parallelization of these tasks is not easy. For example, problems appear in algorithms that usually work with small populations (PAES) [22], in algorithms where the calculation of distances must be done sequentially after the determination of dominance relationships (PSFGA) [23], or in those algorithms where the calculation of dominance relationships, distance, and selection takes place at the same time (NPGA) [24].

Equation (1) provides an estimation of the processing time for *gen* iterations of a sequential evolutionary multi-objective algorithm:

$$T_1 = gen(kNt_{fitness} + kN^2 t_{non_{d}om} + Nt_{evolution} + N^r t_{niching}) \ , \qquad (1)$$

The terms in parenthesis in equation (1) respectively correspond to the evaluation of the fitness for a population of $N$ individuals in a problem with $k$ objectives; the determination of the non-dominated individuals (it requires to compare the individuals of the population by using their $k$ objectives); the application of the evolutionary operators (mutation, crossover, etc.) to the $N$ individuals of the population or a subset of the individuals of the population; and the application of a procedure to maintain the distribution of individuals along the present Pareto front (the complexity of these operations is taken into account through the parameter $r$). The parameters $t_{fitness}$, $t_{non-dom}$, $t_{evolution}$ and $t_{niching}$ determine the relative weights of these different terms.

Equation (2) provides a generic expression for the computing time in a parallel implementation executed on P processors.

$$T_p = \delta(par,1)(gen((kN^2 t_{non_{d}om} + Nt_{evolution} + N^r t_{niching}) + (k\tfrac{N}{P}t_{fitness}))) + \\ t_{comm}(N,P) + (1-\delta(par,1))(genser(kNt_{fitness} + kN^2 t_{non-dom} + Nt_{evolution} + \\ N^r t_{niching}) + genpar(k\tfrac{N}{P}t_{fitness} + k(\tfrac{N}{P})^2 t_{non-dom} + \tfrac{N}{P}t_{evolution} + (\tfrac{N}{P})^r t_{niching})) \ , \qquad (2)$$

In equation (2), *$\delta$(i,j)=1* if *i=j* and *$\delta$(i,j)=0* otherwise, and *par* is used to take into account the parallelization alternative (see Fig. 2). The case *par=1* corresponds to a parallel model that distributes the fitness computation implemented through a master-worker parallel procedure, while whenever *par$\neq$1*, the parallel model is the concurrent execution of evolutionary algorithms over multiple subpopulations. The parameters *genser* and *genpar* in $T_p$ correspond, respectively, to the number of generations executed in a master processor and in each of the worker processors where the population has been divided into *N/P* individuals. If *genser=0*, a *pure* island model is used to parallelize the algorithm, while if *genser$\neq$0*, we have a master-worker implementation, as it is supposed that after executing *genpar* iterations in parallel, the concurrent evolutionary algorithms communicate themselves through a master process that executes *genser* iterations. Thus, we can set different values for *genser* and *genpar* in our parallel procedure to implement an island model that allows the communication among the subpopulations through a master. The term $t_{comm}$*(N,P)* corresponds to the communication cost, and it depends on the parallel model used, on the amount of individuals that processors exchange (a function of $N$) and on the number (and communication topology) of processors that have to communicate themselves (a function of $P$).

This simple model (equations (1) and (2)) allows the explanation of different speedup ($S=T_1/T_p$) behaviours [25]. Thus, if *genser+genpar* $<$ *gen*, it is possible to observe super-linear speedups (as in curves 1 and 2 of Fig. 3). This situation could appear whenever the parallel evolutionary algorithm provides, for example, better diversity conditions than the sequential implementation and a lower number of iterations is required by the parallel algorithm to get a solution with similar quality that the one obtained by the sequential algorithm.

Moreover, the effect of the communication cost, $t_{comm}(N,P)$, can be also shown. Thus, in Fig. 3, as the number of processors $P$ increases, the speedup is lower in curve 2 than in curve 1. The communication cost is higher for curve 2, although in this curve 2 *genpar* is higher and *genser+genpar* is lower than in curve 1. In Fig. 3, curve 3 corresponds to a case where *genser+genpar>gen*.

In [25], a taxonomy is proposed for speedup measurements in evolutionary algorithms. This taxonomy distinguishes between the Class I or strong speedup measurements, which compare the execution times of the parallel evolutionary algorithms and the better known sequential algorithm for the problem at hand; and the Class II measurements that compare the parallel algorithm with its own sequential version executed in only one processor. Inside the Class II measurements, it is also possible to distinguish between other two types of measurements according to the way the algorithm finishes. The group A includes the measurements obtained if the algorithms finish when solutions of similar qualities are found by both, the parallel and sequential algorithms. Whenever the measures are obtained by setting a similar number of iterations for the sequential and the parallel algorithms we have the group B of measurements. As it can be seen, in the performance model described in this section the speedup measurement considered belong to the class B and group II.

In what follows (Sect. 4, 5, and 6), we describe the frameworks pdMOEA, pdMOEA+ and analyze their performances.

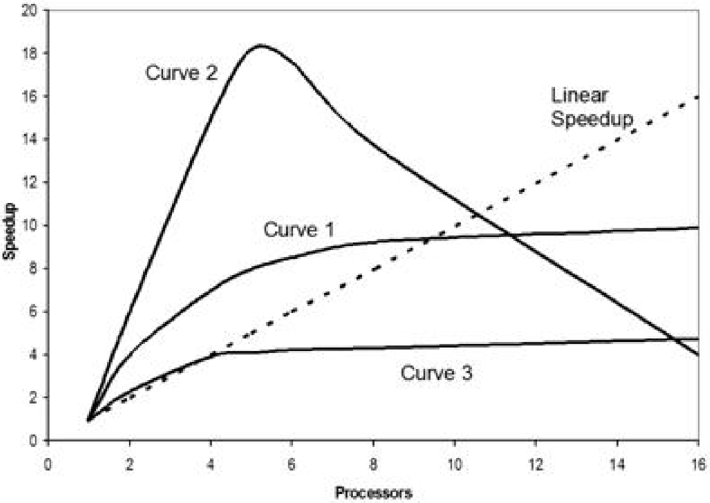
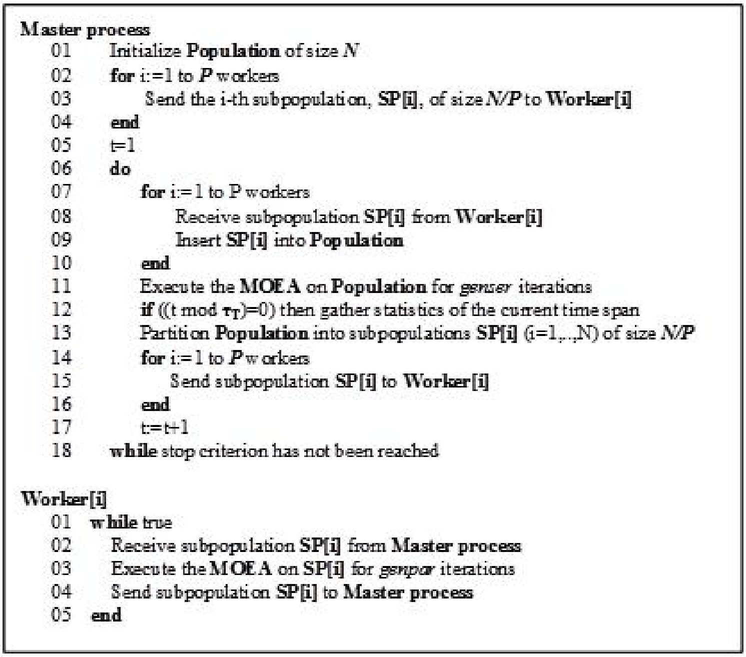


**Fig. 3** Different Speedup behaviors

## 4 A Generic Master-Worker Evolutionary Framework for DMO

The proposed generic master-worker evolutionary framework for DMO is described in Fig. 4. It is called pdMOEA, from *parallel dynamic MOEA*. It is a parallel framework for multi-objective optimization that allows the execution of the both parallel models previously described in section 3.1 (i.e. the distributed fitness computation and the concurrent execution of evolutionary algorithms on subpopulations) by using a master-worker implementation. It divides the population to send subpopulations of the same size to each worker process. For comparison purposes the parallel algorithm has been generalised in order to be able to run and test different multi-objective evolutionary algorithms.

In this generalised version, every worker searches, with the chosen multi-objective evolutionary algorithm (MOEA), the optimal solutions in the search space that has been assigned to it, and keeps only those solutions that are not dominated by the others. However, in this parallel procedure the workers share the same search space. After a fixed number of iterations (*genpar*), the workers send the solutions found to the master, who after gathering all the solutions into a new population, runs an instance of the MOEA (along *genser* iterations) over the whole population before sending new subpopulations again to the worker processes.



```
Master process
    Initialize Population of size N
    for i:=1 to P workers
        Send the i-th subpopulation, SP[i], of size N/P to Worker[i]
    end
    t=1
    do
        for i:=1 to P workers
            Receive subpopulation SP[i] from Worker[i]
            Insert SP[i] into Population
        end
        Execute the MOEA on Population for genser iterations
        if ((t mod $\tau_T$)=0) then gather statistics of the current time span
        Partition Population into subpopulations SP[i] (i=1,..,N) of size N/P
        for i:=1 to P workers
            Send subpopulation SP[i] to Worker[i]
        end
        t:=t+1
    while stop criterion has not been reached

Worker[i]
    while true
        Receive subpopulation SP[i] from Master process
        Execute the MOEA on SP[i] for genpar iterations
        Send subpopulation SP[i] to Master process
    end
```

**Fig. 4** Description of pdMOEA

The use of this generalised parallel dynamic MOEA (pdMOEA) in DMO allows either the execution of more optimization iterations (*genser* or *genpar*) for a given amount of time (thus exploring more search space and improving the quality of the solutions found), or to speed up the convergence (thus allowing the approach to dynamic problems with higher rates of change).

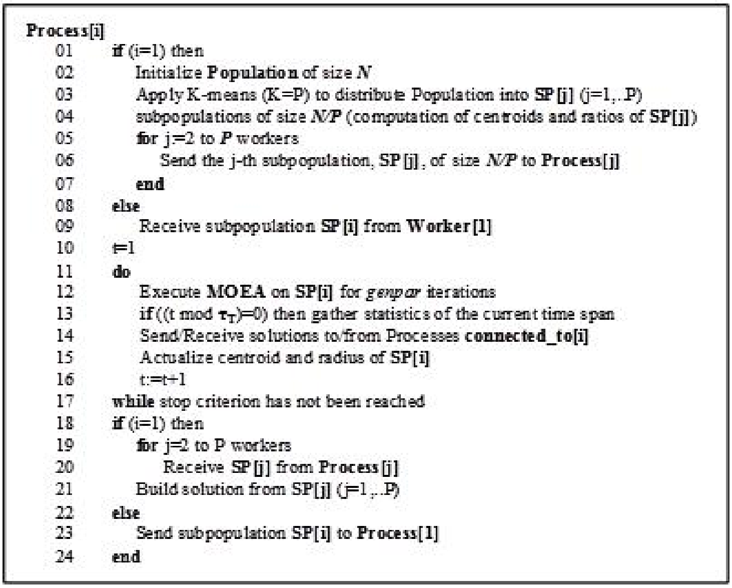
The EAs are implemented with all the required initialisation code outside the main function in order to offer a continuous model of execution, where the population used in the last generation will be intact for the next generation. Furthermore, each MOEA implementation may differ in which subpopulation is sent; for example, depending on the implemented algorithm it can be an exact copy of the current population or a copy of the algorithm archive, but for simplicity in Fig. 4 it is represented just as SP[i] when receiving and sending.

In order to carry out a thorough comparison of this parallel framework, in this paper, we will use as MOEAs our previously proposed SFGA [23] and SFGA2 [7], along with two of the best known MOEAs: SPEA2 [33] and NSGA-II [34].

## 5 A Fully Distributed Framework for DMO

Fig. 5 provides a description of another generic distributed evolutionary framework for DMO. It has been called pdMOEA+ as it tries to achieve better performance by avoiding a possible bottleneck caused by the master processor in pdMOEA. This way, a coarse grained island implementation of the parallel model that uses concurrent evolutionary algorithms on different subpopulations is provided by pdMOEA+.

Each process in pdMOEA+ uses a centroid and a sphere to define the volume in the objective space that includes the individuals of the subpopulation allocated to the process. Firstly, the radii of the spheres must be adjusted so that every sphere contains some solutions within it. In order to do this, the MOEA is executed for a small number of iterations, for example 50, and with the solutions obtained after these 50 iterations have been run, the centroid for each process is adapted to the corresponding nearest solutions with an appropriate radius. After that, the algorithm is run and the centroid is updated subsequently by each process, after executing a given number of iterations (*genpar*) of the corresponding MOEA. The line 14 in Fig. 5 can be implemented according to the different alternatives that define the way migration is implemented and the topology of connection among processes described in the functions *connected_to[i]*. We propose a migration scheme in which individuals that are farest from the centroid of the subpopulation located in a given processor, are asynchronously sent to the other processors, along with the corresponding distance to that centroid. The processor that receives an individual, includes it in their subpopulation, or not, according to its distance to the centroid located in the processor. There are also several alternatives to actualize the centroid and radius of the subpopulation in each processor (line 15). In our case, the center of mass of the individuals in each subpopulation has been considered the centroid, and the longest distance from the centroid to the individuals in the subpopulation determines the radius.



```
Process[i]
if (i=1) then
    Initialize Population of size N
    Apply K-means (K=P) to distribute Population into SP[j] (j=1,..P)
    subpopulations of size N/P (computation of centroids and ratios of SP[j])
    for j=2 to P workers
        Send the j-th subpopulation, SP[j], of size N/P to Process[j]
    end
else
    Receive subpopulation SP[i] from Worker[1]
t=1
do
    Execute MOEA on SP[i] for genpar iterations
    if ((t mod $\tau_T$)=0) then gather statistics of the current time span
    Send/Receive solutions to/from Processes connected_to[i]
    Actualize centroid and radius of SP[i]
    t:=t+1
while stop criterion has not been reached
if (i=1) then
    for j=2 to P workers
        Receive SP[j] from Process[j]
    Build solution from SP[j] (j=1,..P)
else
    Send subpopulation SP[i] to Process[1]
end
```

**Fig. 5** Description of pdMOEA+

At the end of pdMOEA+, as it described in Fig. 5, the solutions found are sent by each process (i=2,..P in Fig. 5) to one of them (i=1 in Fig. 5) that would gather all the solutions, build the final Pareto set, and later would store them into a file. However, with the advanced capabilities found in parallel libraries such as MPI 2, distributed I/O it is possible, and every process can directly store their solutions into a file. Certainly, this could imply that some solutions in the file would be dominated by others, but on the other hand, the processes would run at their fastest pace.

## 6 Experimental Results

In this section, both proposed parallel frameworks, pdMOEA and pdMOEA+, are analyzed with regard to their experimental results in different test cases. The experiments have been carried out on an 8-node cluster with two 2 GHz AMD Athlon processors and 2 Gbytes RAM by node, connected via Gigabit Ethernet. The data shown in the tables has been gathered after running the parallel procedure in 1, 2, 4 and 8 worker processors for each of the MOEAs: SFGA, SFGA2, SPEA2 and NSGA-II. The code is implemented in C++ with MPI. SPEA2 and NSGA-II were added anew from the implementations kept in the authors' sites.

The MOEA parameters were set to those values that showed the best results and to those that are the commonly used among researchers. These values are the following ones: *master population*=800 individuals; *crowding distance*=0.0075, for SFGA and SFGA2; the iterations executed in pdMOEA are *gen*=200, *genser*=50 and *genpar*=150; five executions have been done for each algorithm and number of workers; *simulated binary crossover (SBX)* with $\eta_c$=15 [35] and *crossover probability*=0.75; *polynomial mutation* with $\eta_m$=20 [36] and *mutation probability*=1/(*number of decision variables*).

For the sake of the evaluation of DMO algorithms, the FDA set of functions [8] has gained the highest relevance among the researchers in this field. In this section, our results have been obtained by using the benchmarks FDA1 to FDA5. More specifically, the FDA2 and FDA3 benchmarks proposed in [8] have been substituted by their versions FDA2-mod and FDA3-mod, described in [37], to avoid some difficulties with their firstly proposed versions in [8].

The parameter $\tau_i$ in the FDA functions corresponds to the time instant where the evaluation of the function takes place. Due to the long running time of some of the MOEAs, only data from $\tau_i$=1 to $\tau_i$=20 have been taken into account. Moreover, the parameter $\tau_T$ sets the time period with stationary objective functions. As in all our examples, we have used $\tau_T$=5, this means that every five time instants there is a change in the current Pareto set [8], and the solutions should be recalculated.

Table 1 shows the cumulative time of the execution of the algorithms for different number of worker processes and benchmarks. The resulting speedups reached by the parallel algorithm are provided in Table 3. In order to allow the biggest stability in the algorithms, Tables 2 and 3 provide the results achieved by each algorithm after completing 20 time instants of the function, i.e. $\tau_i$=20. It can be seen in Table 3 that super-linear speedup was achieved for some runs of SPEA2 and NSGA-II. This behavior can be explained by the model proposed in Sect. 3.3. The communication cost modeled in this case is linear with the number of processors. As the parallel algorithm allows more diversified populations, it may have to do with the achieved improvement in the performance, particularly with the observed super-linear behavior.

Nevertheless, from Table 1, it is clear that the running times of the SFGA family of algorithms are always much smaller than the running times of SPEA2 and NSGA-II algorithms. Two different behaviors can be outlined:

1. First of all, for the FDA2-mod and FDA3-mod functions, SFGA and SFGA2 are at least one order of magnitude faster than SPEA2 and NSGA-II. This is a very important advantage for SFGA and SFGA2 in comparison to those state-of-the-art MOEAs. In addition, for these two functions, SPEA2 and NSGA-II are on a par with respect to the running times.
2. Moreover, for the three-objective functions FDA4 and FDA5, the SFGA and SFGA2 prove themselves as the fastest algorithms. However, in this case they are only four times faster than SPEA2. It is important to note that NSGA-II behaves very badly in terms of the running time for these functions, and once more it is one order of magnitude slower than the family of SFGA algorithms.

With regard to the speedups, shown in Table 2, it can be seen that:

1. SFGA and SFGA2 achieve slightly better speedup results for FDA4 and FDA5 than those obtained for FDA1, FDA2-mod and FDA3-mod.

**Table 1** Running time (in seconds) for pdMOEA until $\tau_i = 20$

| Problem | Workers | SFGA | SFGA2 | SPEA2 | NSGA-II |
|---|---|---|---|---|---|
| FDA1 | 1 | 49.3±0.3 | **44.9±0.8** | 1318.3±9.4 | 987.1±52.3 |
| | 2 | 46.3±0.8 | **35.2±0.7** | 381.3±13.1 | 364.4±27.6 |
| | 4 | 28.5±0.3 | **26.3±0.3** | 230.6±2.1 | 208.0±7.6 |
| | 8 | 17.1±0.1 | **15.5±0.1** | 186.3±0.9 | 121.5±2.2 |
| FDA2-mod | 1 | 138.3±2.7 | **64.67±2.3** | 3087.0±36.0 | 2351.0±46.0 |
| | 2 | 162.0±3.6 | **46.18±4.1** | 931.8±31.9 | 756.1±32.1 |
| | 4 | 79.75±1.2 | **40.64±2.7** | 683.7±13.0 | 422.7±11.3 |
| | 8 | 60.71±1.0 | **34.25±1.4** | 629.5±12.1 | 455.1±20.1 |
| FDA3-mod | 1 | 125.9±3.6 | **68.3±2.0** | 2477.0±34.0 | 2534.0±50.0 |
| | 2 | 93.5±1.1 | **45.9±1.3** | 888.0±21.0 | 757.3±31.5 |
| | 4 | 74.9±0.9 | **38.8±1.6** | 831.2±15.7 | 493.7±5.3 |
| | 8 | 58.9±1.0 | **34.6±0.5** | 683.9±8.3 | 631.0±13.5 |
| FDA4 | 1 | **641.9±15.1** | 776.1±7.5 | 2733.0±37.0 | 23774.0±89.0 |
| | 2 | **291.2±8.1** | 353.1±4.7 | 1089.0±13.0 | 3793.0±73.0 |
| | 4 | **198.7±6.9** | 238.5±3.1 | 805.3±9.9 | 2285.0±40.0 |
| | 8 | **169.9±6.2** | 208.5±3.2 | 689.6±8.6 | 2439.0±53.0 |
| FDA5 | 1 | **746.6±13.8** | 921.7±8.5 | 2721.0±43.0 | 22446.0±71.0 |
| | 2 | **329.6±9.1** | 411.0±6.3 | 1102.0±17.0 | 4001.0±69.0 |
| | 4 | **216.7±4.6** | 285.7±5.2 | 1102.0±17.0 | 2371.0±46.0 |
| | 8 | **189.2±4.3** | 246.8±4.9 | 728.0±9.1 | 2390.0±44.0 |

2. SPEA2 speedup results for FDA4 and FDA5 are almost the same that the ones obtained by SFGA and SFGA2.
3. NSGA-II shows super linear speedups for four workers. These are also the maximum peaks of its speedup for all the functions.
4. All the algorithms but NSGA-II show increasing speedup values as the number of workers increases.

Table 2 also shows that speedups for SFGA and SFGA2, which are indeed very close to each other, are not as good as the speedups shown by SPEA2 (in FDA1, FDA2, and FDA3), and by NSGA-II. But it should be kept in mind that the cumulative time needed for the execution of NSGA-II and, especially, SPEA2 are, by far, bigger than the time needed by SFGA and SFGA2. Thus, Table 1 shows that the SFGA family algorithms differ in more than one order of magnitude to the times needed by the SPEA2 and NSGA-II.

Table 3 shows the number of Pareto optimal solutions found by each algorithm. The sets of Pareto optimal solutions found by each algorithm represent very good approximations to the real Pareto front. It can be seen that SPEA2 is the algorithm

**Table 2** Speedups for pdMOEA until $\tau_i = 20$

| Problem | Workers | SFGA | SFGA2 | SPEA2 | NSGA-II |
|---|---|---|---|---|---|
| FDA1 | 2 | 1.10 | 1.30 | **3.50** | 2.70 |
| | 4 | 1.73 | 1.70 | **5.72** | 4.75 |
| | 8 | 2.88 | 2.90 | 7.08 | **8.12** |
| FDA2-mod | 2 | 0.85 | 1.40 | **3.31** | 3.11 |
| | 4 | 1.73 | 1.59 | 4.51 | **5.56** |
| | 8 | 2.28 | 1.89 | 4.91 | **5.17** |
| FDA3-mod | 2 | 1.35 | 1.49 | 2.79 | **3.35** |
| | 4 | 1.68 | 1.76 | 2.98 | **5.13** |
| | 8 | 2.14 | 1.97 | 3.62 | **4.02** |
| FDA4 | 2 | 2.20 | 2.20 | 2.51 | **6.27** |
| | 4 | 3.23 | 3.25 | 3.39 | **10.40** |
| | 8 | 3.78 | 3.72 | 3.96 | **9.75** |
| FDA5 | 2 | 2.26 | 2.24 | 2.47 | **5.61** |
| | 4 | 3.44 | 3.22 | 2.46 | **9.46** |
| | 8 | 3.94 | 3.73 | 3.74 | **9.39** |

that obtains by far the highest number of non-dominated solutions, but it does so at the cost of employing the biggest time among all the algorithms (see Table 1).

Furthermore, in Table 3 there is also a compilation of measures consisting on the number of non-dominated solutions found by each algorithm divided by the time needed for that algorithm. This measure cannot be used to indicate whether a certain algorithm is better in terms of quality of the solutions to a multi-objective problem, be it in diversity of the solutions or closeness to the actual Pareto set, but on the other hand, this measure can be useful in DMO. This is because it can indicate a certain advantage of one algorithm over another. The advantage relies on that the superior algorithm could be able to find more solutions per time unit in comparison with the other algorithm. Although this advantage does not imply directly that solutions found by that algorithm had to be better than those found by other algorithms according to performance indicators commonly used in multi-objective optimization, such as hypervolume (Table 4), having more solutions per time unit is a desired feature of any algorithm meant to be used in DMO.

It is clear that our algorithms SFGA and SFGA2 do not expose super-linear speedups when adding more processors. However, they gave more non-dominated solutions per time unit, which can be seen as an improvement in data throughput instead of time speedup (see Table 3). It is worth reminding that in DMO it is common that the algorithm has to meet strict time restrictions, and so, the possibility of having more solutions in less time is seen as a preferred feature and trade-off over

**Table 3** Solutions and (Solutions per time, in seconds$^{-1}$) for pdMOEA with $\tau_i = 20$

| Problem | Workers | SFGA | SFGA2 | SPEA2 | NSGA-II |
|---|---|---|---|---|---|
| FDA1 | 1 | 115(2.33) | 310(**6.91**) | **800**(0.61) | 370(0.37) |
| | 2 | 142(3.07) | 436(**12.41**) | **800**(2.10) | 370(1.02) |
| | 4 | 143(5.03) | 320(**12.20**) | **800**(3.47) | 300(1.44) |
| | 8 | 136(8.00) | 230(**14.91**) | **800**(4.30) | 240(1.98) |
| FDA2-mod | 1 | 105(0.76) | 77(**1.19**) | **800**(0.26) | 268(0.11) |
| | 2 | 138(0.85) | 149(**3.23**) | **800**(0.86) | 280(0.37) |
| | 4 | 136(1.71) | 197(**4.85**) | **800**(1.17) | 250(0.59) |
| | 8 | 125(2.06) | 175(**5.11**) | **800**(1.27) | 185(0.41) |
| FDA3-mod | 1 | 77(0.61) | 61(**0.89**) | **800**(0.32) | 252(0.10) |
| | 2 | 93(0.99) | 83(**1.81**) | **799**(0.90) | 248(0.33) |
| | 4 | 99(1.32) | 144(**3.71**) | **799**(0.96) | 221(0.45) |
| | 8 | 106(1.80) | 172(**4.79**) | **800**(1.17) | 212(0.34) |
| FDA4 | 1 | 800(**1.25**) | 800(1.03) | 800(0.29) | **981**(0.04) |
| | 2 | 800(**2.75**) | 800(2.27) | 800(0.73) | **968**(0.26) |
| | 4 | 800(**4.02**) | 800(3.35) | 800(0.99) | **995**(0.44) |
| | 8 | 800(**4.71**) | 800(3.84) | 800(1.16) | **993**(0.41) |
| FDA5 | 1 | 800(**1.07**) | 800(0.87) | 800(0.29) | **1077**(0.05) |
| | 2 | 800(**2.43**) | 800(1.95) | 800(0.73) | **1091**(0.27) |
| | 4 | 800(**3.69**) | 800(2.80) | 800(0.73) | **1084**(0.46) |
| | 8 | 800(**4.23**) | 800(3.24) | 800(1.10) | **1123**(0.47) |

that of having more accurate solutions but at the cost of employing much more time. Therefore, it is expected that SFGA and SFGA2 can cope with more restrictive time limits without having to reduce the population because they have a smaller runtime and produce more solutions per time unit in comparison to NSGA-II and SPEA2. It is interesting to note that Table 3 shows that for FDA4 and FDA5 (problems involving more than two objectives) all the algorithms obtain always 800 solutions or even more in the case of NSGA-II.

In Table 4 the quality in terms of the hypervolume of the solutions found by the different algorithms is shown. Hypervolume gives the area covered by the solutions found, from a given reference point in the objective space, so in minimization problems, as the FDA benchmarks are, the hypervolume is to be maximized. It can be seen that the best quality was attained by the SPEA2 algorithm for FDA1, FDA2-mod, and FDA3-mod; by NSGA-II for FDA4; and for SFGA2 for FDA5. Anyway, the four algorithms show very similar results in terms of quality, according to this hypervolume indicator, except for SFGA in FDA4 and FDA5, and for SPEA2 in FDA5. It is important to note that when more worker processes were added the quality did not appreciably worse and it even improved in some cases for the SFGA

**Table 4** Hypervolume for pdMOEA until $\tau_i = 20$

| Problem | Workers | SFGA | SFGA2 | SPEA2 | NSGA-II |
|---|---|---|---|---|---|
| FDA1 | 1 | 0.64±0.00 | 0.63±0.02 | **0.66±0.01** | 0.65±0.00 |
| | 2 | 0.65±0.00 | 0.65±0.01 | **0.66±0.00** | 0.65±0.00 |
| | 4 | 0.65±0.00 | 0.65±0.01 | **0.66±0.00** | 0.65±0.00 |
| | 8 | 0.65±0.00 | 0.65±0.00 | **0.66±0.00** | 0.65±0.00 |
| FDA2-mod | 1 | 0.77±0.04 | 0.74±0.04 | **0.83±0.00** | 0.81±0.01 |
| | 2 | 0.79±0.03 | 0.77±0.04 | **0.83±0.00** | 0.80±0.01 |
| | 4 | 0.80±0.01 | 0.77±0.05 | **0.83±0.01** | 0.80±0.01 |
| | 8 | 0.80±0.02 | 0.79±0.03 | **0.83±0.00** | 0.80±0.01 |
| FDA3-mod | 1 | 3.72±0.02 | 3.70±0.03 | **3.75±0.00** | 3.74±0.01 |
| | 2 | 3.73±0.01 | 3.70±0.05 | **3.75±0.00** | 3.74±0.01 |
| | 4 | 3.73±0.01 | 3.72±0.02 | **3.75±0.00** | 3.74±0.01 |
| | 8 | 3.73±0.01 | 3.73±0.01 | **3.75±0.00** | 3.74±0.01 |
| FDA4 | 1 | 1.09±0.08 | 1.37±0.00 | 1.37±0.00 | **1.38±0.00** |
| | 2 | 1.03±0.10 | 1.37±0.01 | 1.36±0.01 | **1.38±0.00** |
| | 4 | 1.08±0.08 | 1.37±0.00 | 1.36±0.01 | **1.38±0.00** |
| | 8 | 1.13±0.03 | 1.37±0.00 | 1.36±0.01 | **1.38±0.00** |
| FDA5 | 1 | 5.85±0.09 | **6.77±0.00** | 6.03±0.07 | 6.25±0.00 |
| | 2 | 5.93±0.05 | **6.77±0.00** | 5.94±0.13 | 6.25±0.00 |
| | 4 | 5.85±0.12 | **6.75±0.01** | 5.96±0.12 | 6.25±0.00 |
| | 8 | 5.98±0.03 | **6.75±0.01** | 6.07±0.05 | 6.25±0.00 |

and SFGA2 algorithms. However, it cannot be stated that the more workers used, the better the hypervolume.

The framework pdMOEA+, described in Sect. 5, has been tested with the same MOEAs and test functions as it was tested the pdMOEA approach. Nevertheless, in the case of a fully distributed procedure, the tests have been carried out only for four and eight processes. The reason behind this choice is that the pdMOEA+ procedure should show a better behavior mainly when many workers are involved and so, it is not relevant the performance of the algorithm with only two processes. Because of that only results for 4 and 8 proceses are reproduced in Table 5 and Table 6. The parameters used in the experiments with pdMOEA+ take the same values that in the experiments with pdMOEA, the initial value for the radius has been set to 0.35.

Unfortunately, the obtained results were not as good as expected. In terms of the quality of the obtained solutions, the pdMOEA+ results have been clearly worse than those results given by pdMOEA. The framework pdMOEA+ also produces less number of solutions for some combinations of algorithms and number of processes, while it provides more solutions for other combinations. The numbers of solutions for each problem and MOEA are collected in Table 6.

**Table 5** Running time (in seconds) for pdMOEA+ until $\tau_i = 20$

| Problem | Workers | SFGA | SFGA2 | SPEA2 | NSGA-II |
|---|---|---|---|---|---|
| FDA1 | 1 | 49.3±0.3 | **44.9±0.8** | 1318.3±9.4 | 987.1±52.3 |
| | 4 | 14.1±0.7 | **8.7±0.5** | 39.2±1.3 | 67.6±1.7 |
| | 8 | 4.9±0.3 | **4.8±0.3** | 9.4±0.4 | 24.4±0.9 |
| FDA2-mod | 1 | 138,3±2,7 | **64.67±2.3** | 3087.0±36.0 | 2351.0±46.0 |
| | 4 | 18.3±0.6 | **9.0±0.4** | 38.9±1.1 | 53.8±1.2 |
| | 8 | 12.8±0.5 | **6.9±0.4** | 27.5±0.8 | 42.0±0.9 |
| FDA3-mod | 1 | 125.9±3.6 | **68.3±2.0** | 2477.0±34.0 | 2534.0±50.0 |
| | 4 | 15.7±0.4 | **9.0±0.4** | 91.7±1.3 | 59.1±1.0 |
| | 8 | 16.1±0.5 | **12.8±0.4** | 27.0±0.7 | 45.9±0.6 |
| FDA4 | 1 | **641.9±15.1** | 776.1±7.5 | 2733.0±37.0 | 23774.0±89.0 |
| | 4 | **18.6±0.5** | 27.8±0.5 | 42.6±0.8 | 195.2±2.1 |
| | 8 | **16.3±0.5** | 21.6±0.8 | 29.8±0.9 | 120.9±1.4 |
| FDA5 | 1 | **746.6±13.8** | 921.7±8.5 | 2721.0±43.0 | 22446.0±71.0 |
| | 4 | **20.5±0.9** | 31.3±1.0 | 36.9±0.7 | 177.6±1.8 |
| | 8 | **17.2±0.6** | 23.3±0.8 | 30.8±0.5 | 45.7±0.9 |

**Table 6** Solutions and (Solutions per time, in seconds$^{-1}$) for pdMOEA+ with $\tau_i = 20$

| Problem | Workers | SFGA | SFGA2 | SPEA2 | NSGA-II |
|---|---|---|---|---|---|
| FDA1 | 1 | 115(2.33) | 310(**6.91**) | **800**(0.61) | 370(0.37) |
| | 4 | 138(9.79) | 226(**25.98**) | 30(20.41) | **251**(3.71) |
| | 8 | **313**(**63.88**) | 289(60.21) | 27(2.87) | 308(12.62) |
| FDA2-mod | 1 | 105(0.76) | 77(**1.19**) | **800**(0.26) | 268(0.11) |
| | 4 | 111(6.07) | 122(**13.56**) | 21(0.54) | **203**(3.77) |
| | 8 | 120(9.38) | 109(**15.90**) | 42(1.53) | **270**(6.42) |
| FDA3-mod | 1 | 77(0.61) | 61(**0.89**) | **800**(0.32) | 252(0.10) |
| | 4 | 96(6.11) | 105(**11.67**) | 36(0.39) | **192**(3.25) |
| | 8 | 257(15.96) | 292(**22.81**) | 41(1.52) | **311**(6.78) |
| FDA4 | 1 | 800(**1.25**) | 800(1.03) | 800(0.29) | **981**(0.04) |
| | 4 | **540**(**29.03**) | 67(2.41) | 64(1.50) | 73(0.37) |
| | 8 | **288**(**17.67**) | 107(4.95) | 91(3.05) | 68(0.56) |
| FDA5 | 1 | 800(**1.07**) | 800(0.87) | 800(0.29) | **1077**(0.05) |
| | 4 | 153(**7.46**) | 126(4.03) | 55(1.49) | **166**(0.93) |
| | 8 | 216(**12.56**) | **222**(9.53) | 73(2.37) | 164(3.59) |

Table 5 shows that the pdMOEA+ approach provides better running time improvements than the pdMOEA approach (see Table 1) as more workers are employed. Nonetheless, the improvement shown by these results should be taken into account with care. The reason is that, as it has been said before, for some of the tests, the number of found solutions has decreased with respect to the pdMOEA and to the sequential run of the algorithms. Thus, if the MOEAs are producing a lower number of solutions, they need less time to compute them.

Therefore, it can be said that these results did not show enough improvement neither in the quality of the solutions nor in speedup. The reason behind these results is that the processes converge towards very narrow areas. This suggests that some more knowledge should be added to the algorithm in order to acquire a more generic behaviour, as that shown in Bui's results [29]. Nevertheless this could imply to increase the computing time and the results should be traded with the improvement in the number of solutions per execution time. These tradeoff problems constitute the core for our future work.

## 7 Conclusions

The use of the parallel processing in DMO problems gives a twofold improvement. On one hand, it allows a reduction in the execution time required to reach a good approximation to the new Pareto fronts, thus widening the field of the problems that can be tackled (problems with faster rate of change in the Pareto front). On the other hand, each worker can run more iterations in the same amount of time, thus it would it possible to increase the explored search space and to make it easy the adaptation to changes.

This chapter, besides providing a summary about the different approaches to parallelize multi-objective algorithms, describes two frameworks, pdMOEA and pdMOEA+, for analyzing different parallel approaches for multi-objective problems with time constraints to find the corresponding solutions.

Results with pdMOEA have shown that some MOEAs obtain almost super-linear speedups, while different MOEAs provide different speedup figures and solutions per time unit, throughput. As it has been said multiple times along this chapter, these two features are desired when dealing with real-world dynamic optimization problems because the practitioner and the researcher are both interested in getting the best possible solutions within the time constraints. SFGA and SFGA2 have proven to be able to accomplish these objectives, whilst other MOEAs such as SPEA2 and NSGA-II require considerable more time to improve slightly the quality of the solutions obtained by SFGA and SFGA2. The most salient feature of pdMOEA when used with SFGA2 is that gives very fast results with a quality not much worse than the obtained quality with SPEA2 or NSGA-II, and in addition, it produces many solutions per time unit, allowing to solve a DMO problem with a very good representation of the Pareto front. Specifically, pdMOEA has shown super-linear speedup when used with SPEA2 and NSGA-II, and a smaller speedup for SFGA and SFGA2.

In addition, the quality of the solutions has improved when more than one process were employed to solve the problems. Finally, all the algorithms have shown increases in the number of solutions per time unit or throughput as the number of processes increases.

In addition, pdMOEA+, a framework aimed at fully distributed computation, has been proposed and tested in this chapter. The first results with this new approach have shown a promising future while it has also shown that more development should be done in order to achieve better results that can rival with those provided by the pdMOEA approach or sequential MOEAs. Concretely, pdMOEA+ has shown faster execution than pdMOEA. Moreover, it offers linear speedup and produces a good number of solutions per time unit. Future research for pdMOEA+ should look for a way to keep, and even to improve, its execution time while sacrificing neither the quality nor the number of the solutions.

This way, the results for the speedup and performance are satisfactory. It has been shown a reduction in the convergence times (speedups), and hence, the ability to adapt to stronger time restrictions in the dynamic problem. We think that the super-linear speedups that have been observed clearly show the usefulness of parallel processing in keeping the diversity of the populations and in the algorithm adaptability. Also, it has been shown that while our SFGA and SFGA2 did not achieve super-linear speedups, they showed an improvement in the data throughput (non-dominated solutions per time unit).

**Acknowledgements.** The work described in this chapter has been funded by project SAF2010-20558 (Ministerio de Educación, Spain)

## References

1. Gupta, A., Sivakumar, A.: Pareto control in multi-objective dynamic scheduling of a stepper machine in semiconductor wafer fabrication. In: Proc. of the 2006 Winter Simulation Conference, pp. 1749–1756. IEEE Computer Science, Los Alamitos (2006)
2. Branke, J., Mattfeld, D.C.: Anticipation and flexibility in dynamic scheduling. Int. Journal of Production Research 43, 3103–3129 (2005)
3. Pillai, P., Shin, K.G.: Real-time dynamic voltage scaling for low-power embedded operating systems. SIGOPS Oper. Syst. Rev. 35(5), 89–102 (2001)
4. Lee, L.H., Teng, S.E., Chew, P., Karimi, I.A., Lye, K.W., Lendermann, P., Chen, Y., Koh, C.H.: Application of multi-objective simulation-optimization techniques to inventory management problems. In: Proc. of the 37th Conference on Winter Simulation, pp. 1684–1691 (2005)
5. Cheng, L., Subrahmanian, E., Westerberg, A.: Multi-objective decisions on capacity planning and production-inventory control under uncertainty. Industrial & Engineering Chemistry Research 43(9), 2192–2208 (2004)
6. Coello, C.A.: An updated survey of GA-based multiobjective optimization techniques. ACM Comput. Surv. 32(2), 109–143 (2000)
7. Farina, M., Deb, K., Amato, P.: Dynamic multiobjective optimization problems: Test cases, approximations, and applications. IEEE Trans. on Evol. Comp. 8, 425–442 (2004)

8. Jin, Y., Branke, J.: Evolutionary optimization in uncertain environments-a survey. IEEE Trans. on Evol. Comp. 9(3), 303–317 (2005)
9. Coello, C.A., Lamont, G.B., Van Veldhuizen, D.A.: Evolutionary Algorithms for Solving Multi-Objective Problems, 2nd edn. Genetic and Evolutionary Computation. Springer, Heidelberg (2007)
10. Cobb, H.G., Grefenstette, J.J.: Genetic algorithms for tracking changing environments. In: Proc. of the 5th International Conference on Genetic Algorithms, pp. 523–530. Morgan Kaufmann Publishers Inc., San Francisco (1993)
11. Vavak, F., Jukes, K., Fogarty, T.C.: Adaptive combustion balancing in multiple burner boiler using a genetic algorithm with variable range of local search'. In: Bäck, T. (ed.) ICGA, pp. 719–726. Morgan Kaufmann (1997)
12. Mori, N., Kita, H., Nishikawa, Y.: Adaptation to a Changing Environment by Means of the Feedback Thermodynamical Genetic Algorithm. In: Eiben, A.E., Bäck, T., Schoenauer, M., Schwefel, H.-P. (eds.) PPSN 1998. LNCS, vol. 1498, pp. 149–158. Springer, Heidelberg (1998)
13. Cedeno, W., Vemuri, V.R.: On the use of niching for dynamic landscapes. In: Proc. of the IEEE Int. Conf. on Evolutionary Computation, pp. 361–366 (1997)
14. Yang, S.: Population-based incremental learning with memory scheme for changing environments. In: Proc. of the 2005 Conference on Genetic and Evolutionary Computation, NY, USA, pp. 711–718 (2005)
15. Branke, J.: Memory enhanced evolutionary algorithms for changing optimization problems. In: Proc. of the IEEE Congress on Evolutionary Computation, pp. 1875–1882 (1999)
16. Branke, J., Kauler, T., Schmidt, C., Schmeck, H.: A multi-population approach to dynamic optimization problems. In: Adaptive Computing in Design and Manufacturing, pp. 299–308. Springer, Heidelberg (2000)
17. Ursem, R.K.: Multinational GAs: Multimodal optimization techniques in dynamic environments. In: Whitley, D., Goldberg, D., Cantu-Paz, E., Spector, L., Parmee, I., Beyer, H.G. (eds.) Proc. of the Genetic and Evolutionary Computation Conference, pp. 19–26. Morgan Kaufmann, Las Vegas (2000)
18. Talbi, E.G., Mostaghim, S., Okabe, T., Ishibuchi, H., Rudolph, G., Coello, C.A.: Parallel Approaches for Multiobjective Optimization. In: Branke, J., Deb, K., Miettinen, K., Słowiński, R. (eds.) Multiobjective Optimization. LNCS, vol. 5252, pp. 349–372. Springer, Heidelberg (2008)
19. Van Veldhuizen, D.A., Zydallis, J.B., Lamont, G.B.: Considerations in engineering parallel multiobjective evolutionary algorithms. IEEE Trans. on Evol. Comp. 7, 144–173 (2003)
20. Luna, F., Nebro, A.J., Alba, E.: Parallel evolutionary multiobjective optimization. In: Nedjah, N., De Mourelle, L., Alba, E. (eds.) Parallel Evolutionary Computations, pp. 33–56. Springer, Heidelberg (2006)
21. Knowles, J., Corne, D.: The pareto archived evolution strategy: a new baseline algorithm for pareto multiobjective optimization. In: Proc. of the Congress on Evolutionary Computation, pp. 98–105 (1999)
22. De Toro, F., Ortega, J., Ros, E., Mota, S., Paechter, B., Martín, J.M.: PSFGA: Parallel processing and evolutionary computation for multiobjective optimization. Parallel Computing 30, 721–739 (2004)
23. Horn, J., Nafpliotis, N.: Multiobjective Optimization using the Niched Pareto Genetic Algorithm. Technical Report IlliGAl Report 93005, Urbana, Illinois, USA (1993)
24. Alba, E.: Parallel evolutionary algorithms can achieve super-linear performance. Inf. Process. Lett. 82(1), 7–13 (2002)

25. Deb, K., Zope, P., Jain, A.: Distributed Computing of Pareto-Optimal Solutions with Evolutionary Algorithms. In: Fonseca, C.M., Fleming, P.J., Zitzler, E., Deb, K., Thiele, L. (eds.) EMO 2003. LNCS, vol. 2632, pp. 534–549. Springer, Heidelberg (2003)
26. Branke, J., Schmeck, H., Deb, K., Maheshwar, R.S.: Parallelizing multiobjective evolutionary algorithms: cone separation. In: Proc. of the Congress on Evolutionary Computation, pp. 1952–1957 (2004)
27. Hiroyasu, T., Miki, M., Watanabe, S.: The new model of parallel genetic algorithm in multiobjective optimization problems – divided range multiobjective genetic algorithm. In: Proc. of the Congress on Evolutionary Computation, pp. 333–340 (2000)
28. Bui, L.T., Abbass, H.A., Essam, D.: Local models – an approach to distributed multiobjective optimization. Comput. Optim. Appl. 42, 105–139 (2009)
29. Streichert, F., Ulmer, H., Zell, A.: Parallelization of Multi-Objective Evolutionary Algorithms Using Clustering Algorithms. In: Coello Coello, C.A., Hernández Aguirre, A., Zitzler, E. (eds.) EMO 2005. LNCS, vol. 3410, pp. 92–107. Springer, Heidelberg (2005)
30. Navarro, A., Allen, C.: Adaptive classifier based on K-means clustering in dynamic programming. Document Recognition IV 36, 31–38 (1997)
31. Kövesi, B., Boucher, J.M., Saoudi, S.: Stochastic K-means algorithm for vector quantization. Pattern Recog. Lett. 22(6-7), 603–610 (2001)
32. Zitzler, E., Laummanns, M., Thielem, L.: SPEA2: improving the Strength Pareto Evolutionary Algorithm for multi-objective optimization. In: Giannakoglou, K., et al. (eds.) Evolutionary Methods for Design, Optimization and Control to Industrial Problems, pp. 95–100. Center for Numerical Methods in Engineering (2002)
33. Deb, K., Agrawal, S., Pratab, A., Meyarivan, T.: A fast elitist Non-dominated Sorting Genetic Algorithms for multi-objective optimisation: NSGA-II. In: Deb, K., Rudolph, G., Lutton, E., Merelo, J.J., Schoenauer, M., Schwefel, H.-P., Yao, X. (eds.) PPSN 2000. LNCS, vol. 1917, pp. 849–858. Springer, Heidelberg (2000)
34. Deb, K., Agrawal, R.B.: Simulated binary crossover for continuous search space. Complex Systems 9, 115–148 (1995)
35. Deb, K.: Multi-Objective Optimization using Evolutionary Algorithms. John Wiley & Sons Inc., New York (2001)
36. Cámara, M., Ortega, J., de Toro, F.: Approaching Dynamic Multi-Objective Optimization Problems by Using Parallel Evolutionary Algorithms. In: Coello Coello, C.A., Dhaenens, C., Jourdan, L. (eds.) Advances in Multi-Objective Nature Inspired Computing. Studies in Computational Intelligence, vol. 272, pp. 63–86. Springer, Heidelberg (2010)

# An Empirical Study of Parallel and Distributed Particle Swarm Optimization

Leonardo Vanneschi, Daniele Codecasa, and Giancarlo Mauri

**Abstract.** Given the implicitly parallel nature of population-based heuristics, many contributions reporting on parallel and distributed models and implementations of these heuristics have appeared so far. They range from the most natural and simple ones, i.e. fitness-level embarrassingly parallel implementations (where, for instance, each candidate solution is treated as an independent agent and evaluated on a dedicated processor), to many more sophisticated variously interacting multi-population systems. In the last few years, researchers have dedicated a growing attention to Particle Swarm Optimization (PSO), a bio-inspired population based heuristic inspired by the behavior of flocks of birds and shoals of fish, given its extremely simple implementation and its high intrinsical parallelism. Several parallel and distributed models of PSO have been recently defined, showing interesting performances both on benchmarks and real-life applications. In this chapter we report on four parallel and distributed PSO methods that have recently been proposed. They consist in a genetic algorithm whose individuals are co-evolving swarms, an "island model"-based multi-swarm system, where swarms are independent and interact by means of particle migrations at regular time steps, and their respective variants enriched by adding a repulsive component to the particles. We show that the proposed repulsive multi-swarm system has a better optimization ability than all the other presented methods on a set of hand-tailored benchmarks and complex real-life applications.

## 1 Introduction

Swarm systems [6, 16] are computational methods introduced to solve difficult problems or model complex phenomena, inspired by natural collective phenomena like the behavior of colonies of social insects such as termites, bees, and ants. One of the

Leonardo Vanneschi · Daniele Codecasa · Giancarlo Mauri
Department of Informatics, Systems and Communication (D.I.S.Co.)
University of Milano-Bicocca
Milan, Italy
e-mail: {vanneschi,codecasa,mauri}@disco.unimib.it

F.F. de Vega et al. (Eds.): Parallel Architectures & Bioinspired Algorithms, SCI 415, pp. 125–150.
springerlink.com

simplest and most often studied and used techniques of this type is Particle Swarm Optimization (PSO) [13, 27, 8]. In PSO, which is inspired by flocks of birds and shoals of fish, candidate solutions to the problem at hand, that are often vectors of real numbers or other continuous values, also called *particles*, are seen as moving points in a Cartesian space. In the canonical form, a set of particles, called *swarm*, is kept in memory. The particles in the swarm are iteratively shifted in the space of solution by means of simple update rules. In particular, at each iteration, each particle evaluates the fitness at its current location; then it determines its movement through the solution space by combining some aspect of the history of its own fitness values with those of one or more other members of the swarm. The members of the swarm that a particle can interact with are called its *social neighborhood*. More precisely, in the canonical version of PSO, the movement of each particle depends on two elastic forces, one attracting it with random magnitude to the fittest location so far encountered by the particle, and one attracting it with random magnitude to the best location encountered by any of the particles social neighbors in the swarm.

Parallel and distributed approaches are natural in swarm intelligence and they have been used intensively since the early years of this research field [21, 15]. Swarm systems, in fact, have often been described as *intrinsically parallel* computational methods. The reason for this is that many of the main computational tasks characterizing this family of heuristics are independent of each other; thus it is straightforward to perform them at the same time. This is the case, for instance, of the evaluation of the fitness of the particles in a swarm. Furthermore, by attributing a non-panmictic structure to the population, something that also finds its inspiration in nature, the operations that allow particles to update their position can also be performed independently of each other (once the coordinates of the swarm global best position so far have been distributed among the processors) and thus can also potentially be parallelized. These approaches can be useful even when there is no actual parallel or distributed implementation, thanks to the particular information diffusion given by the more local structures of the swarms. But of course parallel and distributed approaches are at their best when the structures of the models are reflected in the actual algorithm implementations. In fact, when compared with other heuristics, swarm systems are relatively costly and slow. But parallel and distributed implementations can boost performance and thereby allow practitioners to solve, exactly or approximately, larger and more interesting problem instances thanks to the time savings afforded.

These advantages have been known and appreciated since several years and some of the previous efforts are summarized in Section 2. Motivated by the need of developing high quality work on parallel and distributed approaches in PSO, in [32], [33] and [34] four new highly parallelizable PSO variants were introduced. This chapter contains a systematic presentation of the models we defined in [32], [33] and [34] and of the results of a set of experiments carried on to test the respective optimization abilities between each other and with standard PSO.

One of the most delicate aspects of experimental studies aimed at comparing different optimization methods is to find a suitable set of test problems on which to perform the simulations. In this chapter we use a large set of problems composed by:

- two sets of test functions (that we call *cosff* and *wtrap* in the continuation) whose difficulty can be tuned by simply modifying the values of some parameters and that were introduced for the first time in [32],
- a new set of functions that integrates *cosff* and *wtrap* (we call it *CosTrapFF* from now on). This set of functions is defined and studied for the first time here.
- the well known set of Rastrigin test functions (see for instance [38]),
- four complex real-life problems in the field of drug discovery, whose objective is the prediction of as many important pharmacokinetic parameters (Human Oral Bioavailability, Median Oral Lethal Dose, Plasma Protein Binding levels and Docking Energy) as a function of a set of molecular descriptors of potential candidate new drugs (the first three of these problems were introduced in [3] and the fourth one in [2]),
- one further complex real-life problem in the field of drug discovery, introduced in [1] and whose objective is the prediction of the response of cancer patients to a pharmacologic therapy based on a drug called Fludarabine.

This chapter is structured as follows: Section 2 contains a discussion of previous and related works, focusing on multi-swarm PSO, on models that integrate Evolutionary Algorithms (EAs) with PSO and on attractive/repulsive PSO methods defined so far. In Section 3 the new PSO methods we are proposing are discussed. In Section 4 we describe the sets of test problems used. In Section 5 we present and discuss the obtained experimental results. Finally, Section 6 concludes the chapter.

## 2 Previous and Related Work

In [20, 12], the existing PSO publications have been classified into two broad areas: modifications/improvements to the canonical PSO algorithm (presented in Section 3.1) and applications. The present chapter belongs to the first one of these two classes, even though a set of real-life applications is used as a test case to validate the proposed models. For this reason, only some of previous modifications/improvements to the canonical PSO algorithm will be discussed here, while we refer to [20] for a survey of the main applications.

Several different variants of the standard PSO algorithm have been proposed so far. For instance, methods to optimize parameters such as the inertia weight and the constriction and acceleration coefficients have been proposed [7, 4, 30, 36]. Even though interesting, these contributions are orthogonal to the present work, where one particular, so to say, ”standard” parameter setting is considered, and they could be integrated with the present work in the future. Another variant of PSO, whose popularity is steadily increasing, consists in establishing a “structure” (or “topology”) to the swarm. Among others, Kennedy and coworkers evaluate different kinds of topologies, demonstrating the suitability of random and Von Neumann neighborhoods [14] for a wide set of benchmarks, even though the authors themselves also remark that selecting the most efficient neighborhood structure is in general a hard and problem-dependent task. In [9], Oltean and coworkers evolve the structure of an asynchronous version of PSO. They use a hybrid technique that combines PSO with a genetic algorithm (GA), in which each GA chromosome is defined as an array which encodes an update strategy for the particles. This contribution represents one of the first efforts to integrate GAs and PSO, which is tackled also in the present chapter by the definition of the PSE algorithm (see Section 3.2). The authors of [9] empirically show that PSO with the evolved update strategy performs similarly and sometimes even better than standard approaches for several benchmark problems. After [9], many other improvements based on the integration of Evolutionary Algorithms (EAs) and PSO have been proposed. For instance, a modified genetic PSO has been defined by Jian and colleagues [39], which takes advantage of the crossover and mutation operators, along with a differential evolution (DE) algorithm which enhances search performance, to solve constrained optimization problems. Other work aimed at solving global non-linear optimization problems is presented by Kou and colleagues in [37]. They developed a constraint-handling method in which a double PSO is used, together with an induction-enhanced evolutionary strategy technique. Two populations preserve the particles of the feasible and infeasible regions, respectively. A simple diversity mechanism is added, allowing the particles with good properties in the infeasible region to be selected for the population that preserves the particles in the feasible region. The authors state that this technique could effectively improve the convergence speed with respect to plain PSO.

As explained in Section 3, the PSE algorithm introduced in this work substantially differs from the variants presented in [39, 9, 28, 37] mainly because it consists in a GA, where each evolving individual is a swarm. This idea is new and, to the best of our knowledge, it has never been exploited before. The interested reader is referred for instance to [21] for a detailed survey on the different variants of the standard PSO algorithm that have been proposed so far.

In this chapter we study multi-swarm PSO and attractive/repulsive PSO methods. In [5] a multi-swarm PSO method for dynamic optimization environments was proposed. The main idea was to extend the single swarm PSO, integrating it with another model called Charged Particle Swarm Optimization, that integrates interacting multi-swarms. The main difference between contribution [5] and the present one is that in [5] the goal is clearly the one of improving the PSO optimization ability and self-adaptability in presence of dynamic environments, i.e. where the target function changes with time according to some unknown patterns. The overhead of computation introduced by [5] for dynamic adaption clearly slows down the performance in presence of static problems. Here we adopt a different perspective: our goal is to improve the PSO optimization ability in case of complex (i.e. characterized by difficult – rugged or deceptive – fitness landscapes) but static (i.e. where the target is fixed and does not change with time) problems.

In [18] an interesting extension of multi-swarm PSO was proposed, where an independent local optimization is performed in all the different swarms. When these local optimization processes terminate, all the particles in the system are once again randomly partitioned in several different swarms and the process is iterated. Even though interesting, the model presented in [18] differs from the models presented here since we do not introduce any local optimization strategy in the algorithm.

In fact, we want to test our models on difficult problem spaces, possibly characterized by the presence of many local optima, and we believe that adding local optimization strategies would favor premature convergence or stagnation. A variant of the algorithm proposed in [18] was presented in [17], where a mechanism to track multiple peaks by preventing overcrowding at a peak is introduced. Instead of introducing local optimization and then defining criteria for avoiding premature convergence, we prefer not to use local optimization here, and to tackle complex problems using different ideas, like isolating interacting sub-swarms and repulsion.

In [11] PSO was modified to create the so called Multi-Swarm Accelerating PSO, which is applied to dynamic continuous functions. In contrast to the previously introduced multi-swarm PSOs and local versions of PSO, the swarms are this time dynamic. The whole population is divided into many small swarms, which are regrouped frequently by using various regrouping schedules, and exchange information among them. Accelerating operators are combined to improve its local search ability. As previously pointed out, also in the case of [11] the focus is on dynamic environments, while here we concentrate on static, yet complex, problems.

In [19] a multi-swarm PSO method inspired by symbiosis in natural ecosystems is presented. This method is based on a master-slave model, in which a population consists of one master swarm and several slave swarms. The slave swarms execute a single PSO or its variants independently to maintain the diversity of particles, while the master swarm evolves based on its own knowledge and also the knowledge of the slave swarms. Paper [19] represents one of our main sources of inspiration. Nevertheless, it differs from the present work by the fact that we do not partition the swarms into masters and slaves here, but we assign to all of them the same hierarchic importance. This has interesting effects on the allocation of our swarm/processes on computational resources: we do not have to identify the most powerful resources and assign them to master swarms. In other words, we do not need complex allocation algorithms: our system can work on clusters of machines that have all the same computational power and the allocation can be made arbitrarily.

A domain in which multi-swarm PSO methods find a natural application is multi-objective optimization, in which typically each swarm is used to optimize a different criterion. It is the case, among many other references, of [35]. This contribution is interesting and has to be considered for a future extension of our approach. But for the moment, we restrict our work to problems that have only one target/objective.

The concept of repulsive PSO or attractive/repulsive PSO has been exploited in some references so far. It is the case, for instance, of [23], where a PSO variant is presented where the attractive or repulsive behavior of the particles is changed in function of the swarm's diversity. Even though interesting, the algorithm proposed in [23] has the drawback of calculating diversity at each iteration of the algorithm, to decide if particles have to be attractive or repulsive. This implies an overhead of computation, that we want to avoid. In fact, in our models we identify some particles as attractive and some others as repulsive once for all, on the basis of precise principles (that will be discussed in Section 3), and we do not change their characteristics during evolution.

# 3 PSO Algorithms

In this section we give the definitions of the basic PSO algorithms and of the four algorithms that will be compared.

## 3.1 Algorithm 1: PSO

This algorithm is the canonic PSO [27], where each particle is attracted by one global best position for all the swarm and one local best position. Velocity and position-update equations are as follows:

$$\mathbf{V}(t) = w * \mathbf{V}(t-1) \ + C_1 * rand() * [\mathbf{X}_{best}(t-1) - \mathbf{X}(t-1)] +$$

$$C_2 * rand() * [\mathbf{X}_{gbest}(t-1) - \mathbf{X}(t-1)] \qquad (1)$$

$$\mathbf{X}(t) = \mathbf{X}(t-1) + \mathbf{V}(t)$$

where $\mathbf{V}(t)$ and $\mathbf{X}(t)$ are the velocity and the position of the particle at time $t$, $C_1, C_2$ are two positive constants that set the relative importance of the social and individual attraction components, $w$ is the inertia weight (constriction factor), $\mathbf{X}_{best}(t-1)$ is the position with the best-fitness among the ones reached by the particle up to time $t-1$ and $\mathbf{X}_{gbest}(t-1)$ is the best-fitness point ever found by all the particles in the whole swarm. When a particle reaches a border of the admissible range on one dimension, velocity on that dimension is set to zero.

In our experiments, we have considered a swarm size equal to $100$ particles. We have used this value because we have to compare the results of standard PSO with the ones of various versions of distributed and multi-swarm PSO and we want the total number of particles in each system to be the same (for instance, in the PSE algorithm described below, we will use $10$ swarms of $10$ particles each). The other parameters we have used are: $C_1 = C_2 = 2$. The value of $w$ has been progressively decremented during the execution of the algorithm from the initial value of $1$ to the final value of $0.001$. Maximum particle velocity has been set to $0.5$ for the *cosff*, the *wtrap*, the *CosTrapFF* and the *Rastriging* test functions, while for the real-life applications we have used $1/3$ of its size. The motivation for this choice is that while the *cosff*, *wtrap* and *CosTrapFF* functions are limited to the $[0, 1]$ range and the *Rastriging* function to the $[-5.12, +5.12]$ range, the real-life applications generally work on much larger ranges. The maximum number of fitness evaluations depends on the test function and on the used parameters. For the *cosff*, the *wtrap* and the *CosTrapFF* functions it varies between $10^5$, $2 \times 10^5$ and $3 \times 10^5$. For the *Rastriging* function and for the real-life applications it varies between $2 \times 10^5$ and $3 \times 10^5$ (for much precision, the reader can refer to the horizontal axis of the average best fitness plots in Figures 3, 4, 5, 6, 7 and 8). Section 4 introduces these test problems.

### 3.2 Algorithm 2: Particle Swarm Evolver (PSE)

This algorithm functions as a GA in which each individual in the population is an evolving swarm. In synthesis, all the swarms in a population work independently, applying iteratively the PSO algorithm described above for a certain number of steps (say $p$ which stands for $p$eriod). After that, we execute one generation of the GA by selecting the most promising swarms for mating and by evolving them with some particular crossover and mutation operators. Each swarm has exactly the same number of particles and this number remains constant during the whole evolution. The fitness of an individual/swarm is defined as the fitness of its global best particle. In this first simple implementation, the crossover between two parent swarms is just a random mixing of their particles, in such a way that each offspring swarm contains the same number of particles as the parent swarms but some of them (at random) belong to one parent and the rest to the other parent. The mutation of a swarm is the replacement of a random particle in that swarm (chosen with uniform probability) with another random particle (we point out that this process do *not* alter the global best of the swarm unless the new randomly created particle is itself the new global best).

In our experiments, we have used the following parameters: number of independent iterations of each swarm in the population before a GA generation: $p = 10$. Number of particles in each swarm: $10$. Number of swarms in the GA population: $10$. Crossover probability between swarms: $0.95$. Probability of mutation of a swarm: $0.01$. Selection of the most promising swarms have been obtained by tournament selection with tournament size equal to 2. All the other parameters are like in the PSO algorithm described in the first part of this section.

### 3.3 Algorithm 3: Repulsive PSE (RPSE)

This algorithm works as PSE defined above, except that each swarm also has a *repulsive* component: each particle of each swarm is *attracted* by the global best of its own swarm and by its local best position and *repulsed* by the global best of *all* the other swarms in the GA population (this is true only in the case the global best of the other swarm is different from the global best of the swarm that particle belongs to). For each swarm different from the current one, the repulsive factor of each particle is given by:

$$\mathbf{V}(t) = \mathbf{V_{PSO}}(t) + C_3 * rand() * f(\mathbf{X}_{foreign-gbest}(t-1), \mathbf{X}(t-1), \mathbf{X}_{gbest}(t-1)) \quad (2)$$

$V_{PSO}(t)$ is the velocity calculated with the standard PSO update rule (see equation (1)), $\mathbf{X}_{gbest}(t-1)$ is the position of the global best of the current swarm and $\mathbf{X}_{foreign-gbest}(t-1)$ is the position of the global best of the other considered swarm. The function $f$ always ignores $\mathbf{X}_{gbest}(t-1)$ and repulses the particle by pushing it in the opposite direction of $\mathbf{X}_{foreign-gbest}(t-1)$ except in case the repulsor is between the particle and the global best of the current swarm. In the latter case, $f$ further accelerates the particle towards $\mathbf{X}_{gbest}(t-1)$. The functioning of $f$ is described by the following pseudo-code:

```
if $(\mathbf{X}(t-1) < \mathbf{X}_{foreign-gbest}(t-1))$ and $(\mathbf{X}_{foreign-gbest}(t-1) < \mathbf{X}_{gbest}(t-1))$
    or $(\mathbf{X}_{gbest}(t-1) < \mathbf{X}_{foreign-gbest}(t-1) < \mathbf{X}(t-1))$
        return $-\phi(\mathbf{X}(t-1), \mathbf{X}_{foreign-gbest}(t-1))$
else
        return $\phi(\mathbf{X}(t-1), \mathbf{X}_{foreign-gbest}(t-1))$
```

where:

$$\phi(\mathbf{X}(t-1), \mathbf{X}_{foreign-gbest}(t-1)) = \\ sig(dis) * (1 - |dis/(U-L)|)$$

and where: $dis = \mathbf{X}(t-1) - \mathbf{X}_{foreign-gbest}(t-1)$, $sig$ indicates the sign function and $U$ and $L$ are the upper and lower bound of the interval respectively.

Informally, $\phi$ is equal to the sign of the difference of its arguments multiplied by 1 minus the normalized distance of its arguments. In other words the repulsive factor increases with the proximity to the repulsor and the obtained value does not depend on the space dimension.

In our experiments, we have used $C_3 = 2/N$, where $N$ is the number of repulsors, i.e. the number of swarms minus 1. All the other parameters we have used used had the same values as in the PSE algorithm described above.

### *3.4 Algorithm 4: Multi-swarm PSO (MPSO)*

As for the previous methods, also MPSO uses a set of swarms that run the standard PSO algorithm independently for a given number of iterations and then synchronize and have an interaction. The interaction, this time, simply consists in the exchange of some particles. In particular, as it is often the case in island EAs models [10], the set of the $k$ best particles in the sender swarm is copied into the receiver swarm. The new particles replace the worst $k$ ones in the receiver swarm, while a copy of them also remains in the sender swarm (the process is a copy instead of a migration).

The number of independent PSO iterations in each swarm before communication has been set to 10 (as in PSE). The number $k$ of migrating particles has been set to $1/5$ of the number of particles in each swarm (all the swarms have exactly the same number of particles). The swarms communicate using a *ring* topology (see for instance [10]). As for the PSE algorithm, we have used 10 swarms of 10 particles each. All the other parameters we have used are as in the PSO algorithm described above.

### *3.5 Algorithm 5: Multi-swarm Repulsive PSO (MRPSO)*

This algorithm works as MPSO defined above, except that the particles in the swarms with an *even* index in the ring topology (i.e. only a half of the swarms) also have a repulsive component. Each particle of those swarms is attracted by the global best of its own swarm and by its local best position so far and it is repulsed by the global best of the previous swarm in the ring, i.e. the swarm from which it receives individuals at migration time (given that this swarm will be in an odd position in the ring topology, its particles will not have a repulsive component). In this way, the particles that migrate in the even swarms should be as different as possible from the particles already contained in those swarms (given that they have been repulsed by the global best of the sender swarm until the previous generation). This should help maintaining diversity in the whole system. The repulsive component of each particle is exactly the same as for the RPSE algorithm described above, except that this time we have used $C_3 = 0.5$ because we have only *one* repulsor for each particle. All the other parameters we have used are the same as for the MPSO algorithm described above.

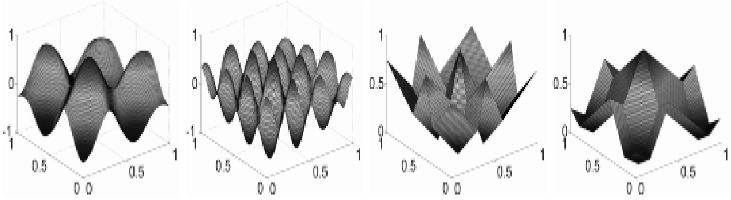
## 4 Test Functions

### *4.1 Cosff Functions*

The first set of test functions we propose in this work is defined as:

$$\mathit{cosff}(\mathbf{x}) = (\sum_{i=1}^{n} f_i(x_i, M_i))/n \tag{3}$$

where $n$ is the number of dimensions of the problem, $\mathbf{x} = (x_1, x_2, ..., x_n)$ is a point in an $n$-dimensional space and for all $i = 1, 2, ..., n$ given two floating point numbers $x$ and $M$:



(a) (b) (c) (d)

**Fig. 1** Two dimensional graphical representations of two *cosff* (plots (a) and (b)) and two *wtrap* (plots (c) and (d)) functions. Plot (a) shows the *cosff* function with $K = 10$, $M_1 = M_2 = 0.3$. Plot (b) shows the *cosff* function with $K = 20$, $M_1 = M_2 = 0.3$. Plot (c) shows the *wtrap* function with $B = 0.3$ and $R = 0.75$ and plot (d) shows the *wtrap* function with $B = 0.7$ and $R = 0.25$. See the text for an explanation of the $K$, $M_1$ and $M_2$ parameters of the *cosff* functions and an explanation of the $B$ and $R$ parameters of the *wtrap* functions.

$$f_i(x, M) = \begin{cases} \cos(K * (x - M)) * (1.0 - (M - x)), & \text{if } x \leq M \\ \cos(K * (x - M)) * (1.0 - (x - M))), & \text{otherwise} \end{cases}$$

and where $(M_1, M_2, ..., M_n)$ are the coordinates of the known maximum value of the function and $K$ is a constant that modifies the ruggedness of the fitness landscape (the higher $K$ the most complex the fitness landscape).

The two dimensional graphical representations of function *cosff* with $K = 10$ and $K = 20$ are reported in Figures 1(a) and 1(b) respectively. In both plots, we have used $(M_1, M_2) = (0.3, 0.3)$ as the coordinates of the global maximum. From those plots it is clear that increasing the value of $K$ we are able to increase the ruggedness of the fitness landscape and thus its complexity.

### *4.2 WTrap Functions*

The second set of test functions we use in this work is called *W-trap* functions (see for instance [31] for a preliminary and slightly different definition of these functions). It is defined as follows:

$$wtrap(\mathbf{x}) = (\sum_{i=1}^{n} g(x_i))/n \tag{4}$$

where $n$ is the number of dimensions and, given a floating point number $x$:

$$g(x) = \begin{cases} R_1 * (B_1 - x)/B_1, & \text{if } x \leq B_1 \\ R_2 * (x - B_1)/(B_2 - B_1), & \text{if } B_1 < x \leq B_2 \\ R_2 * (B_3 - x)/(B_3 - B_2), & \text{if } B_2 < x \leq B_3 \\ R_3 * (x - B_3)/(1 - B_3) & \text{otherwise} \end{cases}$$

and where $B_1 < B_2 < B_3$ and $B_1$, $B_3$ are the coordinates of the two minima in the search space while the global maximum has coordinates in $B_2$[1]. $R_1$ is the fitness of the first local maximum placed in the origin (all coordinates equal to 0.0), $R_2$ is the fitness of the global maximum and $R_3$ is the fitness of the second local maximum, that has all its coordinates equal to 1.0.

In this chapter, for simplicity, we wanted to modify the functions difficulty by changing the values of only two parameters, instead of the 6 typical parameters of *wtrap* functions. For this reason, given two parameters $B$ and $R$, we have used: $B_1 = 0.4 - B/3$, $B_2 = 0.4$, $B_3 = 0.4 + B * 2/3$, $R_1 = R$, $R_2 = 1.0$ and $R_3 = R$ and we have obtained different test problems by modifying $B$ and $R$.

[1] I.e. the first minimum has coordinates $(B_1, B_1, ..., B_1)$, the global maximum has coordinates $(B_2, B_2, ..., B_2)$ and the second minimum has coordinates $(B_3, B_3, ..., B_3)$. In other words the first minimum has all its $n$ coordinates equal to $B_1$, the global maximum has all its $n$ coordinates equal to $B_2$ and the second minimum has all its $n$ coordinates equal to $B_3$.

The two dimensional graphical representations of functions $wtrap$ with $B = 0.3$ and $R = 0.75$ and with $B = 0.7$ and $R = 0.25$ are reported in Figures 1(c) and 1(d) respectively. We can see that changing the values of $B$ and $R$ we can modify the relative importance of the basins of attraction of global and local maxima, thus tuning the difficulty of the problem.

## 4.3 CosTrapFF Functions

These functions are defined in terms of the *cosff* and *wtrap* previously introduced. In particular, if $x < B_1$ or $x > B_3$ then we use the *wtrap*, otherwise we use the *cosff* with $M = B_2$. In order to reduce the number of parameters, as for the *wtrap* functions, we have defined two values $B$ and $R$, where: $B1 = 0.4 - B/3$, $B2 = 0.4$, $B3 = 0.4 + B * 2/3$, $R1 = R2 = R$, while the parameters of the *cosff* have been are $M = B2$ and $K$. Experiments have been performed changing the values of the following parameters:

- the number of dimensions (10 and 20);
- $B = 0.10, 0.30, 0.50, 0.70$ and $0.90$;
- $R = 0.25, 0.50$ and $0.75$;
- $K = 10$ and $20$.

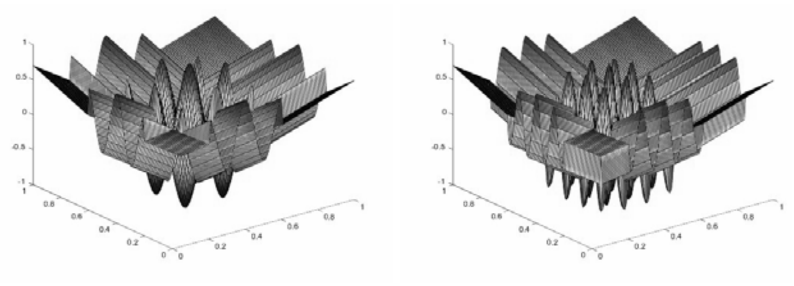
For the experiments on 10 dimensions we have used 100000 as the maximum number of fitness evaluations, while for the tests on 20 dimensions this limit was set to 200000 fitness evaluations. Figure 2 reports a two-dimensional graphical representation of two of these functions using the following parameters: $B_1 = 0.2$, $B_2 = 0.4$, $B_3 = 0.6$, $R_1 = 0.7$ and $R_2 = 0.7$. In Figure 2(a) we have set $K = 30$, while in Figure 2(b) we have $K = 50$.

## 4.4 Rastrigin Functions

The *Rastrigin* functions are a well known and widely used set of test functions for floating point parameters optimization, given its high multimodality, the regular distribution of the minima and the possibility of changing the ruggedness of the induced fitness landscape by modifying one single real-valued parameter. These functions are defined as follows:

$$Rastrigin(\mathbf{x}) = n \cdot A + \sum_{i=1}^{n} (x_i^2 - A \cdot cos(2\pi x_i)) \tag{5}$$

where for all $i = 1, 2, ..., n$, $x_i \in [-5.12, 5.12]$, $n$ is the dimension of the function and $A$ is the parameter that determines the steepness of the local optima and thus the complexity of the induced fitness landscape. For a deeper introduction to these functions the reader is referred to [38] and for their



(a)

(b)

**Fig. 2** Two dimensional graphical representations of two *CosTrapFF* functions. For both functions we have used $B_1 = 0.2$, $B_2 = 0.4$, $B_3 = 0.6$, $R_1 = 0.7$ and $R_2 = 0.7$. Plot (a) shows the *CosTrapFF* function with $K = 30$. Plot (b) shows the *CosTrapFF* function with $K = 50$.

graphical representations for various different values of the $A$ parameter to: http://www.cs.rtu.lv/dssg/en/staff/rastrigin/astr-function.html.

## *4.5 Real-Life Applications in Drug Discovery*

We also consider a set of real-life applications characterized by a large dimensionality of the feature space. Four of them consist in predicting the value of as many important pharmacokinetic parameters and the fifth consists in predicting the response of a set of cancer patients to the treatment of the Fludarabine drug. These problems are briefly discussed in the continuation of this section. The interested reader is referred to the contributions quoted below for a more detailed introduction.

**Prediction of pharmacokinetic parameters.** These problems consist in predicting the value of four pharmacokinetic parameters of a set of candidate drug compounds on the basis of their molecular structure. The first pharmacokinetic parameter we consider is human oral bioavailability (indicated with %F from now on), the second one is median oral lethal dose (indicated with LD50 from now on), also informally called toxicity, the third one is plasma protein building levels (indicated with %PPB from now on) and the fourth one is called Docking Energy (indicated with DOCK from now on). %F is the parameter that measures the percentage of the initial orally submitted drug dose that effectively reaches the systemic blood circulation after the passage from the liver. LD50 refers to the amount of compound required to kill 50% of the test organisms (cavies). %PPB corresponds to the percentage of the drug initial dose that reaches blood circulation and binds the proteins of plasma. DOCK quantifies the amount of target-drug chemical interaction, i.e. the energy that binds the molecules of the candidate drug to the ones of the target tissue. For a more detailed discussion of these four pharmacokinetic parameters, the reader is referred to [3, 2]. The datasets we have used are the same as in [3] and [2]: the %F (LD50, %PPB and DOCK respectively) dataset consists in a matrix composed by 260 (234, 234 and 150 respectively) rows (instances) and 242 (627, 627 and 268 respectively) columns (features). Each row is a vector of molecular descriptor values identifying a drug; each column represents a molecular descriptor, except the last one, that contains the known target values of %F (LD50, %PPB and DOCK respectively). These datasets can be downloaded from:

http://personal.disco.unimib.it/Vanneschi/bioavailability.txt,
http://personal.disco.unimib.it/Vanneschi/toxicity.txt,
http://personal.disco.unimib.it/Vanneschi/ppb.txt, and
http://personal.disco.unimib.it/Vanneschi/dock.txt.

For all these datasets training and test sets have been obtained by random splitting: at each different PSO run, 70% of the molecules have been randomly selected with uniform probability and inserted into the training set, while the remaining 30% formed the test set. These problems were solved by PSO by means of a linear regression, were the variables in a PSO candidate solutions represent the coefficients of the linear interpolating polynomial. We have imposed these coefficients to take values in the range $[-10, 10]$. As fitness, we have used the root mean squared error (RMSE) between outputs and targets. For more details on the solving process for the bioavailability dataset via a linear regression by means of PSO, the reader is referred to [7].

**Prediction of response to Fludarabine treatment.** This problem consists in predicting anticancer therapeutic response on the basis of the genetic signature of the patients. To build the data, we have used the NCI-60 microarray dataset [25, 26, 22], looking for a functional relationship between gene expressions and responses to the Fludarabine oncology drug. Fludarabine (indicated by FLU from now on) is a drug for the treatment of chronic lymphocytic leukemia. The dataset we have used can be represented by a matrix with 60 lines (instances) and 1376 columns (features). Each line represents a gene expression. Each column represents the expression level of one particular gene, except the last one that contains the known value of the therapeutic response to the chosen drug (Fludarabine). Thus, as for the previous problems, the last column of the matrix contains the known values of the parameter to estimate. The reader is referred to [22] for a more detailed documentation of this dataset. The dataset itself can be downloaded from the web page:

http://personal.disco.unimib.it/Vanneschi/gp_nci_datasets.htm.

## 5 Experimental Results

The results presented in this section have been obtained performing 200 independent runs of each considered PSO method for the *cosff*, *wtrap*, *CosTrapFF* and *Rastriging* functions and 25 independent runs for five real-life applications (because the fitness calculation for those problems is more time consuming than for the other ones). Furthermore, we have studied three different performance measures. These measures are three of the evaluation criteria requested in [29]: the number of successful runs, the success performance and the average best fitness.

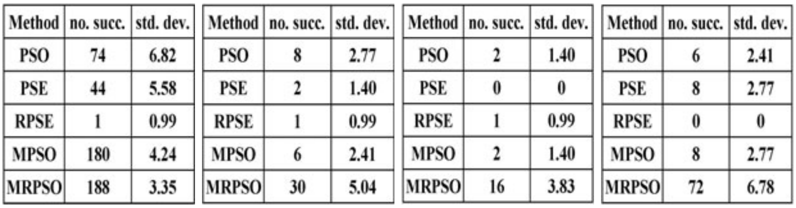
- The number of successful runs is defined as the number of runs in which an individual has been found that approximates the global optimum with an error smaller than a given threshold. For the *cosff*, *wtrap*, *CosTrapFF* and Rastrigin test functions the threshold we have used is equal to $10^{-8}$. For the real-life applications it is equal to $10^{-5}$.
- The success performance is defined as the mean of the fitness evaluations requested for successful runs, multiplied by the total number of runs, divided by the number of successful runs (this measure is introduced in [29], at page 41). By its definition, small values of the success performance are better than large ones, but this has an exception, given that we have forced the value of the success performance to be equal to zero when no run has been successful (thus, a success performance equal to zero is the worst possible value and not the best possible one).
- Finally, the average best fitness reports the average of the best fitness in the whole PSO system at each iteration, calculated on all the performed runs.

For the number of successful runs, we have calculated standard deviations (for verifying the statistical significance of the presented results) following [10], where experimental runs are considered as a series of independent Bernoulli trials having only two possible outcomes: success or failure. In this case, the number of successes (or of failures) is binomially distributed [24]. The maximum likelihood estimator $\hat{p}$ for the mean of a series of Bernoulli trials, and hence for the probability of success, is simply the number of successes divided by the sample size (the number of runs $n$). With this information at hand, one can calculate the sample standard deviation $\sigma = \sqrt{n.\hat{p}(1-\hat{p})}$. The experimental results that we have obtained for the different studied test problems are discussed below.

Figure 3 reports the results obtained for the *cosff* test function. The dimensionality of the problem (number of elements of the vector coding a particle) is equal to 20. The $K$ constant of the *cosff* function is equal to 10. The number of successful runs, with their standard deviations are reported in tabular form in figures 3(a), 3(b), 3(c) and 3(d). The success performance is reported as histograms in plots 3(e), 3(f), 3(g) and 3(h). The average best fitness curves against fitness evaluations are reported in plots 3(i), 3(l), 3(m) and 3(n). As explained in the figure's caption, the difference between Figures 3(a), 3(e) and 3(i), Figures 3(b), 3(f) and 3(l), Figures 3(c), 3(g) and 3(m) and Figures 3(d), 3(h) and 3(n) stands in the fact that the M parameter (that represents the coordinates of the optimal solution) is modified. In Figures 3(e) to 3(h) the different studied algorithms are reported in abscissa, where: algorithm 1 is PSO, algorithm 2 is PSE, algorithm 3 is RPSE, algorithm 4 is MPSO and algorithm 5 is MRPSO.

Looking at the number of successful runs (Figure 3, tables (a), (b), (c) and (d)), we can see that MRPSO performs better than the other methods and the differences between the results obtained by MRPSO and the other methods are always

**Number of successful runs:**



| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 74 | 6.82 |
| PSE | 44 | 5.58 |
| RPSE | 1 | 0.99 |
| MPSO | 180 | 4.24 |
| MRPSO | 188 | 3.35 |

(a)

| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 8 | 2.77 |
| PSE | 2 | 1.40 |
| RPSE | 1 | 0.99 |
| MPSO | 6 | 2.41 |
| MRPSO | 30 | 5.04 |

(b)

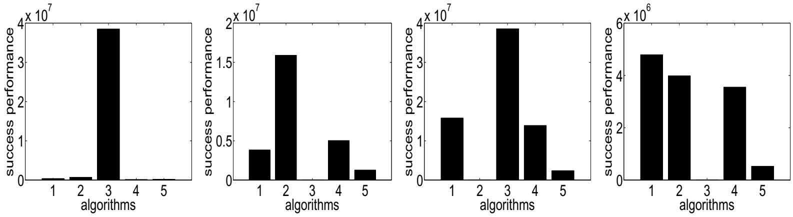
| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 2 | 1.40 |
| PSE | 0 | 0 |
| RPSE | 1 | 0.99 |
| MPSO | 2 | 1.40 |
| MRPSO | 16 | 3.83 |

(c)

| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 6 | 2.41 |
| PSE | 8 | 2.77 |
| RPSE | 0 | 0 |
| MPSO | 8 | 2.77 |
| MRPSO | 72 | 6.78 |

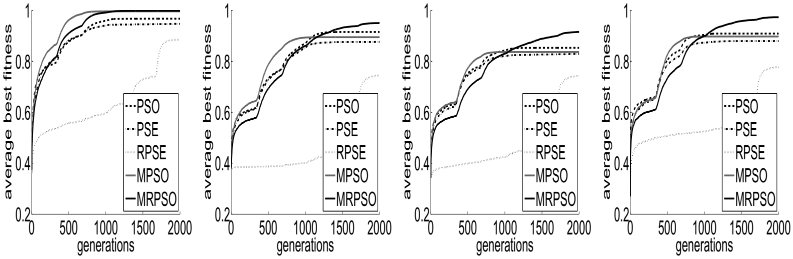
(d)

**Success Performance:**



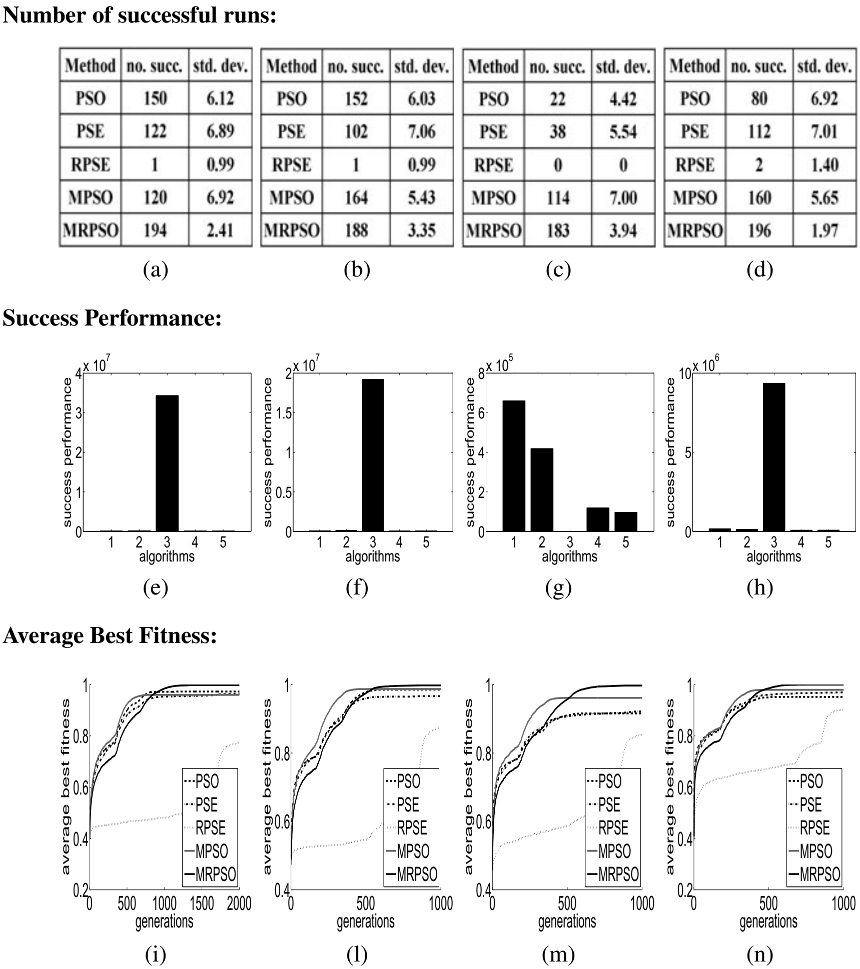
(e) (f) (g) (h)

**Average Best Fitness:**



(i) (l) (m) (n)

**Fig. 3** Results obtained by the five studied PSO variants on the *cosff* functions with 20 dimensions. The value of the $K$ parameter is equal to 10. In table (a) (respectively table (b), (c), (d)) we report the number of successful runs with their standard deviations for $M_1 = M_2 = ... = M_{20} = 0.1$. (respectively $M_1 = M_2 = ... = M_{20} = 0.2$, $M_1 = M_2 = ... = M_{20} = 0.3$, $M_1 = M_2 = ... = M_{20} = 0.4$). In plot (e) (respectively plot (f), (g), (h)) we report success performance for $M_1 = M_2 = ... = M_{20} = 0.1$. (respectively $M_1 = M_2 = ... = M_{20} = 0.2$, $M_1 = M_2 = ... = M_{20} = 0.3$, $M_1 = M_2 = ... = M_{20} = 0.4$). In plot (i) (respectively plot (l), (m), (n)) we report average best fitness against fitness evaluations for $M_1 = M_2 = ... = M_{20} = 0.1$ (respectively $M_1 = M_2 = ... = M_{20} = 0.2$, $M_1 = M_2 = ... = M_{20} = 0.3$, $M_1 = M_2 = ... = M_{20} = 0.4$). In plots (e) to (h) we identify PSO with 1, PSE with 2, RPSE with 3, MPSO with 4 and MRPSO with 5.



**Number of successful runs:**

| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 150 | 6.12 |
| PSE | 122 | 6.89 |
| RPSE | 1 | 0.99 |
| MPSO | 120 | 6.92 |
| MRPSO | 194 | 2.41 |

(a)

| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 152 | 6.03 |
| PSE | 102 | 7.06 |
| RPSE | 1 | 0.99 |
| MPSO | 164 | 5.43 |
| MRPSO | 188 | 3.35 |

(b)

| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 22 | 4.42 |
| PSE | 38 | 5.54 |
| RPSE | 0 | 0 |
| MPSO | 114 | 7.00 |
| MRPSO | 183 | 3.94 |

(c)

| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 80 | 6.92 |
| PSE | 112 | 7.01 |
| RPSE | 2 | 1.40 |
| MPSO | 160 | 5.65 |
| MRPSO | 196 | 1.97 |

(d)

**Success Performance:**

(e) (f) (g) (h)

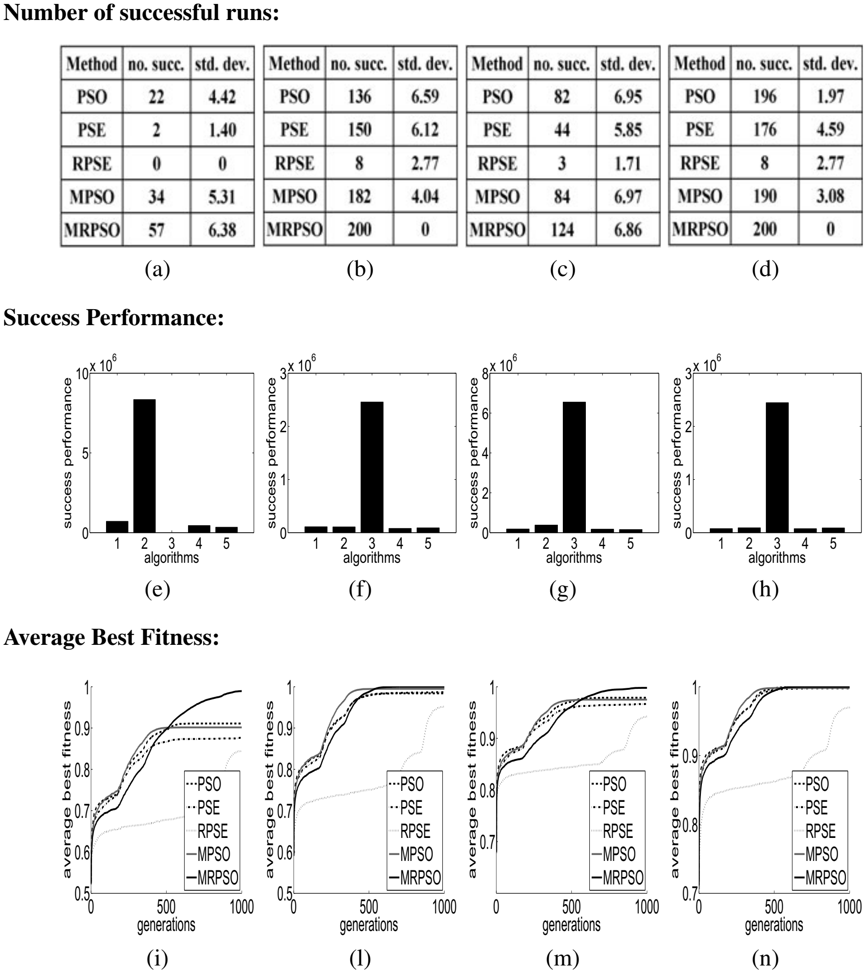
**Average Best Fitness:**

(i) (l) (m) (n)

**Fig. 4** Results obtained by the five studied PSO variants on the *cosff* functions with 10 dimensions. All the rest is as in Figure 3.

statistically significant. If we consider the success performance plots (Figure 3, plots (e), (f), (g), (h)) we as well can see that MRPSO outperforms the other methods, while the performances of MPSO are more or less comparable with the ones of PSO and the performances of PSE and RPSE are always the worse ones. Finally, looking at the average best fitness plots (Figure 3, plots (i), (l), (m), (n)), we can see that in all cases RPSE performs worse than the other methods. Figure 4 reports exactly the same results as Figure 3 for the *cosff* function, but for a dimensionality of the problem equal to 10, and it brings us to the same qualitative conclusions.

Figure 5 reports the results we have obtained on a set of *wtrap* functions for a dimensionality of the problem equal to 10. Also in this case, it is possible to see that MRPSO performs a larger number of successful runs than all the other studied methods, with statistically significant differences. About the success performance,



**Number of successful runs:**

| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 22 | 4.42 |
| PSE | 2 | 1.40 |
| RPSE | 0 | 0 |
| MPSO | 34 | 5.31 |
| MRPSO | 57 | 6.38 |

(a)

| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 136 | 6.59 |
| PSE | 150 | 6.12 |
| RPSE | 8 | 2.77 |
| MPSO | 182 | 4.04 |
| MRPSO | 200 | 0 |

(b)

| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 82 | 6.95 |
| PSE | 44 | 5.85 |
| RPSE | 3 | 1.71 |
| MPSO | 84 | 6.97 |
| MRPSO | 124 | 6.86 |

(c)

| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 196 | 1.97 |
| PSE | 176 | 4.59 |
| RPSE | 8 | 2.77 |
| MPSO | 190 | 3.08 |
| MRPSO | 200 | 0 |

(d)

**Success Performance:**

(e) (f) (g) (h)

**Average Best Fitness:**

(i) (l) (m) (n)

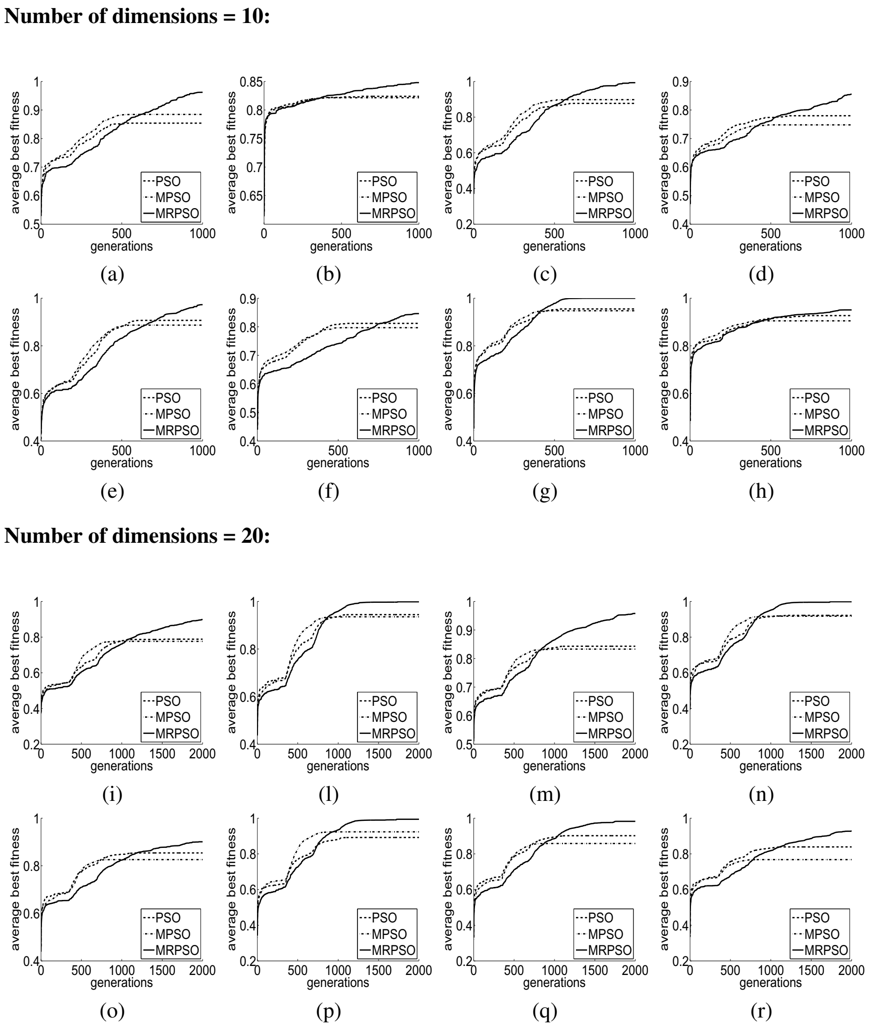
**Fig. 5** Results obtained by the five studied PSO variants on the *wtrap* functions with 10 dimensions. In table (a) (respectively table (b), (c), (d)) we report the number of successful runs for $B = 0.3$ and $R = 0.5$ (respectively $B = 0.5$ and $R = 0.5$, $B = 0.7$ and $R = 0.75$, $B = 0.9$ and $R = 0.75$). In plot (e) (respectively plot (f), (g), (h)) we report success performance for $B = 0.3$ and $R = 0.5$ (respectively $B = 0.5$ and $R = 0.5$, $B = 0.7$ and $R = 0.75$, $B = 0.9$ and $R = 0.75$). In plot (i) (respectively plot (l), (m), (n)) we report average best fitness against fitness evaluations for $B = 0.3$ and $R = 0.5$ (respectively $B = 0.5$ and $R = 0.5$, $B = 0.7$ and $R = 0.75$, $B = 0.9$ and $R = 0.75$). In plots (e) to (h) we identify PSO with 1, PSE with 2, RPSE with 3, MPSO with 4 and MRPSO with 5.

we can see that PSE and RPSE are the methods that have returned the worst results, while MRPSO has returned slightly better results compared to PSO and MPSO. Finally, also the average best fitness plots show that PSE has obtained the worst results and that MRPSO slightly outperforms all the other methods.

In Figure 6 we report the results for various *CosTrapFF* functions, obtained by changing the values of parameters $B$, $R$ and $K$ (see the figure's caption for the values of these parameters for each one of the reported plots). Given that, both for the *cosff* and the *wtrap* functions the PSE and RPSE models had returned the worst results, for the *CosTrapFF* function we have decided to report only the results returned by PSO, MPSO and MRPSO. Furthermore, in this case we report only the average best fitness plots (even though we also have the results of the other performance measures and they all confirm the general trend). All the plots of Figure 6 clearly confirm that MRPSO is the method that performs better among the studied ones. In particular, it is interesting to remark that, for all the studied cases, PSO and MPSO outperform MRPSO in the first part of the run, while MRPSO overcomes the other methods successively. This is clearly due to the repulsive component of MRPSO. In fact PSO and MPSO, that do not have this repulsive component, converge more rapidly, but they probably converge towards a local optimum. Indeed, in all the plots of Figure 6 we can remark that, after a certain number of generations, the curves of PSO and MPSO stabilize (they remain constantly at the same fitness value). This is not the case in general for MRPSO, whose curve is steadily growing up during all the run for all the studied functions, stabilizing only when the globally optimal solution (fitness value equal to one) has eventually been found. Also, we can remark that the curves of MRPSO do not hit the optimal fitness value before the end of the run for all the studied cases (see for instance Figures 6(b), 6(d), 6(f) and 6(b)). This also makes sense; in fact, if on the one hand the repulsive component helps to escape from locally optimal solutions, on the other hand it slows down the convergence to a globally optimal one when the right pick of the fitness landscape has been found. In conclusion, the results in Figure 6 confirm that MRPSO outperforms the other studied methods and they also suggest that MRPSO could be further improved by adding a *hill climbing* local optimizer in the final part of the evolution, as suggested in [18].

Figure 7 reports the results we have obtained on a set of *Rastrigin* functions for a dimensionality of the problem equal to 10. In this case, in the plots 7(i), 7(l), 7(m), 7(n) (concerning the average best fitness results), the curves are decreasing because the *Rastrigin* functions, differently from the *cosff*, *wtrap* and *CosTrapFF* ones, are minimization problems (i.e. low fitness values are better than high ones). What makes the difference more visible between MRPSO and the other methods is once again the number of successful runs: MRPSO has consistently performed a larger number of successful runs for all the studied instances, with statistically significant differences, as indicated by the standard deviations. Also the other measures confirm the suitability of MRPSO.

Figure 8 reports the results we obtained for the %F, LD50 and %PPB problems. For all the considered real-life problems, we report only the average best fitness plots. On the other hand, contrarily to the theoretical hand-tailored test functions



**Number of dimensions = 10:**

(a) (b) (c) (d)

(e) (f) (g) (h)

**Number of dimensions = 20:**

(i) (l) (m) (n)

(o) (p) (q) (r)

**Fig. 6** Average best fitness plots obtained by PSO, MPSO and MRPSO on some of the *CosTrapFF* functions. (a): $B = 0.1$, $R = 0.5$, $K = 10$; (b): $B = 0.1$, $R = 0.75$ and $K = 20$; (c): $B = 0.3$, $R = 0.25$, $K = 20$; (d): $B = 0.3$, $R = 0.5$, $K = 20$; (e): $B = 0.5$, $R = 0.25$, $K = 20$; (f): $B = 0.7$, $R = 0.5$, $K = 20$; (g): $B = 0.9$, $R = 0.5$, $K = 10$; (h): $B = 0.9$, $R = 0.75$, $K = 10$; (i): $B = 0.1$, $R = 0.25$, $K = 10$; (l): $B = 0.3$, $R = 0.25$, $K = 10$; (m): $B = 0.3$, $R = 0.5$, $K = 10$; (n): $B = 0.5$, $R = 0.25$, $K = 10$; (o): $B = 0.5$, $R = 0.5$, $K = 10$; (p): $B = 0.7$, $R = 0.25$ and $K = 10$; (q): $B = 0.9$, $R = 0.25$, $K = 10$; (r): $B = 0.9$, $R = 0.5$, $K = 10$.

**Number of successful runs:**

| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 2 | 1.41 |
| PSE | 6 | 2.41 |
| RPSE | 3 | 1.72 |
| MPSO | 39 | 5.60 |
| MRPSO | 107 | 7.05 |

(a)

| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 0 | 0.00 |
| PSE | 0 | 0.00 |
| RPSE | 0 | 0.00 |
| MPSO | 13 | 3.49 |
| MRPSO | 45 | 5.91 |

(b)

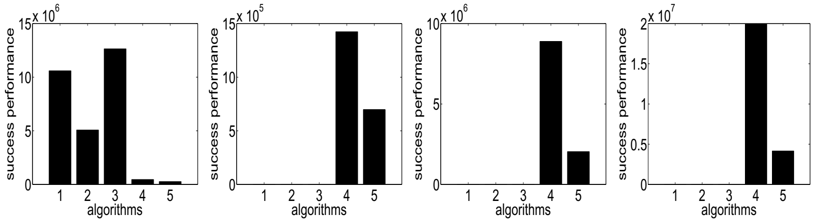
| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 0 | 0.00 |
| PSE | 0 | 0.00 |
| RPSE | 0 | 0.00 |
| MPSO | 2 | 1.41 |
| MRPSO | 16 | 3.84 |

(c)

| Method | no. succ. | std. dev. |
|---|---|---|
| PSO | 0 | 0.00 |
| PSE | 0 | 0.00 |
| RPSE | 0 | 0.00 |
| MPSO | 1 | 1.00 |
| MRPSO | 8 | 2.77 |

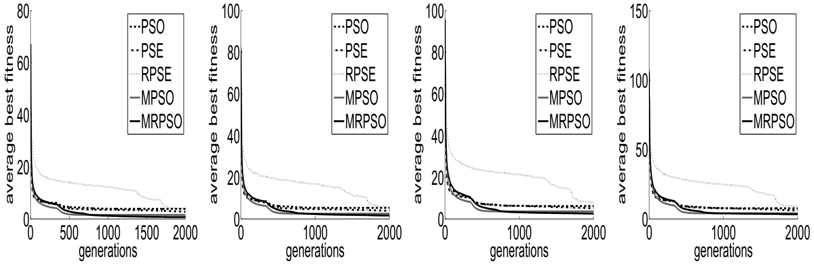
(d)

**Success Performance:**



(e) (f) (g) (h)

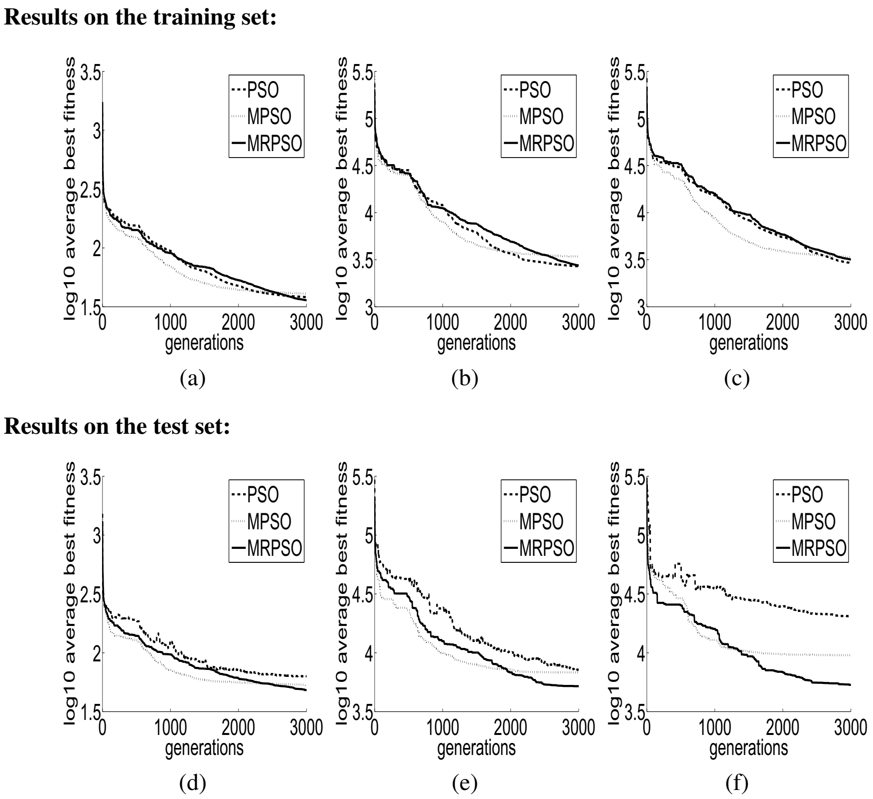
**Average Best Fitness:**



(i) (l) (m) (n)

**Fig. 7** Results obtained by the five studied PSO variants on the *Rastrigin* functions with 10 dimensions. In table (a) (respectively table (b), (c), (d)) we report the number of successful runs for $A = 4.0$ (respectively $A = 6.0$, $A = 8.0$, $A = 10.0$). In plot (e) (respectively plot (f), (g), (h)) we report success performance for $A = 4.0$ (respectively $A = 6.0$, $A = 8.0$ and $A = 10.0$). In plot (i) (respectively plot (l), (m), (n)) we report average best fitness against fitness evaluations for $A = 4.0$ (respectively $A = 6.0$, $A = 8.0$ and $A = 10.0$). In plots (e) to (h) we identify PSO with 1, PSE with 2, RPSE with 3, MPSO with 4 and MRPSO with 5.

considered until now, for the real-life problems we are also interested in studying the generalization ability of the proposed models. For this reason, plots (a), (b) and (c) report the average of the best (i.e. minimum) RMSE between outputs and targets (measure used as fitness) on the training set, while plots (d), (e) and (f) report the


**Results on the training set:**

(a)

(b)

(c)

**Results on the test set:**

(d)

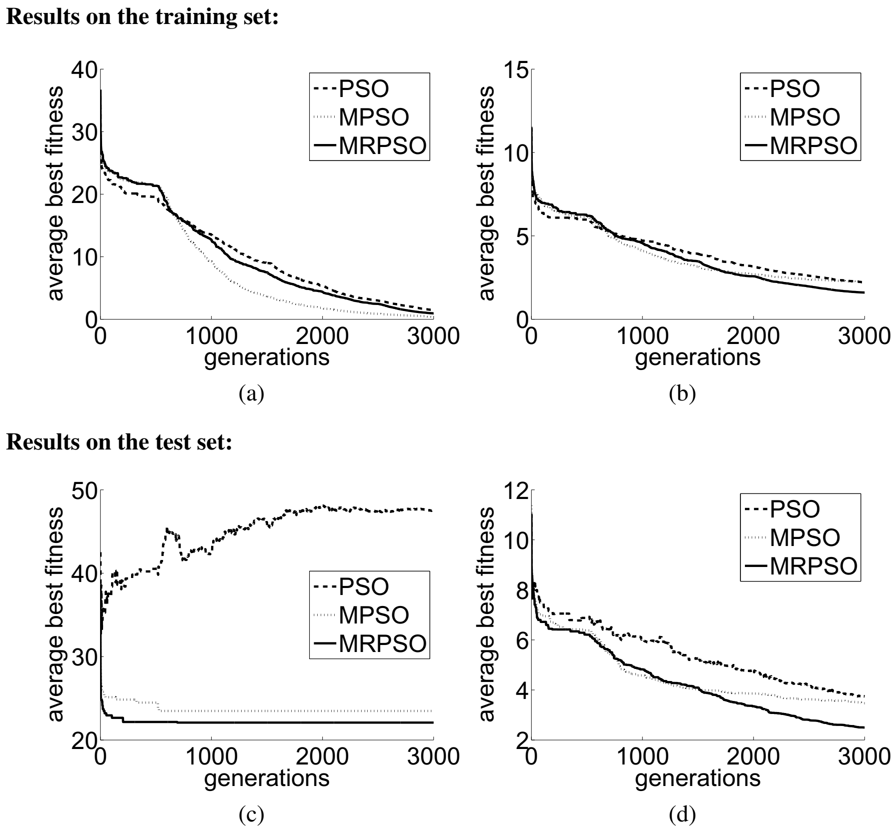
(e)

(f)

**Fig. 8** Average best fitness (RMSE between outputs and targets) against generations for the prediction of bioavailability (%F) (plots (a) and (d)), median oral lethal dose (LD50) (plots (b) and (e)) and plasma protein binding levels (%PPB) (plots (c) and (f)). Plots (a), (b) and (c) report results on the training set and plots (d), (e) and (f) report the average of the RMSE of the best individuals on the training set, calculated on the test set.

average of the RMSE of the best individuals on training, evaluated on the test set (for %F, LD50 and %PPB, respectively). For simplicity, for the real-life problems, we only report results for PSO, MPSO and MRPSO (which are the methods that have returned the best results on these problems). Figure 8 shows that, also for the considered real-life applications, MRPSO outperforms the other models, both on the training and on the test set. Interestingly, while the differences between the fitness values found by the different methods at termination is not statistically significant on the training set, it is statistically significant on the test set for LD50 and %PPB (Figure 8, plots (e) and (f)). Once again, MRPSO seems the best among the studied methods.

In Figure 9 we report the average best fitness results obtained for the FLU and DOCK datasets. As for the three previously considered applications, also in these



(a)

(b)

(c)

(d)

**Fig. 9** Average best fitness (RMSE between outputs and targets) against generations for the prediction of the response to Fludarabine (FLU) (plots (a) and (c)), and the prediction of docking energy (DOCK) (plots (b) and (d)). Plots (a) and (b) report results on the training set and plots (c) and (d) report the average of the RMSE of the best individuals on the training set, calculated on the test set.

cases PSO, MPSO and MRPSO obtain very similar fitness values on the training set, but once again MRPSO is the method that generalizes better, given that it obtains the best results on the test set, with statistically significant differences. Even more interestingly, for the FLU dataset we observe that the fitness on the training set steadily improves for all the three reported methods, included PSO; but for PSO this improvement on the training set corresponds to a visible deterioration of the results on the test set. On the other hand, the RMSE on the test set of both MPSO and MRPSO never worsens during the whole evolution. This indicates that PSO overfits training data for the FLU problem, while MPSO and MRPSO have a better generalization ability.

In synthesis, we can conclude that MRPSO consistently outperforms all the other studied methods for all the considered problems.

## 6 Conclusions and Future Work

The amount of computational effort required by search heuristics to generate high quality solutions to complex problems is generally very high. But the implicit parallelism of population-based heuristics make their parallelization a very natural and straightforward job. This particularly true for particle swarm optimization (PSO), given the high level of independency of the particles that form a swarm and also of the different regions of a swarm, if the latter in organized in a non-panmictic structure. In this chapter, we have introduced and studied four parallel and distributed PSO methods, that investigate four different ways of organizing a PSO system into sub-swarms.

The presented PSO methods, in fact, are variants of multi-swarm and attractive/repulsive PSO. They include a version in which swarms are interpreted as individuals of a genetic algorithm (GA), called particle swarm evolver (PSE); a variant in which a repulsive factor is added to the particles, called repulsive PSE (RPSE); a multi-island parallel and distributed model, called multi-swarm PSO (MPSO) and a variant of MPSO in which particles also contain a repulsive component, called multi-swarm repulsive PSO (MRPSO). The idea behind the definition of these models is that exchanging individuals at fixed time rates between the different swarms should allow to reinject diversity into those swarms, that otherwise would have the tendency to converge prematurely, in particular in case of complex problems. Furthermore, the repulsive component of two out of the four studied models should allow us to further stress the diversity maintenance into the whole system.

The performances of the studied models have been compared on a set of theoretical hand-tailored test functions and on five complex real-life applications. As already pointed out in our preliminary work [32, 33, 34], the experimental results presented here confirm that MRPSO outperforms the other considered PSO methods on all the studied problems. This is probably due to the fact that the double role played by the distribution of particles into island and by the repulsive factor of each particle allows us to maintain a higher degree of diversity compared to the other models. Interestingly, MRPSO has also obtained better results than the other methods on out-of-sample test data for all the considered real-life applications. MPSO has obtained better, or at least comparable, results than the ones of standard PSO, while PSE and RPSE are the methods that have obtained the worst results. The poor performances obtained by PSE and RPSE are probably due to the fact that in the GA system individuals are complicated structures (swarms), and this forces us to use relatively few individuals (10), which limits the exploration ability of the GA. Furthermore, the choice of defining as the fitness of a swarm the fitness of the best particle that belongs to it is questionable and variants deserve further investigation. Future work also includes an implementation of the proposed methods on

GPUs and also on geographically distributed networks. Last but not least, we plan a deeper study of the proposed models generalization ability.

## References

1. Archetti, F., Giordani, I., Vanneschi, L.: Genetic programming for anticancer therapeutic response prediction using the NCI-60 dataset. Computers and Operations Research 37(8), 1395–1405 (2010); Impact factor: 1.789
2. Archetti, F., Giordani, I., Vanneschi, L.: Genetic programming for QSAR investigation of docking energy. Applied Soft Computing 10(1), 170–182 (2010)
3. Archetti, F., Messina, E., Lanzeni, S., Vanneschi, L.: Genetic programming for computational pharmacokinetics in drug discovery and development. Genetic Programming and Evolvable Machines 8(4), 17–26 (2007)
4. Arumugam, M.S., Rao, M.: On the improved performances of the particle swarm optimization algorithms with adaptive parameters, cross-over operators and root mean square (rms) variants for computing optimal control of a class of hybrid systems. Journal of Applied Soft Computing 8, 324–336 (2008)
5. Blackwell, T., Branke, J.: Multi-Swarm Optimization in Dynamic Environments. In: Raidl, G.R., Cagnoni, S., Branke, J., Corne, D.W., Drechsler, R., Jin, Y., Johnson, C.G., Machado, P., Marchiori, E., Rothlauf, F., Smith, G.D., Squillero, G. (eds.) EvoWorkshops 2004. LNCS, vol. 3005, pp. 489–500. Springer, Heidelberg (2004)
6. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence: From Natural to Artificial Systems. Oxford University Press, Santa Fe Institute Studies in the Sciences of Complexity, New York, NY (1999)
7. Cagnoni, S., Vanneschi, L., Azzini, A., Tettamanzi, A.G.B.: A Critical Assessment of Some Variants of Particle Swarm Optimization. In: Giacobini, M., Brabazon, A., Cagnoni, S., Di Caro, G.A., Drechsler, R., Ekárt, A., Esparcia-Alcázar, A.I., Farooq, M., Fink, A., McCormack, J., O'Neill, M., Romero, J., Rothlauf, F., Squillero, G., Uyar, A.Ş., Yang, S. (eds.) EvoWorkshops 2008. LNCS, vol. 4974, pp. 565–574. Springer, Heidelberg (2008)
8. Clerc, M. (ed.): Particle Swarm Optimization. ISTE (2006)
9. Dioşan, L., Oltean, M.: Evolving the Structure of the Particle Swarm Optimization Algorithms. In: Gottlieb, J., Raidl, G.R. (eds.) EvoCOP 2006. LNCS, vol. 3906, pp. 25–36. Springer, Heidelberg (2006)
10. Fernández, F., Tomassini, M., Vanneschi, L.: An empirical study of multipopulation genetic programming. Genetic Programming and Evolvable Machines 4(1), 21–52 (2003)
11. Jiang, Y., Huang, W., Chen, L.: Applying multi-swarm accelerating particle swarm optimization to dynamic continuous functions. In: 2009 Second International Workshop on Knowledge Discovery and Data Mining, pp. 710–713 (2009)
12. Kameyama, K.: Particle swarm optimization - a survey. IEICE Transactions 92-D(7), 1354–1361 (2009)
13. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proc. IEEE Int. conf. on Neural Networks, vol. 4, pp. 1942–1948. IEEE Computer Society (1995)
14. Kennedy, J., Mendes, R.: Population structure and particle swarm performance. In: IEEE Congress on Evolutionary Computation, CEC 2002, pp. 1671–1676. IEEE Computer Society (2002)

15. Kennedy, J., Poli, R., Blackwell, T.: Particle swarm optimisation: an overview. Swarm Intelligence 1(1), 33–57 (2007)
16. Kennedy, J., Eberhart, R.C.: Swarm Intelligence. Morgan Kaufmann Publishers (2001)
17. Li, C., Yang, S.: Fast multi-swarm optimization for dynamic optimization problems. In: ICNC 2008: Proceedings of the 2008 Fourth International Conference on Natural Computation, pp. 624–628. IEEE Computer Society, Washington, DC (2008)
18. Liang, J.J., Suganthan, P.N.: Dynamic multi-swarm particle swarm optimizer with local search. In: 2005 IEEE Congress on Evolutionary Computation, CEC 2005, vol. 1, pp. 522–528 (2005)
19. Niu, B., Zhu, Y., He, X., Wu, H.: MCPSO: A multi-swarm cooperative particle swarm optimizer. Applied Mathematics and Computation 2(185), 1050–1062 (2007)
20. Poli, R.: Analysis of the publications on the applications of particle swarm optimisation. J. Artif. Evol. App. 2008, 3:1–3:10 (2008)
21. Poli, R.: Analysis of the publications on the applications of particle swarm optimisation. Journal of Artificial Evolution and Applications (2009) (in press)
22. N. C. M. Project. National Cancer Institute, Bethesda, MD (2008), `http://genome-www.stanford.edu/nci60/`
23. Riget, J., Vesterstrm, J.: A diversity-guided particle swarm optimizer - the arpso. Technical report, Dept. of Comput. Sci., Aarhus Univ., Denmark (2002)
24. Ross, S.M.: Introduction to Probability and Statistics for Engineers and Scientists. Academic Press, New York (2000)
25. Ross, D.T., et al.: Systematic variation in gene expression patterns in human cancer cell lines. Nat. Genet. 24(3), 227–235 (2000)
26. Sherf, U., et al.: A gene expression database for the molecular pharmacology of cancer. Nat. Genet. 24(3), 236–244 (2000)
27. Shi, Y.H., Eberhart, R.: A modified particle swarm optimizer. In: Proc. IEEE Int. Conference on Evolutionary Computation, pp. 69–73. IEEE Computer Society (1998)
28. Srinivasan, D., Seow, T.H.: Particle swarm inspired evolutionary algorithm (ps-ea) for multi-objective optimization problem. In: IEEE Congress on Evolutionary Computation, CEC 2003, pp. 2292–2297. IEEE Press (2003)
29. Suganthan, P., Hansen, N., Liang, J., Deb, K., Chen, Y., Auger, A., Tiwari, S.: Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization. Technical Report Number 2005005, Nanyang Technological University (2005)
30. Valle, Y.D., Venayagamoorthy, G., Mohagheghi, S., Hernandez, J., Harley, R.: Particle swarm optimization: Basic concepts, variants and applications in power systems. IEEE Transactions on Evolutionary Computation 12(2), 171–195 (2008)
31. Vanneschi, L.: Theory and Practice for Efficient Genetic Programming. Ph.D. thesis, Faculty of Sciences. University of Lausanne, Switzerland (2004)
32. Vanneschi, L., Codecasa, D., Mauri, G.: An empirical comparison of parallel and distributed particle swarm optimization methods. In: Pelikan, M., Branke, J. (eds.) GECCO, pp. 15–22. ACM (2010)
33. Vanneschi, L., Codecasa, D., Mauri, G.: A study of parallel and distributed particle swarm optimization methods. In: Proceeding of the 2nd Workshop on Bio-Inspired Algorithms for Distributed Systems, BADS 2010, pp. 9–16. ACM, New York (2010)
34. Vanneschi, L., Codecasa, D., Mauri, G.: A comparative study of four parallel and distributed PSO methods. New Generation Computing (2011) (to appear)
35. Wang, Y., Yang, Y.: An interactive multi-swarm pso for multiobjective optimization problems. Expert Systems with Applications (2008) (in press), `http://www.sciencedirect.com` (to appear)

36. Wu, Z., Zhou, J.: A self-adaptive particle swarm optimization algorithm with individual coefficients adjustment. In: Proc. IEEE International Conference on Computational Intelligence and Security, CIS 2007, pp. 133–136. IEEE Computer Society (2007)
37. You, X., Liu, S., Zheng, W.: Double-particle swarm optimization with induction-enhanced evolutionary strategy to solve constrained optimization problems. In: IEEE International Conference on Natural Computing, ICNC 2007, pp. 527–531. IEEE Computer Society (2007)
38. Zhigljavsky, A., Zilinskas, A.: Stochastic Global Optimization. Springer Optimization and Its Applications, vol. 9 (2008)
39. Zhiming, L., Cheng, W., Jian, L.: Solving contrained optimization via a modified genetic particle swarm optimization. In: Workshop on Knowledge Discovery and Data Mining, WKDD 2008, pp. 217–220. IEEE Computer Society (2008)

# The Generalized Island Model

Dario Izzo, Marek Ruciński, and Francesco Biscani

**Abstract.** The island model paradigm allows to efficiently distribute genetic algorithms over multiple processors while introducing a new genetic operator, the migration operator, able to improve the overall algortihmic performance. In this chapter we introduce the generalized island model that can be applied to a broad class of optimization algorithms. First, we study the effect of such a generalized distribution model on several well-known global optimization metaheuristics. We consider some variants of Differential Evolution, Genetic Algorithms, Harmony Search, Artificial Bee Colony, Particle Swarm Optimization and Simulated Annealing. Based on an set of 12 benchmark problems we show that in the majority of cases introduction of the migration operator leads to obtaining better results than using an equivalent multi-start scheme. We then apply the generalized island model to construct heterogeneous "archipelagos", which employ different optimization algorithms on different islands, and show cases where this leads to further improvements of performance with respect to the homogeneous case.

## 1 Parallelizing Optimization Tasks

Parallel computing is an indispensable tool of modern science. Dividing the given task into independent units that can be performed in parallel can lead to significant reduction of the time needed to obtain desired results. While in the past decades parallel computing machines were a scarce resource, available exclusively to the

Dario Izzo · Francesco Biscani
Advanced Concepts Team, European Space and Technology Center (ESTEC),
The Netherlands
e-mail: dario.izzo@esa.int,bluescarni@gmail.com

Marek Ruciński
Centre for Robotics and Neural Systems, Univeristy of Plymouth, United Kingdom
e-mail: marek.rucinski@plymouth.ac.uk

F.F. de Vega et al. (Eds.): Parallel Architectures & Bioinspired Algorithms, SCI 415, pp. 151–169.
springerlink.com

customers of computing centers, nowadays virtually every personal computer sold has parallel computing capabilities in the form of multi-core/multi-threaded CPU or even massively parallel-capable GPU. It is not surprising then that much research is being done about how to utilize best these now omnipresent capabilities.

In the context of optimization, parallel architectures were considered from very early on, actually the first concepts appeared when parallel computing machines were still in a primitive stage [1]. Much of both practical and theoretical research has been done on parallelization of many types of algorithms [14, 2, 16]. Equally much work was then needed to classify and categorize the multitude of existing approaches in order to provide the global view on the subject. From our standpoint, the ways in which parallel computing has been utilized in order to speed up the processes of optimization could be roughly divided into three categories described in the following paragraphs.

The first class of algorithms benefits from the possibility of using parallel computing to speed up the computation of the objective function value for a given solution. This is of course most practical in cases where such calculation requires a relatively large amount of computational effort, such as those where simulations of some sort are involved. Because calculation of the objective function is the core operation of every optimization process, any gain in the execution time of this operation translates directly to shortening of the optimization process itself, often resulting in a linear speed-up. Because parallelism remains "encapsulated" in the objective function, the optimization algorithm does not need to be modified in any way.

Second class of approaches to exploit parallel architectures in optimization takes advantage of the fact that often one step of the optimization algorithm requires evaluating many solutions. This is usually the case for algorithms like the Generic Algorithm (GA) or Differential Evolution (DE), which operate on a set of solutions, often called a *population*. Because objective function evaluations for different solutions from the population are completely independent, they can be easily performed in parallel, shortening the time needed to evaluate the whole set of solutions, and again resulting in a linear speed-up. Also in the case of this class only simple modifications to the original sequential algorithm are required, as the only thing that changes is that the solutions in the population are evaluated in parallel instead of iteratively in a loop. This type of parallelization/distribution strategy is also called master-slave model [5] in the context of GAs, where it was born, as one CPU (the Master) carries out the computations necessary to apply the genetic operators, while the slaves CPUs are delegated to evaluate the chromosome fitness. The common characteristic of the two approaches described above is that the original optimization algorithm is not modified in any significant way. Although some changes naturally have to be done to allow for a parallel implementation, the logic and flow of the optimization process does not change – the results obtained with the sequential and parallel variant of the algorithm are exactly the same, only that in the latter case they are available faster.

The final, third category of approaches is the one in which introduction of the parallel execution is done not only to speed up the processing, but also to exploit completely new dynamics present because of the existence of parallel populations and possible interactions between them. A classical example of such an approach is so-called *coarse-grained* or *island model* parallelization scheme designed for GAs in which many populations co-exist and exchange individuals at certain intervals. Our definition includes however also other schemes, like the *fine-grained* model, in which a local structure of the neighborhood between solutions is introduced and exploited. The important difference between this category of algorithms and previously introduced ones is the necessity of adaptation of the original optimization paradigm.

In this chapter we focus on the island model parallelization scheme. For parallel GAs (PGAs), it has been shown to be superior to the *global* approach (i.e. with only one population) both in practice and by theoretical analysis [5]. In one of our previously published works we have shown that the island model paradigm can be relatively easily used with algorithms other than GAs, both those explicitly utilizing populations and not [17]. While in that work we were focussed mainly on the impact of the network topology defining the island connectivity (migration paths) here we take a step back and we formally introduce the generalized migration operator which allows for a coarse-grained parallel implementation of virtually any optimization algorithm (provided that it fulfills few simple criteria) as well as study heterogeneous clusters of *islands* implementing various algorithms which cooperate together in one optimization process to "evolve" good solutions. Based on computational experiments involving 10 algorithms and 12 benchmark problems we show that introduction of the migration operator allows in case of most of the considered optimization algorithms to obtain better solutions than in a sequential multi-start case with the same amount of computational effort. Subsequently, we make use of the knowledge about preferences of particular algorithms toward migration we gained in those experiments in order to construct a heterogeneous archipelago of various algorithms and compare its performance with the homogeneous case.

Motivations behind performing the study presented herein are the following:

- because of the technological advances, implementation of parallel optimization algorithms is these days much less constrained by the hardware architecture than in the past. This enables much more flexible experimentation with various parallel designs, as the problem of communication overheads became negligible in most cases;
- computation of the objective function is, in a number of representative cases, fast enough to allow for parallel processing involving a large number of solutions grouped in many generously sized populations on one machine;
- parallel processing involving many different optimization algorithms and exchange of information between them is expected to improve the exploration-exploitation balance thanks to the additional variety;

The remaining part of the chapter is organized as follows. First, we present the generalized island model paradigm and an in-depth discussion of the parameters involved. Next, we provide a short overview of PaGMO, our open-source implementation of the model. Then, we present two computational experiments: #1) "the migration dilemma" aimed at establishing for which algorithms introduction of the migration operator yields better results than an equivalent multi-start execution, and #2) "can they cooperate?" comparing performance of homogeneous and heterogeneous archipelagos. The chapter ends with the presentation of conclusions from the computational experiments and a short discussion of future prospects.

## 2 The Generalized Island Model

### *2.1 Definition*

The island model has been proposed as an extension of a traditional GA with the intention of improving the diversity of solutions and thus delaying stagnation. First works to mention the idea of using a number of subpopulations can be dated as early as 1967. With the dawn of parallel and distributed computing machines in the 1980s, the research on PGAs gained significant momentum (see [1] for a more detailed historical note). The core idea behind the island model is the introduction of a structure to the population used in the original GA. Instead of permitting recombination between any two individuals in the solution pool, this possibility becomes restricted only to solutions belonging to the same sub-population, or *deme*, which number traditionally varies between a few to a few dozen. The sub-populations evolve mostly independently, however at certain relatively sparsely distributed moments of time they are allowed to exchange solutions in a process called migration. From the point of view of the GA theory, the transition from the original model to the island model has thus been frequently viewed as a modification of the selection operator [18]. However, the perspective that allows for a few more general conclusions is to consider the island model simply as running a certain number of global GA algorithms in parallel with the additional introduction of a new operator called the migration operator. Informally, this operator is engaged at certain points of time between two consecutive generations of the original GA algorithm, and its job is both to select individuals from the current island to be sent to other islands, as well as potentially introduce foreign individuals to the local population. Note how, from the point of view of the migration operator, the details of the optimization process that take place in between two events of migration are irrelevant. GAs on different islands could easily use different parameters for example the selection rule, crossover operator or mutation probability. The optimization process would still work, and actually could perform even better than the variant with fixed parameters across islands thanks to the additional variability. One can go even a next step further, and state that for the migration operator it does not matter at all what kind of algorithm is used on the islands, and whether different islands use the same algorithms or not, as long as all the islands use the same problem solution coding and the migration can be "plugged in" on every island. This informal discussion shows that the island model is a general paradigm that can be applied not only to genetic or evolutionary algorithms, but to a much broader family of optimization processes, and even can be used to form heterogeneous archipelagos of islands which use various algorithms. Let us now introduce the concept more formally.

We define an *archipelago* $\mathbb{A}$ as a couple:

$$\mathbb{A} = \langle \mathbb{I}, \mathscr{T} \rangle \tag{1}$$

where $\mathbb{I} = \{I_1, I_2, \ldots, I_n\}$ is the set of *islands*, and $\mathscr{T}$ is the *migration topology*, a directed graph with $\mathbb{I}$ as the set of vertices. Every island $I_i$ is a quadruple:

$$I_i = \langle \mathscr{A}_i, P_i, \mathscr{S}_i, \mathscr{R}_i \rangle \qquad i = 1, 2, \ldots, n \tag{2}$$

where $\mathscr{A}_i$ is the *optimization algorithm* used by the $i$-th island, $P_i$ the population there contained, and $\mathscr{S}_i$ and $\mathscr{R}_i$ are respectively the *migration-selection policy* and *migration-replacement policy* for the island. A population $P_i$ is a couple $\langle \mathscr{P}, \mathbb{P} \rangle$ containing a set of individuals $\mathbb{P}$ whose fitness value is always referred to the problem $\mathscr{P}$ we are considering. Note that, unlike other works, we speak of populations always in connection with their fitness values (i.e. the problem they refer to).

The optimization algorithm $\mathscr{A}$ is any optimization process supporting an evolution operator $P' \leftarrow \mathscr{A}(P, \mu)$, where $\mu$ denotes the migration interval (i.e. the number of allowed algorithmic iterations before a migration is allowed). Algorithms may or may not have the following distributive property:

$$\mathscr{A}(\mathscr{A}(P, \mu_1), \mu_2) = \mathscr{A}(P, \mu_1 + \mu_2) \tag{3}$$

if they do not they are referred to as *adaptive* . Note that $\mathscr{A}$ can also operate on populations containing one only individual, in which case we write $|P| = 1$ (simulated annealing is one of such algorithms)

The migration-selection policy $\mathscr{S}$ determines the deme $\mathbb{M} \subseteq \mathbb{P}$ to be sent to other islands via migration. We write this as $\mathbb{M} \leftarrow \mathscr{S}(P)$. In turn, given a deme $\mathbb{M}$, the migration-replacement policy $\mathscr{R}$ specifies how $\mathbb{M}$ could be inserted into a population $P$. We write this as $P' \leftarrow \mathscr{R}(P, \mathbb{M})$.

We are now ready to specify the flow of the general coarse-grained optimization algorithm. On every island $I_i$, processing follows the pseudo-code presented on listing 1. Optimization on all islands is performed in parallel. The final result of the optimization is the best solution found over all islands.

**Listing 1.** Pseudo-code for the island $I_i$

```
initialize $P$
while !stop_criteria
    $P' \leftarrow \mathscr{A}_i(P, \mu_i)$
    $\mathbb{M} \leftarrow \mathscr{S}_i(P')$
    /* Send $\mathbb{M}$ to islands adjacent to $I_i$ in $\mathscr{T}$ */
    /* Let $\mathbb{M}'$ be the set of solutions received from adjacent islands */
    $P'' \leftarrow \mathscr{R}(P', \mathbb{M}')$
    $P \leftarrow P''$
```

## *2.2 Parameters*

The generalized island model just introduced has a number of parameters which require further clarification and discussion.

### 2.2.1 Implementation of Migration

The formal definition given above does not specify all details involved in the implementation of the migration. One question is whether the migration should be *synchronous* or *asynchronous*, or in other words: should the communication event happening in line 5 of 1 block all islands until communication is completed?. Both approaches have advantages and disadvantages. Asynchronous communication seems to be an obvious choice for distributed computing architectures, because no or little global control over the execution of the code on islands is required (just initialization of the tasks and gathering of the final results). It provides good scalability and maximally utilizes available computing resources, as communication overheads are limited to the necessary minimum. On the other hand, because processing time on islands is usually unpredictable, it is also not possible to predict when migration events occur, an thus the resulting algorithm is non-deterministic, non-repeatable and hard to control or debug. These issues are solved if the communication between all islands is synchronized. Under this assumption it is possible to obtain a deterministic process. The price that one has to pay, however, is a potentially significant overhead resulting from the synchronization of all islands and the fact that the slowest island dictates the execution time of the whole process. The presence of a global synchronization mechanism thus limits the speed-up and the scalability of the system.

Should the asynchronous migration be the choice, there are still certain decisions to be made about its implementation. Two opposite approaches could be called migration *initiated by source* and migration *initiated by destination*. The former is the most intuitive implementation of a simple message passing protocol. As soon as an island reaches line 5 of the island model code introduced above, the migrating individuals are sent to adjacent islands. It is very likely that a destination island will not be able to insert incoming individuals immediately to the local population because of being for the moment busy executing the optimization step in line 3, thus every island needs a buffer in which incoming migrating individuals are stored in between two events of execution of line 7 of the pseudo-code. We call this approach initiated by source, because the *transmission* of solutions from one island to another is performed when the *source* island reaches the communication step. In contrast, in the migration initiated by destination, when an island reaches line 5 of the island model code, the solutions are not sent to individuals just yet, put placed in a local buffer or “database”. Then, when an island is about to execute the line 7 of the code, it contacts adjacent islands and fetches individuals available in their “databases” at the moment. The individuals are thus *transmitted* when the *destination* island ask for them. At the first glance this may seem to be just an implementational detail, however these two strategies differ quite significantly in a way that may affect the performance of the optimization. If one consideres a topology with one island acting as a "hub" in the center, in the case of migration initiated by source this island will receive much more individuals than in the case of migration initiated by the destination. Or, one can imagine a situation where one island is much slower than its neighbors: here migration initiated by destination could mean that the neigbors would receive the same individuals several times, as the slow island would not manage to update its "database" in between communication events initiated by fast neighbors.

Another detail of the migration implementation that is not explicitly specified in our definition is the *method of distribution* of the migrating solutions. Migration topology $\mathscr{T}$ specifies which islands are adjacent and thus are allowed to exchange solutions. However when an island has more than one neighbor, there is plenty of different scenarios possible. One strategy could be to send the migrating individuals to all adjacent islands. On the other side of the spectrum of available choices, solutions could be sent only to one neighbor – for example selected randomly or in an algorithmic fashion (e.g. round-robin). It is likely that the choice will have an impact on the performance of the optimization process. It is reasonable to expect that in the former case, the effect of the choice of the migration topology should be more pronounced than in the latter. The frequency of communication between two particular islands in the latter case is lower, and also the total amount of information being exchanged in the whole archipelago is smaller, what may be significant for densely connected archipelagos with many islands. The method of distribution of individuals is moreover connected with other parameters of the island model, especially the migration-selection policy $\mathscr{S}$: a common strategy of migration is to allow the whole local population to be sent to the neighbors, but partitioned in such a way that each neighbor receives a distinct part of it [5].

### 2.2.2 Number of Islands $n$

The *number of islands* $n$ is one of the most straightforward parameters of an archipelago. There may be many factors influencing the choice of $n$ for a particular application: the computing platform used, the number of available computing machines, or the choice of the migration topology. Along with the progress in computing technology, this choice becomes now less and less constrained by the hardware. More and more researchers have access to large numbers of relatively powerful computers (for example on a university or company network) that could be used to perform distributed optimization in their idle time. Common sense prediction is that the more islands are involved in the computations, the better the final result of the optimization should be, as more computational resources are employed. It is reasonable to expect that using an archipelago with many islands will provide various benefits. For example this can remove the necessity (or at least reduce the importance of) fine-tuning of the parameters of the employed optimization algorithm. One can just create an archipelago in which different islands use different combinations of parameters and hope that islands with "good" parameters will drive the optimization process, compensating for the other, underperforming islands [11]. One should keep in mind however, that the bigger the number of islands, the more important the choice of the migration topology becomes. For instance, comparing archipelagos with 8 and 1024 islands, for a fully-connected topology this means nearly 20000 more connections between islands, while for the hypercube topology, the increase is only 500-fold.

### 2.2.3 Migration Topology $\mathscr{T}$

Another parameter of the archipelago is the *migration topology* $\mathscr{T}$. In the past, the migration topology was often dictated by the hardware on which the island model was implemented. Dedicated parallel machines without shared memory were used and the topology of the connections between available processors was thus used also when implementing an island model. Distributed computing architectures available today provide much greater flexibility, and the migration topology can be selected appropriately to the task at hand. The choice of the migration topology was the focus of a previous study [17] where the effect of the topology choice was related to the quality of the final results. A selection of migration topologies that can be found in the literature about the island model and other parallelization schemes is presented in figure 1. How to choose the migration topology for a given problem and optimization algorithm(s) remains an open question. Probably the most significant way in which the migration topology affects computation in the island model is shaping the information flow in the archipelago. From this point of view, most relevant parameters of the topology graph are: the average length of a path between two islands (or its diameter, i.e. the longest of the shortest paths between any two islands), the number of edges (which can be viewed as the measure of the communication overheads), and the degree of connectivity (average or maximum number of neighbors per island).

### 2.2.4 Migration Interval $\mu$

The *migration interval* $\mu$ specifies the iterations of $\mathscr{A}$ before a migration occurs (an equivalent term sometimes found in the literature is *migration frequency*). The value of the migration interval should be chosen by taking into account the properties of the considered optimization algorithm $\mathscr{A}_i$, in particular, the convergence rate of the algorithm in relation to $\mathscr{P}$. It is reasonable to assume that the algorithm should be allowed to achieve a sensible progress in optimization in between two migration events. It has been shown that in certain cases too short migration interval may cause the algorithm to stop working as expected [21]. This is why the choice of the migration interval should be proceeded at least by experimental observation of the optimization progress on an isolated island. Another idea is to use a dynamically changing migration interval, and performing migration only after the algorithm achieves stagnation [4].

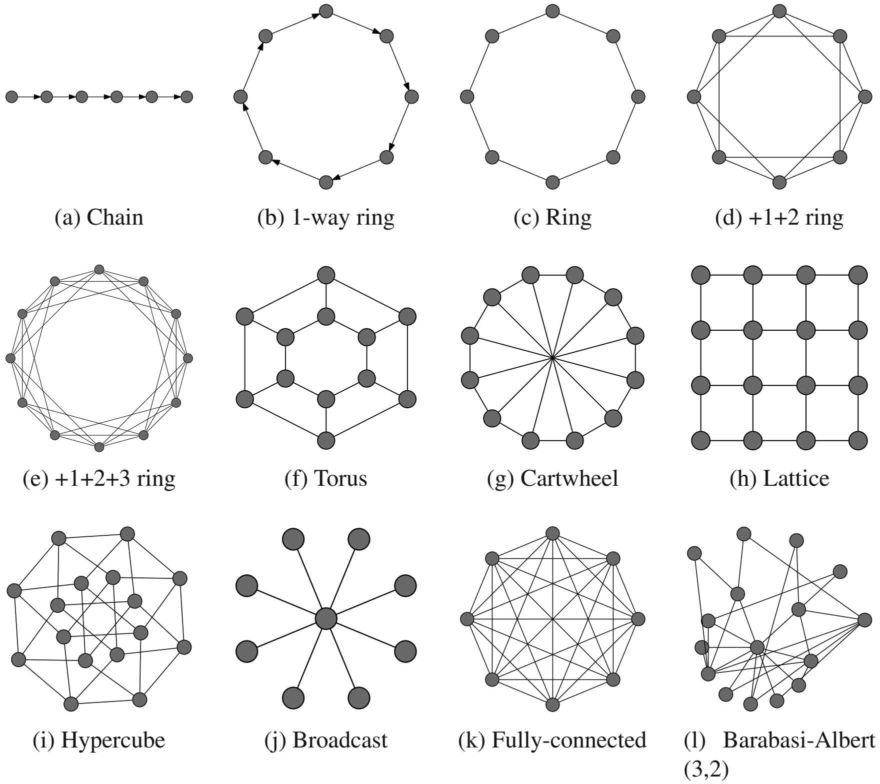


**Fig. 1** Common migration topologies

### 2.2.5 Migration-Selection Policy $\mathscr{S}$ and Migration-Replacement Policy $\mathscr{R}$

The island model requires specification of two operations: the strategy of the *selection* of the solutions from the local population to be sent to adjacent island(s), as well as the method of *integration* of incoming individuals into the local population. In our generalised island model, these details are specified by the migration-selection policy $\mathscr{S}$ and migration-replacement policy $\mathscr{R}$. Note that among the details of selection and replacement there is the number of solutions to send or accept (usually called the *migration rate*), and that in principle $\mathscr{S}_i$ and $\mathscr{R}_i$ could differ from island to island. The spectrum of available reasonable choices for $\mathscr{S}$ and $\mathscr{R}$ could be narrowed down to *randomness* and *elitism* [18], however also more sophistcated mechanisms could be used, for example the Metropolis criterion.

## 3 The Code: PaGMO

All the experiments described in this chapter have been performed using a software platform for global optimization called PaGMO [3]. PaGMO has been developed within the Advanced Concepts Team of the European Space Agency, originally with special focus on spacecraft trajectory optimization problems, and later extended to be a general-purpose optimization framework.

The generalized island model is implemented within PaGMO using different threads of execution and a shared memory model, in order to take full advantage of contemporary multicore architectures. Each island executes the optimization algorithm in a separate thread, whose scheduling is managed asynchronously by the operating system (which will typically migrate the threads among the available computing cores as needed to maintain a balanced workload). Locking primitives such as mutexes, condition variables and thread barriers are used to avoid contention issues and manage access to resources shared among different threads. The parallelization of optimization tasks through the coarse-grained approach of the island model results in an "embarrassingly parallel" workload, since most of the CPU time is typically spent in the execution of the optimization algorithm in separate threads. Synchronization points, mostly due to the implementation of the migration operator, are sparse and lightweight, and as result PaGMO's performance scales linearly with the number of available cores.

Although not used in the experiments described in this chapter, PaGMO also has the capability of implementing the island model over a network of connected computers (e.g., a high-performance computing cluster) using the Message Passing Interface (MPI) [19]. In this operative mode, each island on the master node first serializes the objects representing the algorithm, the optimization problem and the population, and then transmits them over the network to another computer in the cluster, a slave node, where the optimization is actually executed in a local process. When the optimization process finishes, the new (optimised) population is serialised on the slave node and sent back to the master node, where it replaces the original population in the island. With respect to the multithreaded, shared memory version of the island model, this implementation incurs in the additional overheads and latencies of object serialization and network transmission. On the other hand, the ability to run on computer clusters greatly enhance the parallelisation potential which would otherwise be limited by the number of cores available on a single machine.

Written in C++ with the availability of Python bindings for increased ease of use and user-friendliness, designed with portability in mind and tested on GNU/Linux, OSX and Windows, PaGMO is free/libre/open-source software licensed under the GNU Public License. It can be downloaded from the website `http://pagmo.sourceforge.net`

# 4 Experiments

## *4.1 Problems*

We consider some standard multimodal mathematical function with diverse properties (separability, etc.) together with "real world" problems. The problems have been selected to pose a real challenge to the algorithms studied and to maintain the overall CPU time for the entire test reasonable. The standard test functions we selected are: Rastrigin, Rosenbrock, Griewank, Ackley, De Jong, Levy5 (exact details on the implementation of these common mathematical functions can be found directly in the PaGMO code [3]. All problems are here instantiated with a dimension $d = 50$. We also consider two examples of the Lennard-Jones test function [23] corresponding to 11 and 17 atoms. We then extract some cases from the GTOP database [22], a European Space Agency's repository of difficult global optimization test functions related to interplanetary space travel. Out of such a database we have selcted the three problems named Rosetta, Cassini 2 and Messenger Full, also implemented and available in PaGMO [3].

## 4.2 Algorithms

We consider a number of global optimization algorithms based on diverse paradigms. Our selection is certainly not exaustive but it includes some of the arguably most popular algorithms that have proved their value extensively in the past decades.

1. DE: Differential Evolution – This algorithm by Storn and Price [20] has a rather standard implementation we here test. In particular we consider two variants of the algorithm commonly called rand/1/exp and rand/1/bin, that differ, essentially, in the crossover type (binomial or exponential). The algorithm parameters [20] are set to be $CR = 0.9$ and $F = 0.8$. We allow for 500 generations over a population of dimension 20 for each algorithmic call, corresponding to 2000 function evaluations.
2. PSO: Particle Swarm Optimization – The original algorithm proposed by Kennedy and Eberhart [13] was later improved by the introduction of a so called constriction factor [6]. We here consider this version of the algorithm together with the variant named Fully Informed Particle Swarm [15]. For both algorithms we select rather canonical values for the different coefficients [6]. In the canonical algorithm a ring topology with two neighbours is used, while for the FIPS we use a lattice topology, as suggested in [15].
3. SA: Simulated Annealing – There are many different implementations of simulated annealing available in the literature, in this work we consider the adaptive neighbourhood simulated annealing as introduced by Corana et al. [8]. The main algorithm parameters are the starting and final temperatures ($T_s$, $T_f$) and the number of total iterations $n$ (these can be approximately be taken as the number of function evaluation made). Other parameter also control the performances of the we use $N_s = 20$ and $N_T = 1$, where $N_s$ is the number of cycles and $N_T$ the number of step adjustments (see Corana et al. [8]). The cooling schedule implemented is a geometric cooling schedule. For all problems we use $T_s = 0.1$, $T_f = 0.001$.
4. HS: Harmony Search – This metaheuristic algorithm [9] is inspired by the improvisation process of musicians. In the HS metaphor, each decision variable is seen as a musician which, through improvisation, generates new notes together

with the other musicians in order to find a good "harmony" (i.e., the global best). More specifically, in the HS algorithm new candidate solutions are generated either by randomly choosing components of existing solutions and adjusting them by increasing/decreasing their values, or by generating components of the candidate solution completely randomly. The parameters of the algorithm include the population size, the probability of generating new components by adjusting existing components instead of by random selection, and the amount of adjustment. In this study, we have adopted the canonical HS algorithm described in the original paper. To our knowledge, this is the first time that the island model paradigm is tested on a HS algorithm.

5. GA: Genetic Algorithm – This class of algorithms [10] is wide and contains many different variants that do not have an agreed common implementation. Here we consider a rather straight forward implementation with tournament selection, exponential crossover and elitism of one individual (i.e. the best among the parents is reinserted in the new generation substituting the worst of the offsprings if better). As for the mutation, we consider two different algoritmic variants, one with gaussian mutation (with the gaussian bell having a standard deviation of $\sigma = 0.1$ with respect to the width of the linear bounds on that particular component) and one having random mutation implemented. The crossover coefficient $c = 0.95$ and the mutation rate is $m = 0.05$ applied to each component of the cromosome. As to allow for a number of iterations (migration interval) $mu = 2000$ we let the
6. ABC: Artificial Bee Colony – This rather new metaheuristics still did not have the time to "mutate" much from the original version proposed by Karaboga [12] and we thus here consider that original algorithm. The parameters are the iteration number $n$ which we set to $n = 50$ as to allow 2000 function evaluations for each algorithmic call, and the number of tries $m$ after which a source of food is dropped if not improved. We here use $m = 20$.

For all algorithms we set the migration interval $\mu_i$ so that 2000 function evaluation are allowed in between migrations.

## 4.3 Experiment #1: The Migration Dilemma

Consider the pair $< \mathscr{P}, \mathscr{A} >$ containing an optimization algorithm $\mathscr{A}$ and an optimization problem $\mathscr{P}$. A common approach to use $\mathscr{A}$ to solve $\mathscr{P}$ is to run $\mathscr{A}$ indipendently $N$ times over $\mathscr{P}$ and record, at the end, the best result found $x$. Alternatively, one could run $N$ times $\mathscr{A}$ over $\mathscr{P}$ allowing solutions to be exchanged (migrate) among the $N$ runs of $\mathscr{A}$ and and record, at the end, the best result found $y$. The two approaches require the exact same computational effort (i.e. it takes the same CPU time to get $y$ or $x$) if one neglects the overhead of migrating and exchanging information among algorithmic runs (i.e. CPUs) Accumulating statistical evidence on the difference between $x$ and $y$ helps us in evaluating the benefits of the generalized island model and to answer the simple question "should I migrate?" and thus will be the focus of the results here presented. For each pair $< \mathscr{P}, \mathscr{A} >$

and for $N = 8, N = 32$ we build samples (containing 200 instances) of the stochastic variables $x$ and $y$ at different points along the process. We indicate these samples with bold symbols $\mathbf{x}_i$ and $\mathbf{y}_i$, where $i = 1 \dots 30$ indicates at point of the process the value is recorded. In other words:

1. (Without Migration - unconnected topology) We run the algorithm independently $N$ times (on different CPUs) and we record in the variable $x_i$, $i = 1 \dots 30$ the best solution found across the $N$ algorithmic runs each 2000 function evaluation
2. (With Migration - two way ring topololgy) We run the algorithm $N$ times (on different CPUs) and we record the best solution found across the $N$ algorithmic runs each 2000 function evaluation in the variable $y_i$, $i = 1 \dots 30$ when we also let solutions migrate along a two-way ring topology. Thus the index $i$ can be seen as the number of migrations occured and will so be interpreted in the following.

At the end of this process we have the samples $\mathbf{x}_i$ and $\mathbf{y}_i$, $\forall i = 1 \dots 30$ each containing 200 instances of the stochastic variables $x_i$ and $y_i$.

We first detect if there is any statistical difference in the samples for $i = 30$ (i.e. after the very last migration), in case there is none we set $i = i - 1$ and repeat the test until a difference is found. When a dfference is detected at $i = m$ we sort the solutions in $\mathbf{x}_m$ and $\mathbf{y}_m$ and compare the best 60. If they all but one are better in one sample we conclude that that solution strategy is better, otherwise we try again with $i = i - 1$. If we get to $i = 1$ without having found any result we conclude that there is not enough statistical evidence to choose between the two solution strategies for the particular couple $< \mathscr{P}, \mathscr{A} >$ under consideration.

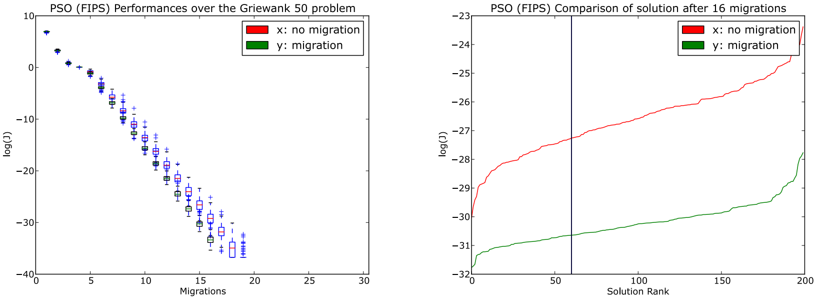


**Fig. 2** Comparison in the case of PSO (FIPS) applied to Griewank 50

**Example:** Assume the problem $\mathscr{P}$ is Griewank 50 and the algorithm $\mathscr{A}$ is a 100 generation Fully Informed Particle Swarm algorithm (PSO FIPS) with population size of 20. In Fig. 2, on the left, we show, after each of the 30 migrations, the box plot in logarthmic scale of the samples $\mathbf{x}_i$ and $\mathbf{y}_i$. Clearly at $i = 30$ there is no difference between the results, as in both cases the global optimum of the problem (i.e. J=0) has been found. Proceeding backwards, we then consider $i = 16$, where there actually is a statistical difference between the two samples. At that stage of the runs, we show in Fig. 2, on the right, the ranked solutions. As 59 out of the best 60 solutions of the sample $\mathbf{x}_{16}$ are better than the best 60 solutions of sample $\mathbf{y}_{16}$ we conclude that for this pair $< \mathscr{P}, \mathscr{A} >$ migration is actually better.

NOTE: in this example the comparison is quite trivial and a visual inspection of the results is enough to conclude, however there are cases where a mathematical formalization of the comparison is necessary to decide conclusively whether migration helped or not.

In the method outlined above, one must be very careful when detecting the statistical difference between the samples (in the exmaple it is rather obvious, but things do get much more complicated). Here we use a method based on random resampling, essentially a jacknifing method [7]. The use of this method is not too common within the global optimization community and we thus briefly explain it in the following. We create a new sample $\mathbf{r}$ joining $\mathbf{x}$ and $\mathbf{y}$. We then randomly create two disjoint new samples of size equal to the original $\mathbf{x}$, $\mathbf{y}$ out of $\mathbf{r}$ and we evaluate, for these two samples, the difference $d$ in a chosen statistics (in our case the mean). We repeat this process $n = 10000$ times, thus building an experimantal distribution of $d$ for two samples of equal size extracted at random from $\mathbf{r}$. We then evaluate, according to the built distribution, the probability of $d$ being as big as the one calcualted from the two original samples $\mathbf{x}$ and $\mathbf{y}$ and we use this as our confidence level that the two distribution actually are different. Only if this is larger than 99.97% we conclude that they indeed are. Note that if $p = 0.9997$ approximate the probability that the two samples come from different random processes, our actual confidence level that the whole optimization process is different is smaller as we need to compare the samples 30 times (after each migration). Thus, our true confidence level will be $p^{30} = 0.99$.

### 4.3.1 Results

We apply the methodology described above to approach “the migration dilemma” for each possible problem-algorithm couple $< \mathscr{P}, \mathscr{A} >$ one can form starting from the algorithms and problems introduced. We repeat the same experiment using $N = 8$ and $N = 32$ islands. This results in a total of 240 experiments. Simple calculation show that for each one of those where $N = 8$, $24,000,000$ objective function evaluations are performed, while for $N = 32$ the number of function evaluation per experiment is $96,000,000$ (we report these number with the sole purpose of giving a feel for the amount of computations involved in our tests). These allow the construction of our statistical samples $\mathbf{x}_i$ and $\mathbf{y}_i$. Making use of the inherent parallelization offered by PaGMO, the 240 experiments take roughly 96 hours to complete on a linux gentoo system installed on top of a dual quad core OSX XServe machine.

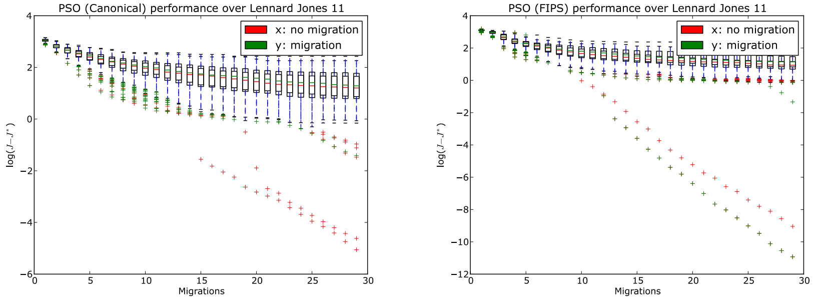


**Fig. 3** Two non obvious cases

The results are summarized in Table 1 where we report, for each experiment, the outcome of our comparison criteria. In reporting only this final information we are synthezising in a single table a very complex procedure producing a great amount of data, thus many of the details on what is actually happening in a case to case basis are inevitabley lost. As an example, we here only investigate in more detail what happens in the case of $N = 8$ for the pairs, Lennard-Jones 11, PSO (canonical) and PSO (FIPS). This case is interesting as our comparison concludes that migrating is actually harmful in the case of PSO, (FIPS) and undecidable in the case of the PSO (canonical), a strange conclusion that is worth further investigation. A closer look to the data reveal more details on the rationale for such a conclusion. In Figure 3 we report the boxplots for $\mathbf{x}_i$ and $\mathbf{y}_i$. In the case of PSO (canonical) we observe how after some migrations one lucky trial, not using migration, finds the optimal solution and is shown as an oulier (or flier) in the boxplot. For this particular experiment, this fact and the generally lower median of the sample relative to the optimization without migration (see the median red line in the boxplot) is not considered as no statistical differenc is found between the samples at any of the migration steps. For this reason our comparison criteria does not choose among the two approaches. Looking at the other case, the one considering the PSO (FIPS) algorithm, we note from Figure 3 that even though a lucky shot finds the optimal solution in the case of migration (shown as an outlier, or flier, in the boxplot), our comparison criteria concludes that not migrating is actually better for this case and gives to the unconnected topology the “winner” cup. This clearly implies that there is statistical significance between the samples at some point during the optimization and that, except that lucky run, the other 59 best solutions found in the case of the unconnected topology are actually better than the ones found by migrating (as evaluated at the point where statistical difference is found between the samples). Other cases are much easier to judge (as shown for example in Figure (2) and here we picked up one of the most troublesome cases in order to show how our comparison criteria is able to make intelligent choices also in difficult situations where, probably, also humans would argue on what conclusion to reach.

**Table 1** Generalized Island Model Results: migration occurs every 2000 function evaluations, algorithms are stopped after 30 migrations, samples are made of 200 instances. M = Migration outperforms non migration. U = Non migration outperforms migration, - = no conclusion is possible as the results are not significantly different

| | PSO (Canonical) | PSO (FIPS) | Corana's SA | HS | HS (Improved) | GA (Gaussian) | GA (Random) | DE (rand/1/exp) | DE (rand/1/bin) | Artificial Bee Colony |
|---|---|---|---|---|---|---|---|---|---|---|
| Griewank ($d = 50$) | M | M | M | M | M | M | M | M | M | M |
| Rastrigin ($d = 50$) | - | U | M | M | M | M | M | M | M | M |
| Rosenbrock ($d = 50$) | U | M | M | M | M | M | M | M | M | M |
| Ackley ($d = 50$) | M | M | M | M | M | M | M | M | M | M |
| De Jong ($d = 50$) | M | M | M | M | M | M | M | M | M | M |
| Levy 5 ($d = 50$) | M | U | M | M | M | M | M | M | M | M |
| Schwefel ($d = 50$) | M | U | M | M | M | M | M | M | M | M |
| Lennard-Jones (11 Atoms) | M | U | M | M | M | M | M | M | M | M |
| Lennard-Jones (17 Atoms) | - | U | M | M | M | M | M | M | M | M |
| Rosetta | - | - | - | M | M | M | - | M | M | M |
| Cassini 2 | - | U | - | M | - | M | - | M | M | M |
| Messenger Full | - | U | - | M | U | M | M | M | M | M |

8 Islands

| | PSO (Canonical) | PSO (FIPS) | Corana's SA | HS | HS (Improved) | GA (Gaussian) | GA (Random) | DE (rand/1/exp) | DE (rand/1/bin) | Artificial Bee Colony |
|---|---|---|---|---|---|---|---|---|---|---|
| Griewank ($d = 50$) | M | M | M | M | M | M | M | M | M | M |
| Rastrigin ($d = 50$) | M | M | M | M | M | M | M | M | M | M |
| Rosenbrock ($d = 50$) | U | U | M | M | M | M | M | M | M | M |
| Ackley ($d = 50$) | M | M | M | M | M | M | M | M | M | M |
| De Jong ($d = 50$) | M | M | M | M | M | M | M | M | M | M |
| Levy 5 ($d = 50$) | M | U | M | M | M | M | M | M | M | M |
| Schwefel ($d = 50$) | M | U | M | M | M | M | M | M | M | M |
| Lennard-Jones (11 Atoms) | M | U | M | M | U | M | M | M | M | M |
| Lennard-Jones (17 Atoms) | M | U | M | M | M | M | M | M | M | M |
| Rosetta | - | - | - | M | M | M | M | M | M | M |
| Cassini 2 | - | U | - | U | M | M | M | M | M | M |
| Messenger Full | - | U | M | M | U | M | M | M | M | M |

32 Islands

Let us now take a look at Table 1. We may observe a number of things that appear quite strongly from the reported data. We list them in the following as they all are observation worth further studies.

- Not surprisingly (the no free lunch theorem applys here) the answer to the question "should I migrate or not?" is "it depends from the algorithms and the problem".
- As a general rule-of-thumb, our generalized migration operator help algorithms find better solutions.
- PSO (and in particular for its FIPS variant) is an exception to the above rule being unable to take consistently advantage of our generalized migration operator.
- Problems such as Griewank, Ackley and De Jong are solved more efficiently making use of migration, regardless of the employed algorithms.
- Increasing the problem complexity, it is more difficult to find statistically significant results using a sample size of 200, as shown by the higher number of inconclusive tests for problems such as Lennar-Jones, Rosetta, Messenger Full and Cassini.
- Increasing the number of islands the conclusions do not change. Statistical significance is easier to be found in the comparisons made using a higher number of islands.

**Table 2** Heterogeneous tests results: parameters as in the previous experiment. Number in every cell reports how many of the 5 heterogeneous archipelagos performed better than best of the two homogeneous algorithms from experiment 1 (with or without migration) for the given problem and algorithm.

| | PSO (Canonical) | PSO (FIPS) | Corana's SA | HS | HS (Improved) | GA (Gaussian) | GA (Random) | DE (rand/1/exp) | DE (rand/1/bin) | Artificial Bee Colony |
|---|---|---|---|---|---|---|---|---|---|---|
| Griewank ($d = 50$) | 4 | 0 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| Rastrigin ($d = 50$) | 5 | 5 | 4 | 5 | 5 | 5 | 5 | 4 | 5 | 5 |
| Rosenbrock ($d = 50$) | 1 | 3 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 1 |
| Ackley ($d = 50$) | 4 | 0 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| De Jong ($d = 50$) | 4 | 0 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 4 |
| Levy 5 ($d = 50$) | 5 | 5 | 0 | 5 | 5 | 4 | 5 | 0 | 5 | 5 |
| Schwefel ($d = 50$) | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| Lennard-Jones (11 Atoms) | 5 | 5 | 2 | 5 | 5 | 5 | 5 | 1 | 1 | 5 |
| Lennard-Jones (17 Atoms) | 5 | 5 | 0 | 5 | 5 | 5 | 5 | 0 | 5 | 5 |
| Rosetta | 2 | 0 | 5 | 5 | 5 | 5 | 5 | 0 | 0 | 5 |
| Cassini 2 | 2 | 5 | 5 | 5 | 0 | 5 | 5 | 0 | 3 | 5 |
| Messenger Full | 0 | 0 | 5 | 5 | 4 | 5 | 5 | 0 | 0 | 5 |

8 Islands

| | PSO (Canonical) | PSO (FIPS) | Corana's SA | HS | HS (Improved) | GA (Gaussian) | GA (Random) | DE (rand/1/exp) | DE (rand/1/bin) | Artificial Bee Colony |
|---|---|---|---|---|---|---|---|---|---|---|
| Griewank ($d = 50$) | 0 | 0 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| Rastrigin ($d = 50$) | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| Rosenbrock ($d = 50$) | 0 | 5 | 2 | 5 | 5 | 5 | 5 | 5 | 5 | 0 |
| Ackley ($d = 50$) | 0 | 0 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| De Jong ($d = 50$) | 0 | 0 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| Levy 5 ($d = 50$) | 5 | 5 | 2 | 5 | 5 | 5 | 5 | 3 | 5 | 5 |
| Schwefel ($d = 50$) | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| Lennard-Jones (11 Atoms) | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 0 | 1 | 5 |
| Lennard-Jones (17 Atoms) | 5 | 5 | 0 | 5 | 5 | 5 | 5 | 0 | 5 | 5 |
| Rosetta | 5 | 3 | 5 | 5 | 5 | 5 | 5 | 0 | 0 | 5 |
| Cassini 2 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 0 | 5 | 5 |
| Messenger Full | 0 | 5 | 5 | 5 | 5 | 5 | 5 | 0 | 0 | 5 |

32 Islands

## 4.4 Experiment #2: Can They Cooperate?

In the previous section we have established that the influx of migrants help populations in islands to converge faster to good solutions for the majority of the evolution startegies tried. We now ask ourself the question: does it matter whether these migrants come from populations being evolved with the same paradigm? In other words: does it help to evolve populations using different algorithms while still exchanging migrants among them? In order to answer this question we compared the results obtained in the previous section (where we used archipelagos of 8 and 32 homogeneus islands, i.e. islands containing the very smae algorithm) with the result obtained using archipelagos of 8 and 32 island containing heterogeneous algorithms. As we have determined that PSO is, as an exception, not taking advantage of our generalized migration operator, we do not allow migrants to islands containing PSO, only from. Different heterogeneous archipelagos can be created out of mixing the 10 different algorithm instances considered, thus we perform our comparison with respect to five archipelagos containing different permutations of the chosen algorithms (a simple round robin startegy is implemented to select the algorithms to instantiate in the different islands). These are then compared pairwise (see the previous section for the comparison criteria) to the corresponding homogeneous archipelago. In Table 2 we report for each pair problem-algorithm the number of heterogeneous archipelagos that performed better than the corresponding homogeneous ones (using migration or not according to the best choice outlined in Table 1).

One would expect that the heterogeneous archipelago performs either a) as good as or b) worse than an homogeneous archipelago having the best algorithm across all islands. This is infact the case, as an example, for the problem Griewank. In this case PSO FIPS is the best performing algorithm and no heterogeneous archipelago (also the ones containing islands with PSO FIPS) can perform better. The surprising result comes from problems such as Rastrigin or Schwefel, where we can conclude that the cooperation between different algorithms via our generalized operator creates defacto a new meta-algorithm that improves over the performance of all its algorithmic components.

## 5 Conclusions

We propose a generalization of the island model to obtain a coarsed grain parallelization strategy valid across global optimization algorithms. The new model, essentially, allows for information exchange among different instances of algorithms. We show how, for algorithms such as Particle Swarm optimization, Differential Evolution, Simulated Annealing, Bee Clolony Search, Harmony Search and Genetic Algorithms such an information exchange is indeed beneficial in a large number of cases. We then test the same migration strategy across heterogeneous algorithms to study whether solutions generated by one algorithm could improve over the performance of a second algorithm and vice versa. We find that on some problems, migration among a set of heterogenous algorithms allows for a better search than all possible set ups where migration occur among different instances of the same algorithm.

## References

1. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Transactions on Evolutionary Computation 6(5), 443–462 (2002)
2. Aydin, M.E., Yiğit, V.: Parallel simulated annealing. Wiley Online Library (2005)
3. Biscani, F., Izzo, D., Yam, C.H.: A global optimisation toolbox for massively parallel engineering optimisation. In: International Conference on Astrodynamics Tools and Techniques - ICATT (2010)
4. Braun, H.: On Solving Travelling Salesman Problems by Genetic Algorithms. In: Schwefel, H.-P., Männer, R. (eds.) PPSN 1990. LNCS, vol. 496, pp. 129–133. Springer, Heidelberg (1991)
5. Cantú-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Norwell (2000)
6. Clerc, M., Kennedy, J.: The particle swarm explosion, stability, and convergence in a multidimensional complex space. IEEE Transactions on Evolutionary Computation 6(1), 58–73 (2002)

7. Cohen, P.R.: Empirical methods for artificial intelligence, vol. 55. MIT press (1995)
8. Corana, A., Marchesi, M., Martini, C., Ridella, S.: Minimizing multimodal functions of continuous variables with the ”simulated annealing” algorithm Corrigenda for this article is available here. ACM Transactions on Mathematical Software (TOMS) 13(3), 262–280 (1987)
9. Geem, Z.W., Kim, J.H., Loganathan, G.V.: A new heuristic optimization algorithm: Harmony search. SIMULATION: Transactions of The Society for Modeling and Simulation International 78, 60–68 (2001)
10. Goldberg, D.E.: Genetic algorithms in search, optimization, and machine learning. Addison-wesley (1989)
11. Izzo, D., Rucinski, M., Ampatzis, C.: Parallel global optimisation meta-heuristics using an asynchronous island-model. In: IEEE Congress on Evolutionary Computation, CEC 2009, pp. 2301–2308. IEEE (2009)
12. Karaboga, D., Basturk, B.: A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. Journal of Global Optimization 39(3), 459–471 (2007)
13. Kennedy, J., Eberhart, R.C.: Particle Swarm Optimization. In: Proceedings of the IEEE International Conference on Neural Networks, Perth, Australia, vol. 4, pp. 1942–1948. IEEE Press (1995)
14. Konfrst, Z.: Parallel genetic algorithms: advances, computing trends, applications and perspectives. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium 2004, p. 162. IEEE (2004)
15. Mendes, R., Kennedy, J., Neves, J.: The fully informed particle swarm: simpler, maybe better. IEEE Transactions on Evolutionary Computation 8(3), 204–210 (2004)
16. Price, K.V., Storn, R.M., Lampinen, J.A.: Differential evolution. Springer, Berlin (2005)
17. Ruciński, M., Izzo, D., Biscani, F.: On the impact of the migration topology on the Island Model. Parallel Computing 36(10-11), 555–571 (2010)
18. Schwehm, M.: Parallel population models for genetic algorithms (1996)
19. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference. MIT Press, Cambridge (1995)
20. Storn, R., Price, K.: Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. Journal of Global Optimization 11(4), 341–359 (1997)
21. Tanese, R.: Distributed genetic algorithms. In: Proceedings of the 3rd International Conference on Genetic Algorithms, pp. 434–439. Morgan Kaufmann Publishers Inc., San Francisco (1989)
22. Vinkó, T., Izzo, D.: Global optimisation heuristics and test problems for preliminary spacecraft trajectory design. Technical Report GOHTPPSTD, European Space Agency, the Advanced Concepts Team (2008)
23. Wales, D., Doye, J.: Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms. Arxiv preprint cond-mat/9803344 (1998)

# Evolutionary Associative Memories through Genetic Programming

Juan Villegas-Cortez, Gustavo Olague, Humberto Sossa, and Carlos Avilés

**Abstract.** Natural systems apply learning during the process of adaptation, as a way of developing strategies that help to succeed them in highly complex scenarios. In particular, it is said that the plans developed by natural systems are seen as a fundamental aspect in survival. Today, there is a huge interest in attempting to replicate some of their characteristics by imitating the processes of evolution and genetics in artificial systems using the very well-known ideas of evolutionary computing. For example, some models for learning adaptive process are based on the emulation of neural networks that are further evolved by the application of an evolutionary algorithm. In this work, we present the evolution of a kind of neural network that is collectible known as associative memories (AM's) and which are considered as a practical tool for reaching learning tasks in pattern recognition problems. AM's are complex operators, based on simple arithmetical functions, which are used to recall patterns in terms of some input data. AM's are considered as part of artificial neural networks (ANN), mainly due to its primary conception; nevertheless, the idea inherent to their mathematical formulation provides a powerful description that helps to reach a specific goal despite the numerous changes that can happen during its operation. In this chapter, we describe the idea of building new AM's through genetic programming (GP) based on the coevolutionary paradigm. The methodology that is proposed consists in splitting the problem in two populations that are used to evolve

Juan Villegas-Cortez · Carlos Avilés
Universidad Autónoma Metropolitana - Azcapotzalco. Departamento de Electrónica. Av. San Pablo 180 Col. Reynosa, 02200. México D.F., México
e-mail: `juanvc@correo.azc.uam.mx`

Gustavo Olague
Centro de Investigación Científica y de Educación Superior de Ensenada,
CICESE. Ensenada, B.C. México
e-mail: `olague@cicese.mx`

Humberto Sossa
Centro de Investigación en Computación CIC-IPN. México D.F., México
e-mail: `hsossa@cic.ipn.mx`

F.F. de Vega et al. (Eds.): Parallel Architectures & Bioinspired Algorithms, SCI 415, pp. 171–188.
springerlink.com

simultaneously both processes of association and recall that are commonly used in AM's. Experimental results on binary and real value patterns are provided in order to illustrate the benefits of applying the paradigm of evolutionary computing to the synthesis of associative memories.

# 1 Introduction

Associative Memories (AM's) are considered as simple and useful devices that are designed to recall output patterns in terms of input patterns by means of simple operations. AM's are considered as part of artificial neural networks (ANN); but instead of using complex structures and operators, it is said that AM's are divided in two abstract processes that are created through a set of elementary operations, which are used within two processes known as association and recall.

As a preliminary introduction, let's say that during the association phase, the AM is built in terms of the patterns $X$ and $Y$, denoted as $(X,Y)^k$, with $k$ an integer. Thus, every pattern association is carried out by the application of simple operations such as: addition, multiplication, maximum, minimum, and other alike operators acting over the input patterns. In this way, an associative memory $\boldsymbol{M}$ is represented by a matrix, which is written from all the pattern associations in order to consider the whole knowledge being provided by the input patterns. The corresponding components $m_{ij}$ can be seen as the synapses of a simple neural network. On the other hand, the operator $\boldsymbol{M}$ is generated from a predefined set of finite known associations called the fundamental set; thus, such association set is represented as $\{(X^k;Y^k) \mid k=1,\ldots,p\}$, where $p$ is defined as the number of associations. Moreover, if $(X^k = Y^k)\ \forall\ k = 1,\ldots,p$, then $\boldsymbol{M}$ is considered auto-associative, otherwise it is called hetero-associative. Finally, if a distorted version of a pattern, denoted as $\widetilde{X}$, is fed up to $\boldsymbol{M}$, and the obtained output is exactly $Y^k$; then, recalling is considered as perfect.

Research on associative memory models is characterized by their simplicity that could be seen as the result of a number of great developments that have been carried out during the last 50 years; for some examples we refer the interested reader to [6],[24], [14], [15], [25], [22, 23], [31, 32]. Nevertheless, most of these models have several limitations that are major research topics; for example: their limited storage capacity, the difficulty to deal with more than one type of pattern (binary, integer or real-valued), their lack of robustness against different kind of noise (additive, subtractive, mixed, Gaussian, etc.); and above all, these models work only for a specific purpose defined during the moment where they were contrived. Moreover, all these models have been manually developed and in general the whole design process for one model could take from one up to two years of research.

The research described in this chapter from the viewpoint of genetic programming it is briefly outlined here. A first attempt to automatically generate AM's has been reported in [16]. This method was conceived for the case of bidirectional associative memories (BAM). On the other hand, the closest work to the approach presented in this chapter is concerned with the automatic design of artificial neural networks[13], which uses some kind of genetic programming (GP) for this purpose. Another reported work applies Particle Swarm Optimization [4]. Bearing in mind the relevant results obtained with GP techniques versus genetic algorithms [21], in this work we describe a GP-based methodology to automatically generate AM's. GP has been successfully applied in a huge range of areas. In particular, we can mention some related to computer vision ([1], [29], [10], [12], [27], [28], [37], [38], [26]).

In this chapter, we deal with the problem of automatically generating AM's through GP; by considering an evolutionary process. Previous results can be found in ([33], [34]) where the initial ideas described there turns into the simplicity of AM's devices. We improve our initial technique and provide a better description of the problems involved in the recall of binary and real-valued patterns. The improvement were focused on two evolutionary processes using coevolutionary mixed aspects as it is reported in [35].

This research is related to [40] that deals with the idea of evolving ANN. Nevertheless, as an alternative to performing the optimization of an ANN regarding the number of parameters and their respective topology; the idea explained in this chapter is to evolve the representation of an associative memory using the powerful paradigm of genetic programming. Moreover, it is important to mention that this work is naturally extended in to the generally accepted perspective of parallel and distributed computing [39]. Thus, we can talk without loss-of-generality of distributed associative memories (DAM's) instead of associative memories. The reason is due to the fact that all input data can be manipulated as separate chunks of information and the whole system could be implemented using a parallel and distributed architecture.

The rest of the chapter is organized as follows, and is oriented towards the explanation and construction of an evolutionary methodology. In section 2, we provide a brief description of our first approach fully described in [35]. In section 3, we present our co-evolutionary based-GP methodology for the automatic generation of AM through GP. In section 4, we show the experimental results for some representative databases, composed of elemental and complex pattern sets. Finally, section 5 presents the conclusions and suggestions for future research.

## 2 Automated Design of AM's through GP

The proposed model for generating AM's using evolutionary algorithms is based on the following two-stage process. First, evolution starts as a simple ANN with a pre-established topology that is evolved from more than one connection [40]. Second, the parallel architecture is evolved with the aim of generating different topological structures. Common AM models uses a connection structure that is fixed, and it is said that such traditional models are confronted with several challenges by considering the full aspects of every possible situation; such as: the capacity of memory recall, as well as noise suppression. Our proposal is based on the linear associator concept introduced in [7], in such a way of considering the general association aspects between pattern sets, but bearing in mind that the local association of all parts is focused on the connectivity of every synapse that is carried out between the vector pattern components. This chapter explains the necessary steps to implement the evolution of associative memories through a coevolutionary paradigm. The basic idea is to divide the evolutionary process in two independent stages, association and recall, that are useful in the modeling and design of associative memories for pattern recognition problems.

Therefore, the proposal describes the evolutionary process by considering three steps that help to define the two stages that characterize an associative memory. First, we define the set of function operators that are used during the association stage; second, we define the set of function operators and terminals that are used for the recalling stage by considering the fundamental aspects of the association matrix previously built; and finally, we implement the co-evolutionary approach, similar to [2], by joining the two evolved individuals in order to create a single one.

## 3 A Coevolutionary Model for the Design of AM's

In this way, our proposed GP based co-evolutionary model consists of two populations; one for association and one for recalling. The idea of managing both processes through two separated populations has helped us to achieve better solutions in contrast to our first study, where the solution consists of a single evolutionary process [33]. In this chapter, we explain how it was considered during the design two sequential processes instead of only one, through the cooperative co-evolutionary paradigm. This new strategy is more adequate to the aim of reaching our goal of developing a simulated correlation between both processes. Thus, each population represents a separate part of the problem, and the coevolutionary process is in charge of solving the whole pattern recognition problem. Hence, the solution is achieved through the coevolutionary process, in which two separate species or populations are jointly evolved in such a way that every species cooperates to attain a global solution that satisfy both stages at the same time. Note, that the proposed model can be naturally extended into a distributed and parallel system, since our two population model can be accessed concurrently. In this case, the DAM's could allow a straightforward implementation of the proposed coevolutionary model by carrying out the associative memory paradigm for parallel and distributed computing. Hence, every individual is a possible solution for the problem in the pre-established environment. The basic idea of coevolution consists in the concept of divide and conquer, that is used to conceptualize the problem (system) as if it was adapted or conformed from many sub-problems (sub-systems), which are integrated by evolving them separately and later combining them into a single solution; i.e., the original sought solution. In other words, all sub-systems are finally joined by implementing a strategy where the entire solution is built from partial components [11].

Figure 1 shows the framework for implementing the cooperative co-evolutionary process. Here, the AM's that are representing suitable solutions to the problem at

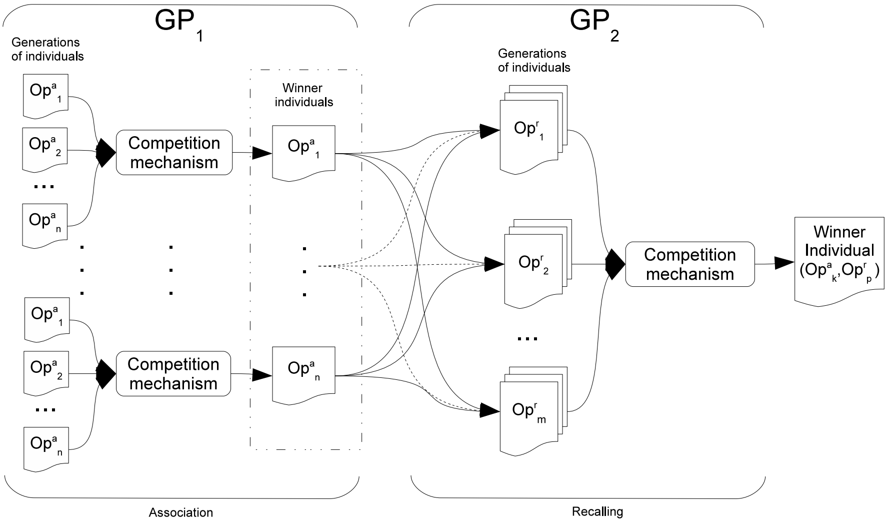


**Fig. 1** This figure shows the conceptual framework of our cooperative coevolutionary process.

hand, work on two complementary search spaces with the aim of fitting the pattern association with the better recall. Thus, the first process, $GP_1$, could be seen as a clustered process working on the association stage that is producing a number of individuals with the goal of identifying the best individual at each cluster and the result is considered as the winner of the evolutionary process. Then, the second process, $GP_2$, known as the recalling stage gets as input all winner individuals from the previous stage to participate in the solution of the second process by recombining their learned features. This process is iterated until a termination criteria is satisfied, and the whole process is said to be completed, and we finally get an individual global winner.

Some important aspects related to the fitness function is the global criteria where the whole process is measured as the minimum of all individual pairs up to the second phase. In this way, the components of our new model are defined in terms of both, function and terminal, sets per every population for each GP evolutionary process; see Figure 2. Hence, the co-evolutionary model is composed of the following aspects:

1. $Op^a_k$ represents the evolutionary operator being applied in each pattern association. The representation or genotype of each individual is encoded in the traditional GP-form of a tree. The local association matrix $\mu_i$ is integrated by the $Op^a_k$ operator acting on the components of each vector of the local association $\{Y_j, X^T_j\}$.
2. $M_k$ describes the general association matrix. This is taken as the sum of all local associations and it provides the cumulative knowledge, which is inspired by the perceptron principle.

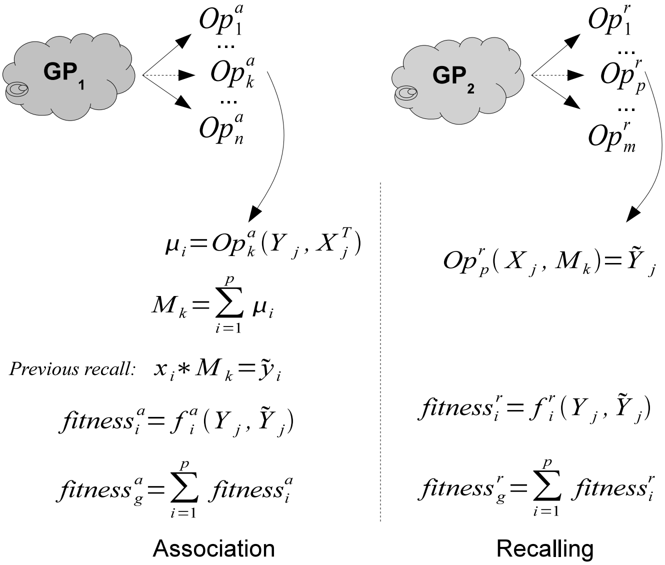

$GP_1$
$Op_1^a$
...
$Op_k^a$
...
$Op_n^a$
$GP_2$
$Op_1^r$
...
$Op_p^r$
...
$Op_m^r$
$\mu_i = Op_k^a(Y_j, X_j^T)$
$M_k = \sum_{i=1}^{p} \mu_i$
Previous recall: $x_i * M_k = \tilde{y}_i$
$fitness_i^a = f_i^a(Y_j, \tilde{Y}_j)$
$fitness_g^a = \sum_{i=1}^{p} fitness_i^a$
$Op_p^r(X_j, M_k) = \tilde{Y}_j$
$fitness_i^r = f_i^r(Y_j, \tilde{Y}_j)$
$fitness_g^r = \sum_{i=1}^{p} fitness_i^r$
Association
Recalling

**Fig. 2** This figure illustrates the proposed coevolutionary model for the GP-developmental process of AM's.

3. $T_a$ is considered as the *Terminal Set* for the association stage. It consists of a set of nodes taking the vector entries, such as $T_a = \{x_j, y_j\}$. These nodes belong to each pattern-vector$\{X\}$ and $\{Y\}$; this is implemented in order to encode a local correlation input.
4. $F_a$ is the *Function Set* for the association stage. These functions have been defined after reviewing the solutions reported in literature. This helped us to devise the set of possible structures in order to target the space with the aim of producing individuals that achieve similar performance in comparison to the reviewed AM models. In this case, $F_a = \{+, -, \wedge, \vee, \times\}$ where $\wedge, \vee$, and $\times$ are the *minimum, maximum* and *multiplication* operators
5. $Op_p^r$ is the evolutionary operator being applied during the pattern recall process. It comprises the input vector $X_j$, as well as the matrix association $M_k$ being generated by the previous evolutionary association operator.
6. $T_r$ is the *Terminal Set* being applied during the recalling stage. This is written as $T_r = \{v, R_1, R_2, \ldots, R_m, M_k\}$; where $v \in X$ is the input vector, and $R_i$ is the $i$th row-vector of $M_k$ corresponding to the $k$th-AM.
7. $F_r$ represents the *Function Set* for the recalling stage. Thus, $F_r = \{+, -, \wedge, \vee, \otimes\}$; where $\otimes$ is defined as the multiplication operator between vector components, $\otimes(X, Y) = [x_1 * y_1, x_2 * y_2, \ldots, x_n * y_n]$, and the association matrix is defined as follows $\otimes(X, M_k) = X * M_k$; hence, it satisfies the dimensionality of the multiplication operator between matrices.

8. Finally $\tilde{Y}_j$ represents the approximated $Y_j$ pattern obtained through the application of the operator rule $Op_p^r$ being applied during the recalling stage.

In this way, the fitness function that provides the measure of every process is an essential evolutionary step of the whole process. For our purpose the fitness value is applied at the end of both stages: association and recall. Therefore, the fitness function covers the sum of all local fitness, related to local associations, and the recalling set, one recalling for every pattern relation, as shown in Figure 2. For this evolutionary approach we considered the normalized correlation coefficient between the goal $(Y)$ and the source processed pattern $(\tilde{Y})$ [18]. The fitness function $f$, is known as *similarity* and it is defined in the range $(0 \leq f \leq 1)$ as follows:

$$f = \frac{Y \cdot \tilde{Y}}{\sqrt{Y \cdot Y}\sqrt{\tilde{Y} \cdot \tilde{Y}}} \tag{1}$$

where $Y$ and $\tilde{Y}$ are vectors of size $1 \times N$, and $Y \cdot \tilde{Y}$ is given by the following equation:

$$Y \cdot \tilde{Y} = \frac{1}{N}\sum_{j=1}^{N} Y(1,j) \cdot \tilde{Y}(1,j) \tag{2}$$

Thus, function $f$ tries to maximize the number of matching component between vectors $Y$ and $\tilde{Y}$. This seems to be a reasonable choice for the fitness function. Here, the optimum is found when $f = 1$ and it corresponds to the matching of all pixels. The worst case takes place for $f = 0$ implying that a single pixel does not match the pattern.

Thus, we used Equation (1) as our fitness function, which was applied in the evaluation of every generated individual $\mu_i$. In this way, for the association stage, the training between the source set $X$ and the goal set $Y$, is used to integrate the *fundamental sets*. In this work, the fitness evaluation is carried out in three steps:

1. The *association* between the pattern sets $X$ and $Y$ is performed through the first stage operator $Op_k^a$; then,
2. The *multiplication* operator is fixed for the recalling task, in order to have a first estimation of the *multiplication* operator. Thus, one recall-pattern set $\widehat{Y}$ is obtained and the computed local fitness is used to sort out the winner for the first evolutionary run, $GP_1$; and this operation is repeated for the the association block as an $n$ batch processes; see Figure 1.
3. The $n-$winner individuals from the first process are considered as input for the second $GP$ process; the recalling stage. At the end of this block, the fitness function is applied again to get the winner for every $m-$population. This step is performed as a way of computing a local estimation for each operator $Op_a^r$. Then, a pair of operators is considered as a single individual $(Op_k^a, Op_a^r)$ and the global fitness is computed for each pair of operators as the fitness value for the solution pair. The architecture of the proposed system is depicted on Figs. 1, 2 and 3.

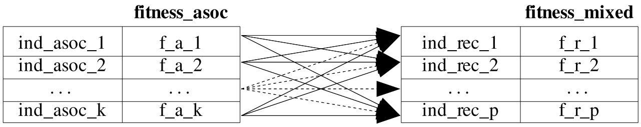


**Fig. 3** Fitness model for the GP-development of AM's.

# 4 Experimental Results and Analysis

All experiments were carried out on a workstation with a 64-bit architecture using Matlab and the GPLab toolbox ver. 3.0 [17]. The proposed coevolutionary system was programmed in two stages, association and recalling. They were performed in two batch processes $n$ and $m$ executing GP1 and GP2 respectively. In this way, the program runs for 50 generations with a population of 70 individuals and for each evolutionary process the best-so-far individual is designated as the solution to the problem. Thus, all experiments have considered GP parameters similar to those suggested in [8]; such as the crossover rate of 0.7 and 0.3 for mutation. Finally, in order to initialize the populations the method of ramped-half-and-half was used.

In this section, we present three experiments that were performed to test the efficiency of the proposed methodology. In the first example, we demonstrate the validity of the proposal with a binary-pattern problem that is known to be difficult for common AM's in the case of mixed noise. In the second experiment, we illustrate the application of our proposal for solving a classical pattern recognition problem. The third experiment, shows the results for a classification problem. In the last two examples we use well-known datasets from the associative memory literature.

## *4.1 Experiments with Binary Patterns*

In this section, a first experimental test was proposed using a set of simple vectors consisting of binary patterns and representing the digital characters 0 to 9, which were written as a matrix of size 7 × 5; see Figure 4. Thus, for these simple

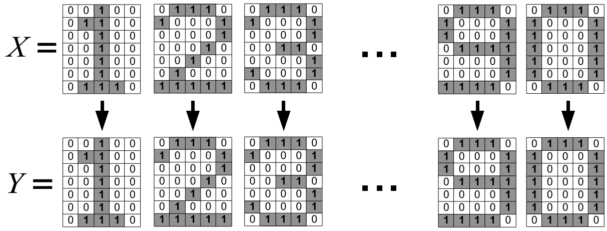
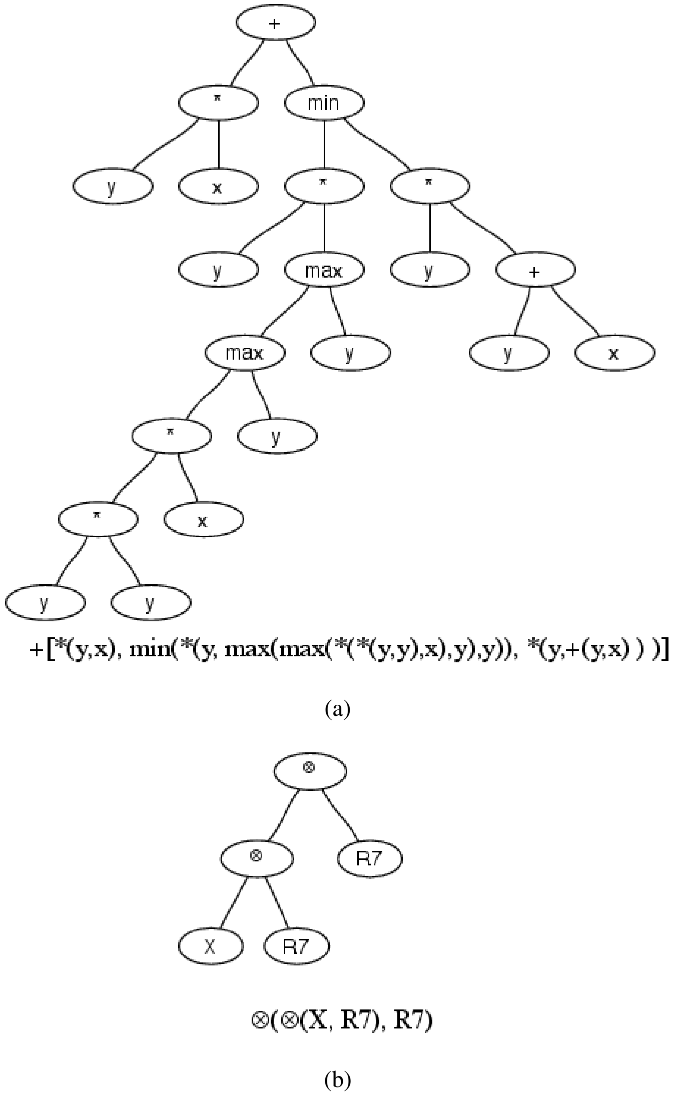


**Fig. 4** These images show the matrices of digital characters that were proposed as the first problem.



+[*(y,x), min(*(y, max(max(*(*(y,y),x),y),y)), *(y,+(y,x) ) )]

(a)

⊗(⊗(X, R7), R7)

(b)

**Fig. 5** This image depicts the evolved AM's for the auto-associative case that solves the binary pattern problem. Figure 5(a) shows the rule for pattern association; while, Figure 5(b) shows the rule for pattern recalling.

images a line-vector representation was considered to represent each matrix using an auto-associative relationship, with $X$ and $Y$ as the source and goal vector sets respectively.

As a by-product of the methodology, it is generated a set of rules for the association; as well as, another set of rules for the recalling. Hence, all these rules are coevolved with the aim of maximizing the global fitness. In this way, after evolution the selected winner is the pair of rules shown in Figure 5. Here, the AM shown in Figure 5 is but an example of a pair of AM's that successfully solves the problem. This solution was evolved through a vast number of individuals and it successfully achieves a solution that fits perfectly for this kind of pattern.

The solution depicted in Figure 5 shows that an important describing feature for this particular pattern set, is row number 7, which belongs to the association matrix $M_k$. It is the only significant one during recalling regarding the auto-associative relationship. As can be seen this solution is generated by the corresponding association rule 5(a).

In order to verify the quality of the evolved AM's of Figure 5, a new test was carried out in which the digital characters were added different levels of mixed noise, salt and pepper, in order to evaluate the AM's robustness. Thus, the mixed noise was increased from 10% to 100 % in steps of 10%; see Figure 6. These noisy images were generated using Matlab. Remarkably, we got perfect recall: zero error; hence, the GP-generated AM outperformed other human-designed memories such as the morphological and alpha-beta models; see [15] and[36].

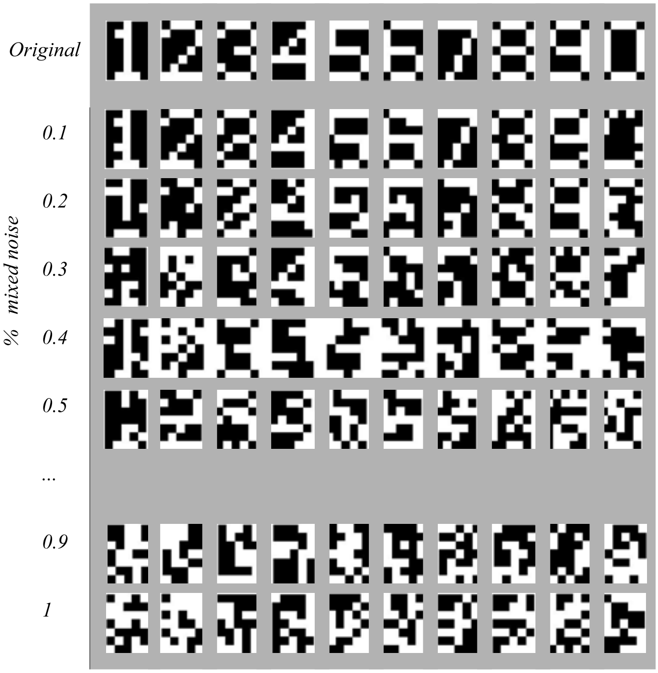
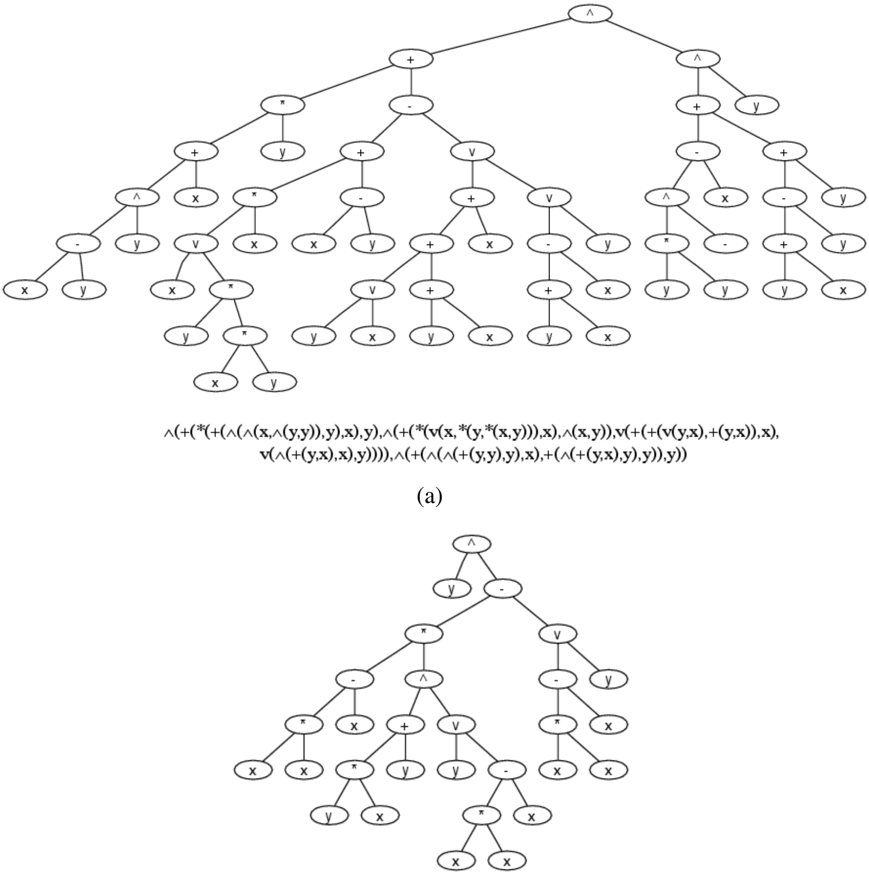


**Fig. 6** Digital characters represented as matrices. The first row shows the digital characters without noise and the following rows from top to bottom shows the resulting images after adding some mixed noise.



∧(+(*(+(∧(∧(x,∧(y,y)),y),x),y),∧(+(*(v(x,*(y,*(x,y))),x),∧(x,y)),v(+(+(v(y,x),+(y,x)),x),
v(∧(+(y,x),x),y)))),∧(+(∧(∧(+(y,y),y),x),+(∧(+(y,x),y),y)),y))

(a)

∧[y, (x*x -x)*∧{v(x*x -x, y), y*x +y } - v(x*x -x, y)]

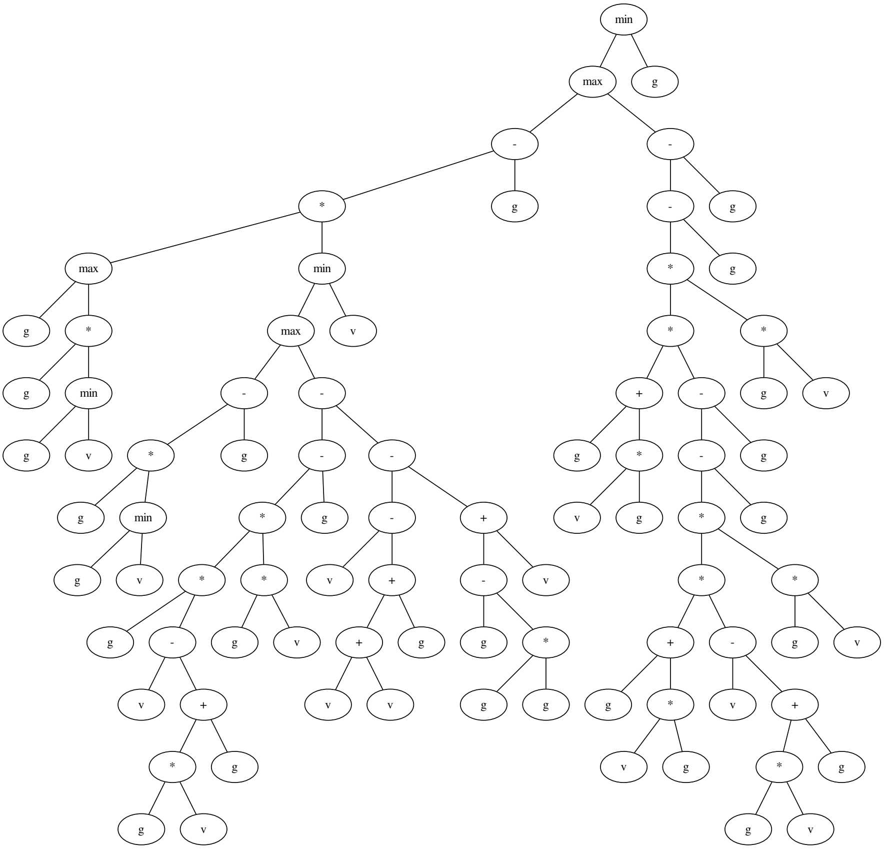
(b)

**Fig. 7** This figure shows operators $Op_1^a$ and $Op_2^a$ used as association rules; where $x$ and $y$ are the input and target patterns.

## 4.2 Experiments on the Iris-Plant Database

The results achieved during the first experiment gave us confidence for testing the proposed methodology in a more complex association problem. Here, the test consists on finding the AM's for the Iris-Plant database that provides a test from a real-world application. In particular, real value patterns should be associated in such a way of solving a problem in a bigger and more complex search space.

The proposed methodology was applied to a very well-known database problem, the Iris-Plant classification database [30], that consists of 4 features, 3 classes and 150 instances. The final AM's that achieve the highest score on this problem

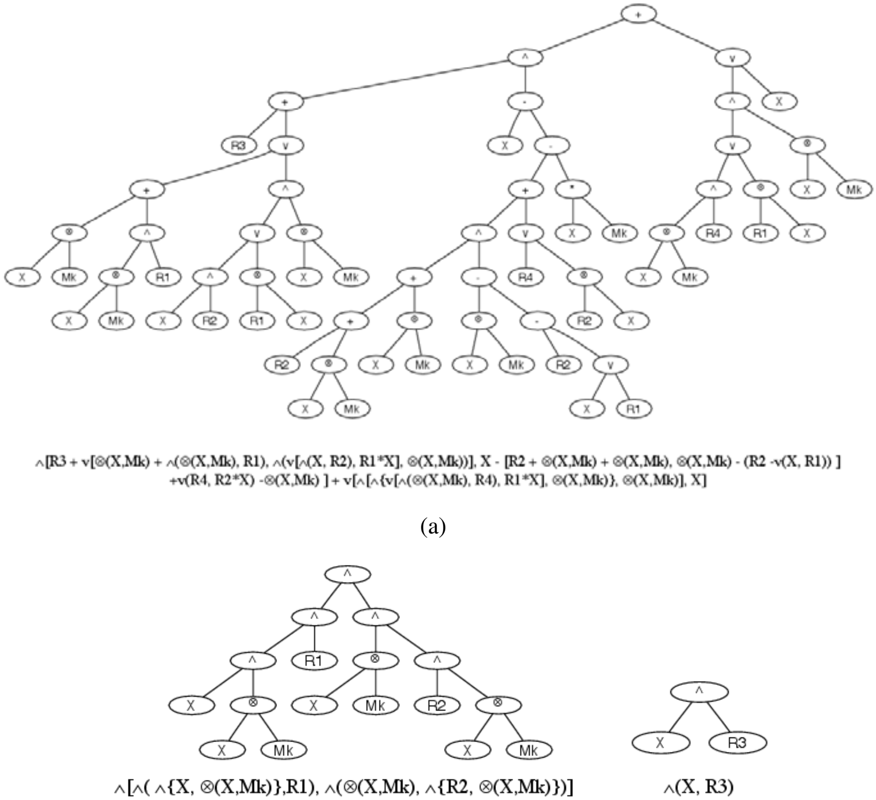


min(max(-(*(max(g,*(g,min(g,v))),min(max(-(*(g,min(g,v)),g),-(-(*(*(g,-(v,+(*(g,v),g))),
*(g,v)),g),-(-(v,+(+(v,v),g)),+(-(g,*(g,g)),v)))),v)),g),-(-(*(*(+(g,*(v,g)),-(-(*(*(+(g,*(v,g)),
-(v,+(*(g,v),g))),*(g,v)),g),g)),*(g,v)),g),g)),g)

**Fig. 8** This figure shows operator $Op_3^a$ used as the association rule for the AM in the Iris-Plant database.

are shown in Figs. 7 and 9. All the experiments were implemented for the auto-associative case. In particular, the model was implemented as a search process to find three-individuals to be used as the association rules; where each class is associated through an individual. In this way, the coevolutionary process was executed in order to find three individuals that recall the patterns. Each individual was coevolved during the association stage following the proposed cooperative coevolutionary model, until a suitable solution is achieved. Thus, each individual was labeled in the evolutionary process in order to track the global fitness for every winner pair as shown in Table 1.

Note that according to Figures 7, 8, 9, and Table 1, the evolutionary process is capable of producing different solutions; like a complex one, see Figure 8, to a simple recalling run, as shown in Figure 9(c). According to Table 1, the association


∧[R3 + v[⊗(X,Mk) + ∧(⊗(X,Mk), R1), ∧(v[∧(X, R2), R1*X], ⊗(X,Mk))], X - [R2 + ⊗(X,Mk) + ⊗(X,Mk), ⊗(X,Mk) - (R2 -v(X, R1)) ]
+v(R4, R2*X) -⊗(X,Mk) ] + v[∧[∧{v[∧(⊗(X,Mk), R4), R1*X], ⊗(X,Mk)}, ⊗(X,Mk)], X]
∧[∧( ∧{X, ⊗(X,Mk)},R1), ∧(⊗(X,Mk), ∧{R2, ⊗(X,Mk)})]
∧(X, R3)

(a)

(b) (c)

**Fig. 9** Evolved recall operators to be used in the recall stage on the AM's for the Iris-Plant database; where $M_k$ is the association matrix.

**Table 1** Fittest individuals pairing the association rules, first column in competition, with its corresponding recalling rule, second column, and the global fitness attained in cooperation. The indexed operators in the first column are related to each association rule. Operator $Op_1^a$ is shown on Figure 7(a); also, $Op_2^a$ and $Op_3^a$ can be seen in Figures 7(b) and 8, respectively. Operator $Op_1^r$, $Op_2^r$, and $Op_3^r$ in the second column is depicted in Figure 9(a).

| Association rule | Recalling rule | Global Fitness |
|---|---|---|
| $Competition(Op_1^a, Op_2^a, Op_3^a) := Op_1^a$ | $Op_1^r$ | 1 |
| $Competition(Op_1^a, Op_2^a, Op_3^a) := Op_1^a$ | $Op_2^r$ | 1 |
| $Competition(Op_1^a, Op_2^a, Op_3^a) := Op_1^a$ | $Op_3^r$ | 1 |

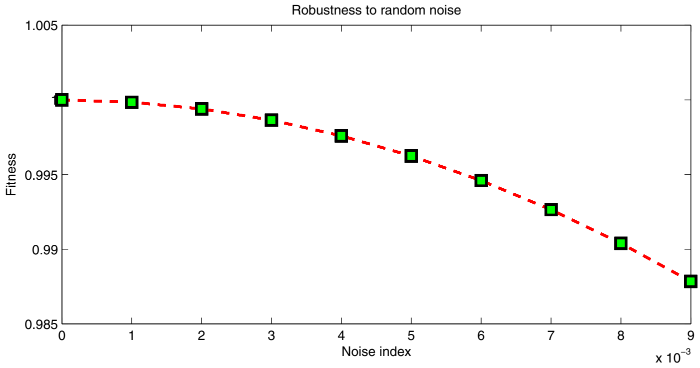


**Fig. 10** Robustness to noisy patterns for the three AM pairs.

rules, see Figures 7(b) and 8, were not selected for the association stage due to the low performance in cooperation with the recalling rules. In this way, all the three-evolutionary rules obtained for the recalling stage achieved the higher fitness in cooperation with the first association rule $Op_1^a$, see Figure 7(a).

On the other hand, the proposed methodology reveals the application of different synapsis or connections in the solution of the recalling stage. In this example, the best recalling rule shows that all rows of matrix $M_k$ are significant for the recalling stage. In particular, the rows are related to the sepal length and width; as well as, the petal length and width. In this way, all four features are relevant to the recall process for the Iris Plant database [30].

Finally, we tested this new AM by adding some random noise to the input pattern set $X$. Here, the samples were affected by a small amount of noise given in the percentage range $[0.01, 0.09]$. The resulting fitness is depicted in Figure 10 showing that the recalling rate decreases as the noise increases.

## *4.3 Experiments on the Pima Indians Database*

The third experiment analyses the dataset of the Pima Indians diabetes problem [30] from a classification standpoint. In this way, 8 attributes are considered as multivariate characteristics using 768 instances. The resulting AM's suitable for this problem consider real and integer valued patterns that are show in Figure 11(a). In this section, all experiments were implemented for the auto-associative case.

Thus, the proposed model searches for a suited individual to be used as an association rule and another one for the recalling rule. The resulting trees are shown in Figure 11. Note that the recalling rule of Figure 11(b) uses the rows R1, R2 and R3 of the association matrix; while, Figure 11(c) is a simpler rule but uses the whole matrix.

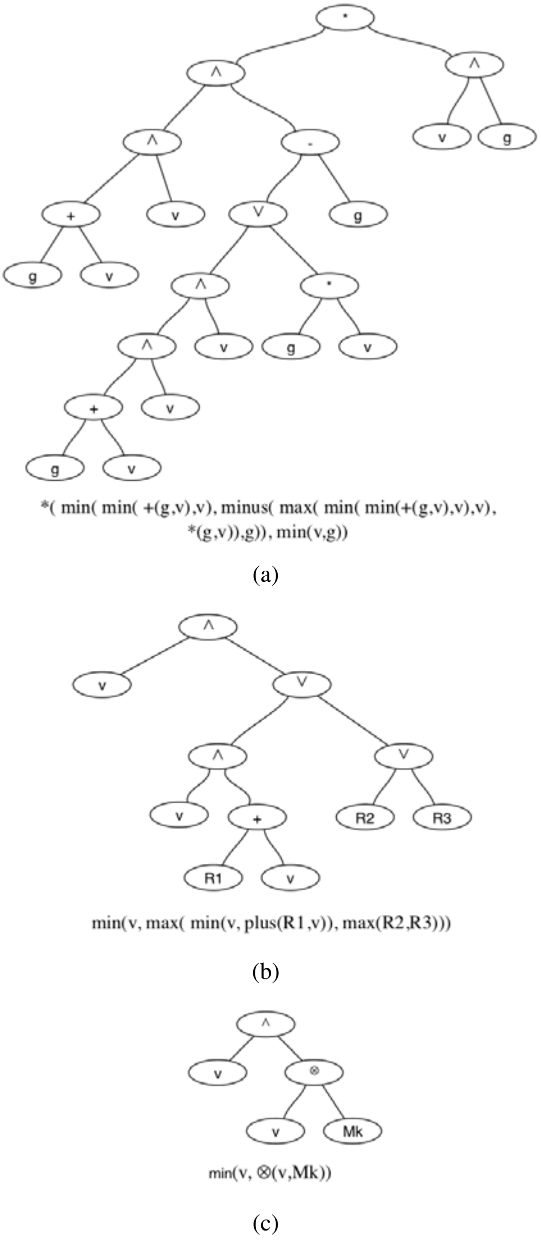

*( min( min( +(g,v),v), minus( max( min( min(+(g,v),v),v),
*(g,v)),g)), min(v,g))
(a)
min(v, max( min(v, plus(R1,v)), max(R2,R3)))
(b)
min(v, ⊗(v,Mk))
(c)

**Fig. 11** This figure shows the best individuals found for the Pima Indian diabetes dataset. The first tree $(a)$ is the operator used as the association rule, then, $(b)$ and $(c)$ are the two best individuals found for the recall stage. Finally, $M_k$ is the association matrix.

## 5 Conclusions

In this chapter, we described our GP-based approach for the synthesis of AM's. In particular, the proposed methodology is based on the coevolutionary paradigm that is used as a way of managing the association and recalling stages that are an integral part of classical AM's. Nevertheless, the idea of applying GP to the design of AM's opens us to the possibility of automating the entire design process; in such a way, of developing AM's in just a few hours instead of the extensive work that is normally applied by human experts. Moreover, the proposed methodology provides a way of creating novel AM's with an standardize procedure. Also, in this chapter it was extensively described the inherent problems that were solved to achieve our final goal [33]. Moreover, the new methodology was tested on three different datasets and the results obtained in the form of several AM's that perfectly fit each pattern set. Here, the time necessary to generate a solution varies from hours to days instead of the traditional approach that takes years of research performed by experts. In this way, the approach depends on the computational effort that is necessary to synthesize about ten or more models. This methodology is similar to the evolution of ANN using genetic algorithms. Thus, our methodology has one very important advantage: the possibility of developing AM's that are specially designed for specific pattern sets. The resulting new AM's are produced by a cost-effective process based on GP that searches for optimal solutions. In particular, these can be implemented in parallel in order to compute several solutions for each pattern recognition problem using a lower computational time. For future research, we are working towards improving our methodology through the use of new fitness functions and metric spaces suited to the kind of pattern recognition problems in challenging real world applications.

**Acknowledgements.** The Authors thank SIP-IPN for the economic support under grant number 20100468. This chapter has been prepared with financial support of the European Commission, the European Union, and CONACyT under grant FONCICYT 93829. The content of this work is an exclusive responsibility of the authors and it cannot be considered as if it could reflect the position of the European Union. We thank also the reviewers for their comments that greatly improve our work.

## References

1. Cagnoni, S., Lutton, E., Olague, G.: Genetic and Evolutionary Computation for Image Processing and Analysis. EURASIP Book Series on Signal Processing and Communications, vol. 8. Hindawi Publishing Corporation (2008)
2. Goh, C.-K., Tan, K.C.: A Competitive-Cooperative Coevolutionary Paradigm for Dynamic Multiobjective Optimization. IEEE Transactions on Evolutionary Computation 13(1), 103–127 (2009)
3. Barricelli, N.: Esempi numerici di processi di evoluzione. Methodos, 45–68 (1954)
4. Garro, B.A., Sossa, H., Vazquez, R.A.: Design of Artificial Neural Networks using a Modified Particle Swarm Optimization Algorithm. In: International Joint Conference on Neural Networks (IJCNN 2009), Atlanta, GE, USA, pp. 938–945 (2009)

5. Holland, J.H.: Adaptation in natural and artificial systems. University of Michigan Press (1975)
6. Hopfield, J.J.: Neural networks and physical systems with emergent collective computational abilities. Proceedings of the National Academy of Sciences 79, 2554–2558 (1982)
7. Kohonen, T.: Correlation Matrix Memories. IEEE Transactions on Computers C-21, 353–359 (1972)
8. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
9. Forsyth, R.: BEAGLE A Darwinian Approach to Pattern Recognition. Kybernetes 10, 159–166 (1981)
10. Hernández, B., Olague, G., Hammoud, R., Trujillo, L., Romero, E.: Visual learning of texture descriptors for facial expression recognition in thermal imagery. Comput. Vis. Image Underst. 106, 258–269 (2007)
11. Potter, M.A., De Jong, K.A.: Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents. Evolutionary Computation 8(1), 1–29 (2000)
12. Perez, C.B., Olague, G.: Learning Invariant Region Descriptor Operators with Genetic Programming and the F-Measure. In: International Conference on Pattern Recognition, ICPR (2008)
13. Rivero, D., Rabuñal, J., Dorado, J., Pazos, A.: Automatic Design of ANNs by Means of GP for Data Mining Tasks: Iris Flower Classification Problem. In: Beliczynski, B., Dzielinski, A., Iwanowski, M., Ribeiro, B. (eds.) ICANNGA 2007, Part I LNCS, vol. 4431, pp. 276–285. Springer, Heidelberg (2007)
14. Ritter, G.X., et al.: Morphological associative memories. IEEE Transactions on Neural Networks 9(2), 281–293 (1998)
15. Ritter, G.X., Urcid, G., et al.: Reconstruction of patterns from noisy inputs using morphological associative memories. International Journal of Mathematical Imaging and Vision 19(2), 95–111 (2003)
16. Shi, G.: Genetic approach to the design of bidirectional associative memory. International Journal of Systems Science 28(2), 133–140 (1997)
17. Silva, S., Almeida, J.: GPLAB - A Genetic Programming Toolbox for MATLAB. In: Gregersen, L. (ed.) Proceedings of the Nordic MATLAB Conference (NMC- 2003), pp. 273–278 (2003)
18. Quintana, M., Poli, R., Claridge, E.: Morphological Algorithm Design for Binary Images Using Genetic Programming. Genetic Programming and Evolvable Machines 7(1), 81–102 (2006)
19. Rechenberg, I.: Evolutionsstrategie Optimierung technischer Systeme nach Prinzipien der biologischen Evolution (PhD thesis). Reprinted by Fromman-Holzboog, Berlin, Germany (1973)
20. Silva Lavalle, A.R.: Un Método de Algoritmos Genéticos para Optimización de Memorias Asociativas Morfológicas. Tésis, Univesidad de Puerto Rico (2006)
21. Seo, K., Hyun, S.: A Comparative Study Between Genetic Algorithm and Genetic Programming Based Gait Generation Methods for Quadruped Robots. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcazar, A.I., Goh, C.-K., Merelo, J.J., Neri, F., Preuß, M., Togelius, J., Yannakakis, G.N. (eds.) EvoApplicatons 2010. LNCS, vol. 6024, pp. 352–360. Springer, Heidelberg (2010)
22. Sossa, H., Barrón, R., Vázquez, R.A.: New Associative Memories to Recall Real-Valued Patterns. In: Sanfeliu, A., Martínez Trinidad, J.F., Carrasco Ochoa, J.A. (eds.) CIARP 2004. LNCS, vol. 3287, pp. 195–202. Springer, Heidelberg (2004)
23. Sossa, H., Barrón, R.: Extended $\alpha\beta$ associative memories. Revista Mexicana de Física 53(1), 10–20 (2007)

24. Steinbuch, K.: Die Lernmatrix. Biological Cybernetics 1(1), 36–45 (1961)
25. Sussner, P.: Generalizing operations of binary auto-associative morphological memories using fuzzy set theory. Journal of Mathematical Imaging and Vision 19(2), 81–93 (2003)
26. Trist, K., Ciesielski, V., Barile, P.: Can't See the Forest: Using an Evolutionary Algorithm to Produce an Animated Artwork. In: Huang, F., Wang, R.-C. (eds.) ArtsIT 2009. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 30, pp. 255–262. Springer, Heidelberg (2010)
27. Trujillo, L., Olague, G.: Using Evolution to Learn How to Perform Interest Point Detection. In: ICPR, pp. 211–214 (2006)
28. Trujillo, L., Olague, G.: Automated Design of Image Operators that Detect Interest Points. Evolutionary Computation 16(4), 483–507 (2008)
29. Olague, G., Puente, C.: Honeybees as an Intelligent based Approach for 3D Reconstruction. In: International Conference on Pattern Recognition, Hong Kong, China (2006)
30. Asuncion, A., Newman, D.J.: UCI Machine Learning Repository (2007)
31. Vázquez, R.A., Sossa, H.: A New Model of Associative Memories Network. In: Third International Workshop on Artificial Networks and Intelligent Information Processing (ANNIP 2007), Angers, France, May 9-12 (2007)
32. Vázquez, R.A., Sossa, H.: Hetero-Associative Memories for Voice Signal and Image Processing. In: Ruiz-Shulcloper, J., Kropatsch, W.G. (eds.) CIARP 2008. LNCS, vol. 5197, pp. 659–666. Springer, Heidelberg (2008)
33. Villegas-Cortez, J., Sossa, H., Aviles, C., Olague, G.: Automatic Synthesis of Associative Memories by Genetic Programming, a first approach. In: Research in Computing Science. Advances in Computer Science and Engineering, vol. 42, pp. 91–102 (2009)
34. Villegas-Cortez, J., Olague, G., Aviles, C., Sossa, H., Ferreyra, A.: Automatic Synthesis of Associative Memories through Genetic Programming: A First Co-evolutionary Approach. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcazar, A.I., Goh, C.-K., Merelo, J.J., Neri, F., Preuß, M., Togelius, J., Yannakakis, G.N. (eds.) EvoApplicatons 2010. LNCS, vol. 6024, pp. 344–351. Springer, Heidelberg (2010)
35. Villegas-Cortez, J., Sossa, H., Aviles, C., Olague, G.: Evolutionary Associative Memories Through Genetic Programming. Rev. Mex. Fis. 57(2), 110–116 (2011)
36. Yáñez Márquez, C., Díaz de León Santiago, J.L.: Memorias asociativas basadas en relaciones de orden y operaciones binarias. Ph. D. Thesis abstract. Computación y Sistemas 6(4), 300–311 (2003)
37. Zhang, M., Andreae, P., Pritchard, M.: Pixel Statistics and False Alarm Area in Genetic Programming for Object Detection. In: Raidl, G.R., Cagnoni, S., Cardalda, J.J.R., Corne, D.W., Gottlieb, J., Guillot, A., Hart, E., Johnson, C.G., Marchiori, E., Meyer, J.-A., Middendorf, M. (eds.) EvoWorkshops 2003, LNCS, vol. 2611, pp. 455–466. Springer, Heidelberg (2003)
38. Bhowan, U., Zhang, M., Johnston, M.: Genetic Programming for Classification with Unbalanced Data. In: Esparcia-Alcázar, A.I., Ekárt, A., Silva, S., Dignum, S., Uyar, A.Ş. (eds.) EuroGP 2010. LNCS, vol. 6021, pp. 1–13. Springer, Heidelberg (2010)
39. Wechsler, H., Zimmerman, G.L.: Distributed Associative Memory (DAM) for Bin-Picking. IEEE Trans. on Pattern Analysis and Machine Intelligence 11(8), 814–822 (1989)
40. Yao, X.: Evolving artificial neural networks. Proceedings of the IEEE 87(9), 1423–1447 (1999)

# Parallel Architectures for Improving the Performance of a GA Based Trading System

Iván Contreras, J. Ignacio Hidalgo,
Laura Nuñez-Letamendía, and Yiyi Jiang

**Abstract.** Research and development of automatic trading systems are becoming more frequent, as they can reach a high potential for predicting market movements. The use of these systems allows to manage a huge amount of data related to the factors that affect investment performance (macroeconomic variables, company information, industry indicators, market variables, etc.), while avoiding psychological reactions of traders when investing in financial markets. Movements in stock markets are continuous throughout each day, which requires trading systems must be supported by more powerful engines, since the amount of data to process grows, while the response time required to support operations is shortened. In this chapter we present two parallel implementations of a GA based trading system. The first uses a Grid Volunteer System based on *BOINC* and the second one takes advantage of a Graphic Processing Unit implementation.

## 1 Introduction

The investment process in real time on the stock market has become an increasingly difficult process as manifested in the daily news we receive in this environment of financial crisis. Several factors are influencing this process. These include the large amount of data available, the difficulty in the process

Iván Contreras
Complutense University of Madrid
e-mail: ivancontrerasFD@gmail.com

J. Ignacio Hidalgo
Complutense University of Madrid
e-mail: hidalgo@dacya.ucm.es

Laura Nuñez-Letamendía · Yiyi Jiang
IE Business School
e-mail: {Laura.Nunez,yiyi.yiang}@ie.edu

F.F. de Vega et al. (Eds.): Parallel Architectures & Bioinspired Algorithms, SCI 415, pp. 189–218.
springerlink.com

of valuation of financial assets and the growing complexity of financial and socio-economic environment. Each asset class has its peculiarities and therefore requires to perform its assessment from a different approach. We need to determine exactly which are the variables that affect their performance in order to anticipate his return, bearing in mind the risk factors and volatility.

Therefore, it seems logical to seek to develop automatic or mechanical trading systems that rely on technology to make predictions of values, selection of factors to consider and, of course, analyze and process all available information for investors. Other factors have contributed to the development and expansion of trading systems supported by computers:

- the popularization of systems with some degree of parallelism has led to the advancement of computing power. At present any desktop has a tremendous computing power unthinkable just a few years ago. This process has accelerated recently with the emergence of new graphics processing units (GPUs) working on massively parallel calculations
- the tremendous expansion of the globalization process symbolized for instance by the tremendous development of the "hedge funds industry", a class of funds which invest in any kind of assets around the world (stocks, indexes, bonds, commodities, currencies, etc.)
- the attempt to avoid the psychological aspects that biases the investment process (known in the literature as behavioral finance ([28])

Automatic and mechanical trading systems are based on rules that uses market, business or macroeconomic information embedded in algorithms that look for the best combination of these rules to drive the stock trades in an attempt of obtaining the maximum possible return for a period. Those systems have evolved from very simple *If-then* algorithms to more sophisticated models that use methods like artificial intelligence, chaos theory, fractals, evolutionary algorithms, nonlinear stochastic representations, econo-physics models, etc., which are ultimately based on market, fundamental or macroeconomic data.

Focusing on the subject of this book, some previous studies document the use of Genetic Algorithms (GA) and Evolutionary Computation (EC) to design and optimize automatic trading systems for the Stock Market (see [1] [23] [24] [17] [25] [14] [5]). We can affirm that, the investment practitioners have begun to use evolutionary algorithms to build automatic trading systems.

In [17] the authors describe a trading system designed with GAs that use different kind of rules with market and companies information, which is applied to trade, in a daily base, to companies belonging to the S&P 500 index. One of the difficulties the authors claim is the computational time required for training the trading system with daily data of stocks prices. This restriction is even more critical when we take into account that the majority of traders invest in an intra-day base (what means that the investment positions are canceled during the same day). Therefore, the operative in financial markets seems to recommend the use of an even shorter frequency in data when training and applying mechanical trading systems.

However, the focus on intra day (instead of daily data) cause some difficulties for the design and application of these trading systems because of the shorter time of respond we require, since it is not possible to wait for hours to obtain the investment decision result provided by the mechanical trading system when the investments have to be done continuously during the day. The necessity for speeding up the GA process, to get in time good results for this kind of trading applications, as well as its possibilities of parallelization allows us to use alternative ways of implementing these applications.

In this Chapter we describe the implementation on two differents parallel computer architectures to speed up the functioning of a GA-based trading system to invest in stocks: a corporative grid and a GPU-CPU architecture. Starting from the work presented in [17] and [10], both plataforms are compared in terms of computational time. This comparison is important when implementing the proposals in a real way, allowing execution times to generate buy and sell signals in times operationally functional. In other words, the more efficient is the parallelization, the more realistic is the possibility of using these systems for intraday data. Although in the scientific literature we can find, with increasing frequency, the use of such architectures for solving complex problems, they only have been marginally applied to markets. Experimental Results show how the combination of the GA and the parallel architectures allows us to obtain solutions for real time (or intra-day data) investment decision. It is very difficult to have access to intra-day data since the commercial databases for stock prices and stock exchange markets do not provide frequency data inferior to that of daily prices. The way to accede to intra-day data is through an investment bank, and usually they do not like to publish results obtained in joined research projects. Nevertheless, since the only difference between a trading-system based on intra-day data versus daily data is the respond time required, we apply our trading systems to daily data, because of the mentioned difficulty of accede to intra-day data. We show how we can use and configure the GA on the parallel architectures in order to analyze results for different companies at once.

The rest of the chapter is organized as follows. In the next section, section 2, we formulate the investment problem describing the proposed trading system to cope with it. In section 3 we explain the parallelization in the grid platform and Section 4 introduces the details of the GPU implementation using the CUDA architecture. Section 2 and Section 3 are based on [17], while section 4 is based on [10]. We summarize and conclude in Section 5, where we show both parallel architectures in opposition.

## 2 Formulation of the Investment Problem and Description of the Proposed Trading System

The investment problem is defined as to maximize return (or risk adjusted return) for a specific time period when investing long or short-sell in a financial asset (in our case a stock)[1]. Since the performance of investment decisions in stock markets is influenced by a wideness of factors of different type: political, macroeconomic, regulatory, local, international, etc. that are uncertain, there is not a single and perfect rule with a specific parameter or threshold value that can be used to maximize future returns, but a broad set of potential rules which combine indicators, representing different factors and driven by a range of values in their parameters. Therefore, the investment problem consists first of finding the best combination of indicators and second of fine-tuning the parameters for these indicators to obtain the maximum return when applied to the investment decision making in a stock. Thus, the input set of variables for a trading system consists of the indicators to be used as investment criteria and the parameters being the threshold values for these indicators, while the output is the return obtained by the trading system this way defined.

## 2.1 Two Types of Financial Analysis

When the objective of the analysis is to determine what stock to buy and at what price, there are two basic methodologies: fundametal analysis and technical analysis. Investors can use any or all of these different but somewhat complementary methods for stock picking. For example many fundamental investors use technicals for deciding entry and exit points. Many technical investors use the fundamental analysis to limit their universe of possible stock to good companies.

### 2.1.1 Fundamental Analysis

Fundamental analysis maintains that markets may misprice a security in the short run but that the “correct” price will eventually be reached. Profits can be made by trading the mispriced security and then waiting for the market to recognize its ”mistake” and reprice the security. Fundamental analysis of a business involves analyzing its financial statements and health, its management and competitive advantages, and its competitors and markets. When applied to bonds and forex, it focuses on the overall state of the economy (interest rates, production,etc...). When applied to companiesśhares. it focuses on accounting information from those companies.

A plethora of indicators can be applied, thus the first step to solve the investment problem is to select the indicators to be included in the decision making trading system. In this document we follow previous works [17] where the authors carried out an exhaustive analysis of the investment literature to select different classes of accounting indicators from which subsequently, they selected one indicator whitin every class by applying a GA. For sale positions the indicators selected were Price Cash Flow (PCF), Debt Over

[1] In this section we present and abridged description of the trading system proposed in [17]. For more details we refer the reader to it.

**Table 1** Indicators used for fundamental analysis based on [17]

| Acronym | Description | Formula |
|---|---|---|
| $PCF$ | Price Cash Flow | $PCF_t = \frac{PRCCQ_t \cdot CSHOQ_t}{NIQ_{t-1}+DPQ_{t-1}}$ |
| $DBE$ | Debt over Book Value Equity | $DBE_t = \frac{DLTTQ_{t-1}+DLCQ_{t-1}-CHEQ_{t-1}}{CEQQ_{t-1}}$ |
| $SG$ | Sales Growth | $SG_t = \frac{REVTQ_{t-1}}{REVTQ_{t-2}} - 1$ |
| $TOG$ | Turnover Growth | $TOG_t = \frac{REVTQ_{t-1}/ATQ_{t-1}}{REVTQ_{t-2}/ATQ_{t-2}} - 1$ |
| $PBV$ | Price Book Value | $PBV_t = \frac{PRCCQ_t \cdot CSHOQ_t}{CEQQ_{t-1}}$ |
| $ROA$ | Return on Asset | $ROA_t = \frac{NIQ_{t-1}}{ATQ_{t-1}}$ |

Book Equity (DBE), Sales Growth (SG) and Turnover Growth (TOG), while for buying, they selected Price Book Value (PBV), Sales Growth (SG) and Return On Assets (ROA). As the authors remark, all these indicators have been reported to be useful in the literature on investment (see [2] [4] [6] [8] [12] [13] [27], among others). The formulation of these indicators is presented in Table 1(for a broader description see [17]).

Indicators in Table 1[2] are applied to the investment process related to a threshold or parameter value[3]. For instance, the PCF can be used as follows: take a long position in the company stock if the PCF is below 8 / invest short or sell in the company stock if the PCF is above 15. Therefore it is necessary to fine-tune the parameter for each one of the indicators within a range following market practices (we use the same range that [17]). The trading systems signals are triggered by comparing the value of every indicator estimated

[2] To compute these indicators Jiang et al. obtained, from the COMPUSTAT database , quarterly data on the nine items (COMPUSTAT codes in parenthesis) from the companies´financial statements for the period January 1986 to December 2006: total assets (ATQ), total common ordinary equity (CEQQ), cash and short-term investments (CHEQ), debt in current liabilities (DLCQ), total long-term debt (DLTTQ), total depreciation and amortization (DPQ), net income / loss (NIQ), total revenue (REVTQ) and common shares outstanding (CSHOQ). From the Center for Research in Security Prices (CRSP) Databases prices (PRCCQ) are obtained and adjusted by stock dividends and splits. To ensure that the items used to compute the indicators at quarter t were known to the market as of quarter t, Jiang et al. used the data on quarter t-1 for all of them, except for the common shares outstanding and prices.

[3] Sales Growth (SG) indicator is used for both investment signals, long and short, however a different parameter is used for triggering each signal.

**Table 2** Fundamental trading Rules

| **Short-Sell Positions (-1)** | **Trading Rules** | **Threshold Value Range** |
|---|---|---|
| Price Cash Flow (PCF) | If $PCF \geq$ threshold value | 5-29 |
| Debt over Book Value Equity ($DBE$) | If $DBE \geq$ threshold value | 60-135% |
| Sales Growth ($SG$) | If $SG \leq$ threshold value | +5% to (-17.5%) |
| Turnover Growth ($TOG$) | If $TOG \leq$ threshold value | +1% to (-29%) |
| **Long or Buying Positions (+1)** | **Trading Rules** | **Threshold Value Range** |
| Price Book Value (PBV) | If PBV $\leq$ threshold value | 0,25 - 5,75 |
| Sales Growth ($SG$) | If SG $\geq$ threshold value | 1% - 23.5% |
| Return on Asset (ROA) | If ROA $\geq$ threshold value | 1% - 15% |
| **Neutral Positions (0)** | When variables values comparing with threshold values are not triggering short-sell or long positions | |

with the company information against the parameter value selected by the GA for each one of the indicators based on maximization of return. The fundamental trading Rules are showed in Table 2.

Consequently, the difficulty of solving the investment problem defined above depends on the number of indicators included in the trading system and the range allowed for the parameters to be used as threshold values for these indicators. Notice that we face a combinatorial optimization hard problem that is growing exponentially with both indicators and parameters range, being explosive when using a high number or range for both variables.

### 2.1.2 Technical Analysis

Technical analysis maintains that all information is reflected already in the stock price. Trends "are your friend" and sentiment changes predate and predict trend changes. Investors' emotional responses to price movements lead to recognizable price chart patterns. Technical analysis does not care what the 'value' of a stock is. Their price predictions are only extrapolations from historical price patterns.

Technicians using charts search for archetypal price chart patterns, such as the well-known head and shoulders or double top/bottom reversal patterns, study technical indicators, moving averages, and look for forms such as lines of support, resistance, channels, and more obscure formations such as flags, pennants, balance days and cup and handle patterns.

Technical analysts also widely use market indicators of many sorts, some of which are mathematical transformations of price, often including up and down volume, advance/decline data and other inputs. These indicators are used to help assess whether an asset is trending, and if it is, the probability of its direction and of continuation. Technicians also look for relationships between price/volume indices and market indicators. Examples include

**Table 3** Indicators used for technical analysis based on [17]

| **Description** | **Formula** |
|---|---|
| Crossing of Moving Average<br>$(MA)$ | $MA = MA_l - MA_s$<br>$MA_n = \frac{PRCCM_t + PRCCM_{t-1} + PRCCM_{t-2} + \ldots + PRCCM_{t-N}}{N}$ |
| Relative Strength Index Divergence<br>$(RSID)$ | $RSID = \begin{cases} 1 \text{ if } (RSI_t - RSI_{t-n} > 0) \& (PRCCD_t - PRCCD_{t-n} < 0) \\ -1 \text{ if } (RSI_t - RSI_{t-n} < 0) \& (PRCCD_t - PRCCD_{t-n} > 0) \\ 0 \quad \text{otherwise} \end{cases}$<br>$RSI = 100 - \frac{100}{1+RS}$<br>$RS = \frac{\sum_{i=1}^{m} UPCP_i}{\sum_{i=1}^{m} DWCP_i}$<br>$UPCP_i = (PRCCD_i - PRCCD_{i-1}) \forall PRCCD_i > PRCCD_{i-1}$<br>$DWCP_i = (PRCCD_{i-1} - PRCCD_i) \forall PRCCD_i < PRCCD_{i-1}$ |
| Support and Resistance Levels<br>$(SRL)$ | $SRL = (SL_n, RL_m)$<br>$SL_n$ Local Minimum PRCCM for the last n periods<br>$RL_m$ Local Maximun PRCCM for the last m periods |
| Volume Price Divergences<br>$(VPD)$ | $VPD = \begin{cases} 1 \text{ if } (VD_t - VD_{t-n} > 0) \& (PRCCD_t - PRCCD_{t-n} > 0) \\ -1 \text{ if } (VD_t - VD_{t-n} > 0) \& (PRCCD_t - PRCCD_{t-n} < 0) \\ 0 \quad \text{otherwise} \end{cases}$ |

the relative strength index, and Moving Average Convergence - Divergence (MACD). Other avenues of study include correlations between changes in options (implied volatility) and put/call ratios with price. Also important are sentiment indicators such as Put/Call ratios, bull/bear ratios, short interest, Implied Volatility, etc.

As it was the case for fundamental analysis, numerous technical indicators can be applied to trigger investment signals, thus, again, to solve the investment problem we need to select the indicators to be included in the trading system for which we follow again previous works in [17] where the authors selected the following technical indicators: Crossing of Moving Averages (MA); Relative Strength Index Divergence (RSID); Support and Resistance Levels (SRL) and Volume Price Divergences (VPD). The formulation of these indicators is presented in table 3 (for a broader description see [17]).

**Table 4** Technical Trading Rules

| Indicator<br>& Parameter Range | Trading Rules | Investment Signals |
|---|---|---|
| $MA$<br>s (1-31)<br>l (33-93) | if $MA_s - MA_l > 0$<br>if $MA_s - MA_l < 0$<br>otherwise | Long Position (+1)<br>Short Position (-1)<br>Neutral Position (0) |
| $RSID$<br>n (1-31)<br>m (1-31) | if $RSID = 1$<br>if $RSID = -1$<br>if $RSID = 0$ | Long Position (+1)<br>Short Position (-1)<br>Neutral Position (0) |
| $SRL$<br>n (1-76)<br>m (1-76) | if $PRCCD > RL_m$<br>if $PRCCD < SLn$<br>otherwise | Long Position (+1)<br>Short Position (-1)<br>Neutral Position (0) |
| $VPD$<br>n (1-31) | if $VPD_n = 1$<br>if $VPD_n = -1$<br>if $VPD_n = 0$ | Long Position (+1)<br>Short Position (-1)<br>Neutral Position (0) |

These technical indicators presented in table 3[4,5] are driven by parameters when used in the investment process. For instance, the MA needs two parameters that are l and s and give us the number of daily prices we need to use to compute each moving average. Then, it is necessary to fine-tune the parameters for each one of the indicators within a range following market practices (we use the same range that [17]). Trading systems signals are triggered by the technical rules as described in Table 4.

## *2.2 Sample Data*

The sample of firms for the trading systems in [17] is the same used here for comparing both platforms and comprises all companies included in the S&P 500 for at least two quarters during the period January 1986 to December 2006 with non-missing values for the variables required for at least the previous consecutive 20 quarters (if available 40 quarters are used), with the exception of those companies belonging to the finance, insurance and

[4] Besides the variables already mentioned in table 1, Jiang at al. take daily prices (PRCCD) and volume (VD) from CRSP Database adjusted by dividends and splits. We use the same data.

[5] *UPCP* and *DWCP* stands for *UP changes in price* and *DOWN changes in price* respectively.

real estate industries. In this way, the number of companies in the sample is 599 with 332.710 observations for quarterly fundamental data and 7.157.320 observations for daily technical data. Original data in [17] are gathered from Compustat and CRSP databases.

## 2.3 Genetic Algorithm

As it is well known, the design of a GA involves some key factors that will depend heavily on the characteristics of the problem at hand. One of them is the chromosome encoding used for representing the solutions by means of some type of code (binary, real, etc.). In this system the GA is the responsible of selecting threshold values of the indicators that comprise the trading systems or investment rules.

The chromosomes representing the parameters or threshold values for both types of indicators fundamental and technical are encoded using binary code with 4 genes for each parameter. For fundamental indicators we need to encode 7 threshold values for the 7 variables used to guide the trading systems (four for short-sell investments and three for long positions). For technical indicators we also need to encode 7 parameters values, since each one of the four indicators uses 2 parameters with the exception of the Volume Price Divergence indicator that needs only one parameter. Therefore, each chromosome for each type of trading system (fundamental and technical) comprises 28 genes (7 parameters X 4 genes) giving a total search space of $2^{28} = 268.435.456$ possible combinations. This huge space implies that the analysis of more than 2.5 millions of potential combinations of indicators with different parameter values would reach only 1% of the full search space.

For selection we use the roulette wheel [15] method initially (on Boinc system) and tournament selection (on the GPU). Crossover and mutation are implemented as usual based on one-point crossover and mutation. The GA choice of threshold values is driven by the value of the fitness function consisting in the accumulated return obtained when applying the trading systems to the sample data computed as described below:

$$AR_f = \prod_{f}^{i=1}(1 + DR_i) \tag{1}$$

Where $AR_f$ is the accumulated return at the end of the trading period and $DR_i$ is the daily return given by:

$$DR_i = \begin{cases} \frac{P_i - P_{i-1}}{P_{i-1}} & \text{if the TS gives a long signal} \\ -\frac{P_i - P_{i-1}}{P_{i-1}} & \text{if the TS signal is short selling} \\ RFDR_i & \text{if the TS signal is neutral} \end{cases} \tag{2}$$

$P_i$ denotes the stock price at day "$i$", while $RFDR_i$ is the risk-free daily return given by the US Treasury Bills, and TS stands for Trading System.

The main problem of this approach to design trading systems is the execution time that takes some time, what makes difficult to apply these systems for intra-day trading. With a sequential execution of the GA we are not able to apply these trading systems to real time problems with intra-day data. In the following sections we explain how we can approach this problem by using several parallel implementations. For this purpose we made first an analysis of the execution time for the whole program and, based on the results, we implemented several important changes not only in the structure of the program, but also in the genetic operators.

## 3 Parallelization with a Grid System

Buying supercomputers requires heavy investment that can be avoided by setting up computer grids. In order to carry out its complex financial operations, a Bank or a company can use the idle time of computers on its Local Area Network (LAN). This solution has many advantages. First, it is relatively cheap and second, it is scalable. If the company needs more computing power, it will only have to tighten its grid by adding more computers to it. Some companies already provide these sevices, thus these allow hiring his services when another company require it, for instance the enterprise *Grid Systems* [11].

Other alternative is to use volunteers computers [9] [16]. We have implemented our grid system using the *Berkeley Open Infrastructure for Network Computing* (*BOINC*[6]), which allows to interconect a set of voluntaries computers. People interested in helping to science can joined these projects allowing to a Boinc server to use his idle computer time. Boinc is an open software system developed with the main intent of achieve a massive computing capacity. This feature is carried out with the interconexion of computers trought ethernet, either LAN or WAN, like a grid system. The huge computation power of Boinc falls in the volunteer users.

### *3.1 The BOINC Architecture*

The Boinc architecture is a client-server model. Therefore the Boinc framework consists of two layers which operate under the client-server architecture. Once the BOINC software is installed on a computer, the server starts sending tasks to the client. The operations are executed in the client and finally, the results are uploaded to the server.

[6] Open-source software for volunteer computing and grid computing. University of California at Berkeley. http://boinc.berkeley.edu

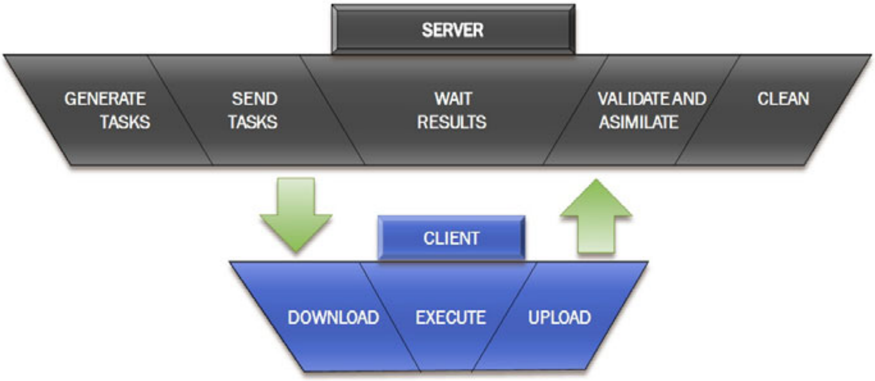


**Fig. 1** The typical Boinc project structure

### *3.2 Boinc Server*

The Boinc Server (BS) is responsible for planning and scheduling tasks for the projects. The server interacts with all others machines, and it is the manager of sending and receiving works and results. There are several ways to install the Boinc server software.

- *Server like a virtual machine*: Boinc provides a virtual machine to use as BS. This BS includes all necessary requirements to be used. These features make the BS one of the best options when you are starting with it. The operative system of the virtual machine is one distribution of Linux, specifically a basic distribution of Debian. This version do not includes a graphical interface, however you can install it later. All Boinc software is provided with the original package and with all programs already compiled. Furthermore this package has a ready user accounts with execution permissions. The virtual machine can be running with several software like VirtualBox (free[7]) or VMware (you can download free limited versions[8]).
- *Independent BS*: For experimentation and debugging, you can use almost any computer as a BOINC server. However, when the size of the project grows, it is necessary an independent BS for an optimal use of this software. If our project is executed in a huge amount of machines it is highly recommended to use the independent BS. Given the fact that we have a specific server machine, we use it for this project, thus we ensure the performance, availability, and security of the BS.

  The features of our server are exposed in Table 5. Together with the mentioned characteristics of the server we should have an internet connection with adequate performance and a static IP address.

[7] https://www.virtualbox.org
[8] http://www.wmware.com

When we want to set up an application in the environment of Boinc, we need to create and to configure a project. A Boinc project is the form of naming a set of applications with a common objective and the configurations needed to be executed in the Boinc platform. When speaking about a Boinc application we refer to a specific program inside a project and each application consists of a set of works to be done. An application must have a name ("short name") that will be used to name folders and files of the application and a common name by which the volunteers know the project ("friendly name").

Once an application is ready for execution the BS should create tasks for starting the grid computation. These tasks are commonly named as *jobs* for the programmers. Boinc jobs have two different parts, workunits and results. A workunit is technically a portion of the program that describes the computation to be performed. A workunit has one or more results, each of which describes an instance of a computation, either unstarted, in progress, or completed. The Boinc client software refers to results as "tasks".

We can use also the *Wrapper* which is a program provided by Boinc able to execute any type of application in the environment of this platform. Thus, Wrapper is a very useful tool when the programmer cannot access to the source code, or in cases where the code is difficult for support changes, even when the code is developed in programming languages not actually supported by Boinc. This program encapsulates the original application, as a result Boinc can process the application. It can execute a sequential number of applications and it can establish checkpoints for heavy applications. Wrapper executes the programs as subprocesses, so the communications with the Boinc environment are fluid.

The BS database is a set of tables and indexes that contains the information stored for all projects in the Boinc server. Boinc stores the data in a MySQL database. We can find tables for the workunits, results, applications, user, etc. The database for each project is generated by the make_project script (this is a script that you should execute when you create a new project). Usually, the user don't have to directly examine or manipulate the database, however sometimes it is very useful to manage the workunits and results, because we can obtain statistics, delete the failed workunits, etc. There are several ways to access to the database, the MySQL command-line interpreter or the Boinc's administrative web interface are the most common.

In order to develop applications with Boinc it is important to know the Boinc client. The Boinc client is a light program (in terms of memory and compuation requirements) that remains in communication with the server. The most important thing that we must know is the structure for the inputs and outputs. On Boinc client computers, each project has an specific directory where all data related to it is stored. Every executed task will have a different folder and these will reside in the directory *slots*. Each of these folder has an identification number. Furthermore, this directory will have the links to the inputs and outputs of the tasks.

### *3.3 Modifications in the Code*

In order to port our GA to the Boinc grid we made some neccesary modifications in the code. The original GA code is an unique program that takes as input a matrix with the data of the companies in the S&P 500 for a period of 1976 to 2006. This program executes the GA approximately 4500 times, once for each company and year . The way for parallelize the program is to divide these thousands of executions in independent tasks. Of course, 4500 tasks are enough for achieving one of the best configurations with our grid. The tasks are enough big for not overloading the server with a lot of requests, and are little enough for not overloading the computers with long executions. Thanks to this way of partition of the program, the modifications needed in the code for achieving a parallelized program are small. Mainly, we need to change the main file where the execution of the GA is processed and create a script for reading the data, executing the algorithm and saving the data, in this order and with the correct parameters.

The script executes the program in the adequate order, it is a file written in Matlab programing language. This script receives an unique string with the year and the company number (acording to the data that the program runs). First the script reads data from the company to be stored in an array. Second, it executes the algorithm with the default parameters, as the number of individuals, generations, mutation and crossover probabilities, etc. The output data is stored in a file with the same name of the data of the company.

In addition, we needed to do modifications in the main file of the GA, changing the code for processing the data just for one company.

### *3.4 Parallelization Tasks*

In order to parallelize the execution we made some parallelization tasks. Regarding the input data, the original code has an Excel book as input. This Excel file is divided by years, the pages of the book, and by companies, all on the same page. These files have a weight between 20 MBytes and 80 MBytes, so if the Boinc server sends the full input to the client computers, we will overload the grid system. This overload is unnecessary because we use only one company in the execution of the program in a grid node. Then we need to divide the input data into independent data sets (one for each company). Thus, the program will run with the minimum amount of data. We have created a little program that divides the original data in a collection of Excel files. Finally we have around 10000 files: 5000 for technical analysis and the same number for the fundamental analysis.

At the same time, when all tasks are ended, we have a great amount of files with the final results (one result for each task). We need to develop a program that combines all the results in the same once more. The results are stored in a “mat”file with several matrices and variables. These matrices and variables represents the final result of our GA, then we take all the results and create a collection of matrices and variables with another dimension. With the pre-process we achieve all the results in one file, and a grid with more fluid communications.

## *3.5 Matlab in the Client*

The first option to consider is to install a version of Matlab in the client. In this way the source program must be a script (sh script) that executes Matlab in the client. The input of this script should be the name of the main file of our program. The main file will be created like an executable. It is highly recommended to run the program without graphical interface, thus the user (computer where the Boinc client is installed) has no knowledge of his execution. If we follow this way, all the files needed for our applications must be added to the input template, including the main file of our program, because the file executed by the wrapper is the script mentioned above. This option has a big problem, because all the clients must have installed the Matlab program, which limited the number of potential volunteers. Moreover, the software is a program with a not free license. Even in the case that the client has the Matlab software licensed and installed, the execution may throw exceptions due to incompatibilities between different versions of the software.

The second and more desirable way is to use a Matlab Executable file encapsulated in the Boinc Wrapper. Matlab has the ability to convert a program written in its language to a single executable file that will depend on the target platform. For the right operation of the executable file, the client computer should have MCR (Matlab component runtime) installed. MCR can run almost all Matlab functions and is freely distributed with the library files generated by the Matlab compiler.

It is also possible to package the application together with MCR libraries. In this way we would have just one executable file which brings together all the elements needed to run the application on any computer. The MCR libraries are distributed by Matlab.

## *3.6 Experimental Results*

In this sections we will present a description of the experimental tests that we have done with the program in the Boinc enviroment.

### 3.6.1 Metrics

The set of experimental tests have been carry out in a Boinc grid installed in CES Felipe II *(A Computer University College of Aranjuez, Madrid, Spain)*, thus we have used the computers of different laboratories and some well-known volunteers *(falua.cesfelipesegundo.com)*. Tests are used for estimating the grid computing power, because the capability to support volunteers carries a non-constant computing power. These tests consist of a series of computing time measurements in a independent CPU and in the computing grid. The execution times of the grid tests were obtained with the administrator page of the Boinc project. Moreover, the execution times of the basics executions in the CPU were measured with the Matlab Profiler tool in a computer with a Pentium 4 processor (see Table 6). Boinc client and the MCR libraries of Matlab were installed in all computers. The trading system has been executed fully, for all companies and for all years, which means approximately 10000 runs of the GA. In Table 5 we can see the main characteristics of the server.

**Table 5** Main characteristics of the Server used in the experiments

| Server | IBM xSeries 236 Type 8841 |
|---|---|
| Processor | Intel Xeon 2,8 GHz |
| RAM | 2 Gb |
| HDD | x4 70Gb - Raid 5 |

Table 6 summarizes the main characteristics of the different groups of computers that comprise the system (processor type and operating system) and the measures undertaken to estimate the computing capacity of the system.

We have carried out two tests of performance in each of the computers for determine both magnitudes (GFLOPS and GMIPS). The name of the test are *Whetstone* and *Dhrystone* provided by Boinc. Once done, it is taken the maximum value of the same for each group of computers with the same processor and operating system. In short, the grid Falua has about 225 GFLOPS and 433 GMIPS. These numbers supposed the grid at full capacity, with all the computers active and available.

Others software packages are used for managing the Boinc grid. We need to manage all computers (no volunteers) in our grid, because it is very uncomfortable to manipulate the computers independently. We used the software EMCO Remote Shutdown[9], to a great extend for turning on/off our computers. We also need a manager for the Boinc client, it is heavy and inefficient going computer by computer, for example joining the computer in a project, or requesting more tasks. For this purpose we used the BoincViewer that facilitates the use of a lot of computers with the Boinc client, that allows you to manage the Boinc Client on a single PC.

### 3.6.2 Execution Time Analysis of the Algorithm in the Grid

Figure 2 shows the experimental results for different number of individuals (X-axis) vs. Total Execution Time (Y-axis) and 500 generations. Note that execution time is represented using a 10-base logarithm scale.

[9] http://emcosoftware.com/

**Table 6** Main components of the Grid

| **Group** | **PCs** | **CPU** | **Operative System** | GFLOPS | GIPS | **Total** GFLOPS | **Total** GIPS |
|---|---|---|---|---|---|---|---|
| **Lab. ITIS 1** | 20 | 2 x Intel P4 3GHz | Windows XP x86 | 2,744 | 5,101 | **54,88** | **102,02** |
| **Lab. ITIS 2** | 21 | 2x Intel P4 3GHz | Windows XP x86 | 2,744 | 5,194 | **57,624** | **109,074** |
| **Lab. ITIS 3** | 22 | 2x Intel P4 3GHz | Windows XP x86 | 2,744 | 5,01 | **60,368** | **111,22** |
| **Lab. I4** | 1 | 2x Intel E2200 2GHz | Ubuntu Linux x86 | 1,853 | 5,436 | **5,905** | **13,204** |
| | 2 | 2x Intel P4 3GHz | Ubuntu Linux x86 | 2,026 | 3,884 | | |
| **Lab. DOSI I+D** | 2 | 2x AMD Athlon 4600+ | Windows XP x86 | 4,906 | 8,974 | **11,84** | **21,556** |
| | 1 | 2x Intel P4 3GHz | Ubuntu Linux x86 | 2,028 | 3,608 | | |
| **Volunteers** | 6 | 2x Intel P4 3GHz | XP x86 XP x86 | 2,722 | 4,747 | **35,128** | **77,53** |
| | 2 | 2x Intel P4 3GHz | Ubuntu Linux x86 | 1,847 | 2,876 | | |
| | 1 | 4x Intel i5 750 2.7GHz | Windows 7 x64 | 11,492 | 36,736 | | |
| | 1 | AMD Athlon 2600+ | Windows XP x86 | 2,129 | 3,578 | | |
| | 1 | 2x Intel T2450 2GHz | Windows XP x86 | 0,802 | 1,508 | | |
| | 1 | Pentium III Coppermine | Ubuntu Linux x86 | 0,679 | 1,474 | | |
| **TOTAL** | | | | | | **225,745** | **433,604** |

The grid implementation provides lower execution times, even for a small number of individuals. Technical Analysis is computational heavier than Fundamental Analysis, since it uses huge amounts of data. Due to that it achieves the highest compuation times. We can observe the inefficient compuation times of an independent CPU implementation. For example, we can analyze the first bars of Figure 2, where the times of technical analysis (500 generations, 500 individuals) are more than 100 days (184,9 days) and approximately one week for the grid version (4,11 days). We can see that for fundamental analysis the grid version achieves the results in a few hours (5,6 hours) while one single computer spends about 10 days (12,64 days). 100 days of execution is a very high time, fully uneasy, even more if we think that the execution depends only on one machine, with an execution without checkpoints. In this way, the system becomes easily susceptible to any danger or event, like a failure of the software or a cut of energy.

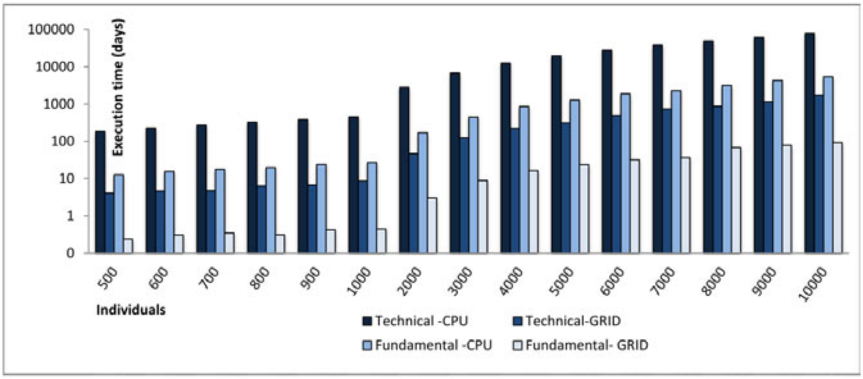


**Fig. 2** Execution times in the grid (y- axis) for 500 generations and different number of individuals (x- axis)

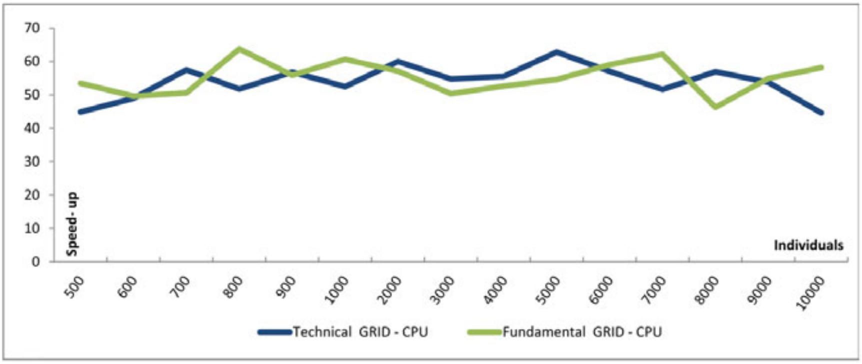


**Fig. 3** Speed-up in the grid

Figure 3 represents the speed-up in the grid. We can observe that figures are approximately constants. The achieved speed-up is around 50 and this ratio remains constant throughout all the experimental tests. The speed-up is not continue at all, this is because the computers in the grid do not have a continued availability, one computer could be off-line for hours.

## 4 Parallelization with a Computer Graphic Card

On last section we have explained a parallel implementation of the trading system using a grid enviroment. Graphic Processing Units (GPUs) are more and more extended for hihgly parallel applications. In this section we presents some results of a GPU-CPU implementation of our GA.

We can find in the literature several approximations for implementing evolutionary algorithms on GPUs ([20] [18] [26] [3] [22] [29]). Most of them rely on the CUDA architecture[10] and provide detailed information on how to configure the control parameters in order to obtain an efficient implementation. Those works show that doing an ad-hoc implementation require a good level of knowledge on a set of computer architecture and programming issues. However, that could represent a serious problem when a trader tries to use GPUs. The proposal explained in this section differs from previous approaches offering an adaptable tool for investors with no special knowledge on computer architecture, although familiar with Matlab tools[11]. In this way, we proposed an implementation based on a software tool named *Jacket* by AceelerEyes[12]. This section explains the general structure of a graphics device, the motivation for this selection and several parallelization and implementation details. This implementation is based on previous authors work [10].

### 4.1 CUDA Architecture

Multithreading in general purpose processors is used for taking full advantage of available resources. The processor in collaboration with the operating system can process instructions of two or more threads simultaneously. In tasks designed for graphics processors, the parallelism is easily exploitable. There are calculations to be performed for each vertex or each fragment, which means repeating the same task over and over again on different data in memory, so the idea of parallelism and multithreading is essential in the design of current GPUs programs.

Figure 4 depicts a high-level view of the GeForce GTX 280 GPU parallel computing architecture. A hardware-based thread scheduler at the top manages the scheduling of threads across the Thread Processing Clusters (TPCs). Furthermore, we have fully operative a texture cache and memory interface units. Texture caches are used to combine memory accesses for more efficient and higher bandwidth memory read/write operations. The elements indicated as "atomic" refer to the ability to perform atomic read-modify-write operations to memory. Atomic access provides granular access to memory locations and facilitates parallel reductions and parallel data structure management.

A Thread Procesing Cluster (TPC) in computing mode is represented in Figure 5. Each TPC is made up of a number of streaming multiprocessors (SMs), and each streaming multiprocessors contains eight processor cores or Processing Elements (PE). It can be observed that a local shared memory is included in each of the three SMs. Each processing core in an SM can

[10] Compute Unified Device Architecture Reference Manual: http://developer.download.nvidia.com

[11] http://www.mathworks.com/products/matlab/

[12] Jacket Documentation: http://www.accelereyes.com/support/documentation

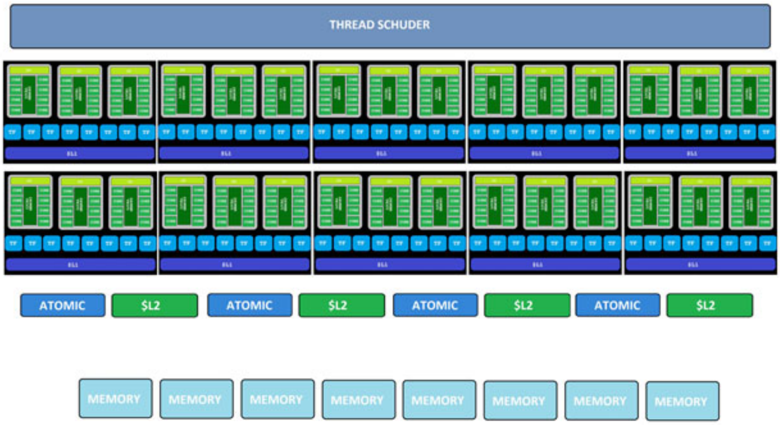


**Fig. 4** GeForce GTX 280 GPU Parallel Computing Architecture

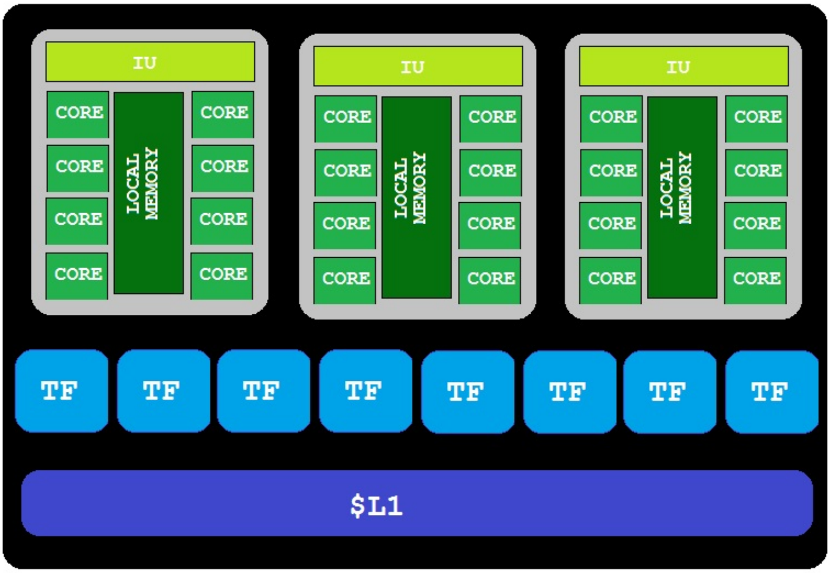


**Fig. 5** Detail of a Thread Procesing Cluster (TPC)

share data with other processing cores in the SM via the shared memory, without reading or writing from any external memory subsystem. This contributes greatly to increased computational speed and efficiency for a variety of algorithms.

### 4.1.1 Execution Time Analysis in CPU

Being a genetic algorithm, the main program structure is a *for loop* with a certain number of generations. Parallelizing directly the execution of this *for* in a lot of threads is unfeasible, since this would prevent the populations evolution. To parallelize these cycles, we would have to change the basic structure of the algorithm as other approximations, such as island model, usually do ([7]). Having in mind the structure of the GA, it is necessary to look for a parallelization in its basic operators: selection, evaluation, crossover, etc. In order to determine the critical elements a time analysis for the different processes that form the main program has been done. To analyze the program execution time, a Matlab profiler has been used, with which the different execution time used by each function can be distinguished. Thanks to these time measurements the parallelism of the most controversial areas can be influenced in so far as to computation time.

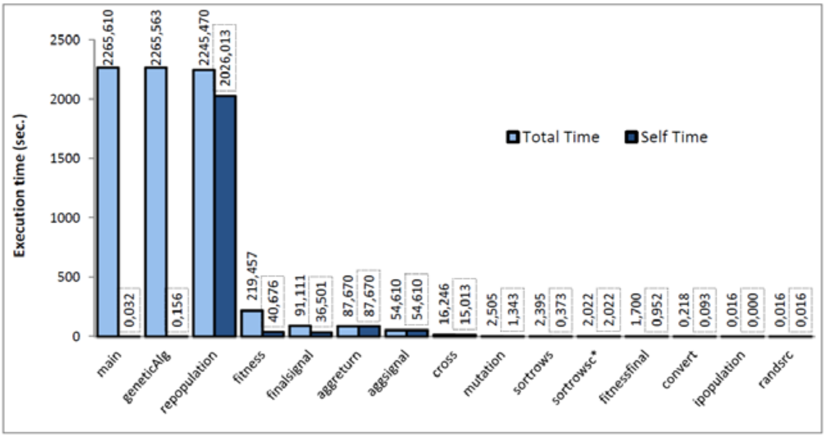


**Fig. 6** Genetic algorithm execution in the CPU. It has been run for a company with 5000 individuals; 500 generations and the roulette wheel selection algorithm. Total time = 2266 seconds.

Figure 6 shows the 15 functions with greater weight in the GA computation time. Their names are listed in the base of each column. For example, "*main*"is the main program and "geneticAlg"is the main genetic loop. Some function names are marked with an asterisk; all these functions are MEX functions. These functions have been written in C language, and are used to manage external libraries, in this case the *sortrows* function. *Total Time* is the time that the program is working, as it can be noticed *main* is active practically all the time during the program execution as expected. *Self time* is the total time minus the time that shares with the calls to other functions. The *repopulation* file is the part of the GA code which evaluates the random population and selects a new population for a further crossover. As it can be observed in the graphic, *repopulation* spends part of the time running in other files that correspond to population assessment (i.e. *fitness*) and other time running a selection algorithm directly embedded in *repopulation*. Knowing the division of this file and analyzing the graphic we can conclude that most of the time is devoted to run the selection algorithm, taking almost 90% of the execution time. One important point to remark is that the initial implementation used roulette wheel selection, widely used in genetic algorithms although computationally expensive.

As we can check, the main limitation of the efficiency of the program was found in the selection algorithm, and it is here where we should focus the parallelization in order to reduce the execution time of the algorithm. However, we not only will affect the parallelism of the selection but also will try to optimize the cost of GPU-CPU context switching. So, in order to make more profitable the parallelization in the GPU, we shall use during the main loop the data located in the graphic card and run the greater part of the program in the GPU. As we have mentioned, we take profit of *Jacket* capabilities to perform the parallelization of the code. For details about the parallelization tasks and more information we refer the reader to [10].

Special attention should be paid to the selection algorithm. The roulette algorithm has a computing cost of quadratic order, since it consists of two nested loops. *Jacket* provides a large capacity of parallelization with *for* loops, thanks to the *gfor* command. However *Jacket* is only compatible with simple while and *for* loops, not with more complex loops (as the needed for the roulette wheel method), and therefore the point where the problem requires more parallelism could not be solved in a sufficiently effective way to get a substantial advantage in the execution on the GPU.

With this motivation we decided to replace the roulette wheel selection algorithm by another one, also classic in genetic algorithms; the tournament selection [21]. This algorithm, which is able to obtain the same (or even better) quality in the results, is clearly less computationally expensive because it has a cost of lineal order. Some other previous implementation of Evolutionary Algorithms on GPUs adopted similar solutions ([1] [20] [19] [18]).

This change in the selection operator, has led to a decrease of approximately 75% of the total execution time. Nevertheless, the reduction of the execution time in this algorithm does not reduce the time spent in the selection in proportion with the other functions used. Still, the weight of the tournament selection algorithm is more than half of the total execution time of the program.

## *4.2 Experimental Results*

### 4.2.1 Metrics

The experimental results presented in this section are based upon a series of tests executed in both CPU and GPU. These tests consist of a series of computing time measurements in both processing units. Times have been measured using Matlab software. Due to the stochastic nature of GAs, all experimental tests have been executed 30 times. Once obtained the required data, a graph is presented to show and interpret the data in a simple way. The data used for graphs were obtained by the arithmetic average of all previous tests. A speed-up graphic is also included to evaluate the improvement of the execution time in the GPU.

Three Different CPU architectures (see table 7) has been used to compare the execution time with the GPU. These CPUs have been chosen due to their great variety of characteristics, for instance the P4 is the oldest CPU and has only capacity to execute one thread, whereas the i7-860 processor is a modern one with a capacity to execute up to 8 threads simultaneously. The SU4100 CPU is an intermediate architecture with a capacity to process two different threads simultaneously.

*Jacket* GPU programming is only compatible with *nVidia* graphic cards with CUDA technology. For the tests conducted here a 460GTX and a 570GTX nVidia GPUS have been used. This is a modern hardware, a range normally used for entertainment and with prices of 300 and 150 euros respectively. These graphic cards have been assembled in the third computer of Table 7, that is, the $i7-860$ CPU computer. Table 8 summarizes the main features of the GPU. The data previously presented on Figure 6 have

**Table 7** CPU Architectures used for comparison

| Processor | Intel Pentium 4 | Intel Pentium SU4100 | Intel Core i7-860 |
|---|---|---|---|
| Number of cores | 1 | 2 | 4 |
| Number of threads | 1 | 2 | 8 |
| Max. Frequency | 2.8 GHz | 1.3 GHz | 3.46 GHz |
| Cache | 512 KB L2 Cache | 2 MB L2 Cache | 8 MB Intel Smart Cache |
| System Bus | 533 MHz | 800 MHz | 2.5 GT/s |
| Operating system | Windows XP-32-bit | Windows 7 64-bit | Windows 7 64-bit |
| RAM -Memory | 768 MB | 4GB | 8GB |

**Table 8** Main characteristics of the GPUs used in the experiments

| Graphic Card | MSI nVidia 460 GTX OC | Gigabyte nVidia 570 GTX |
|---|---|---|
| CUDA Cores | 336 | 480 |
| Memory | 768 MB | 1280 MB |
| Clock for graphics | 725 MHz | 732 MHz |
| Clock for processor | 1350 MHz | 1464 MHz |

been obtained for the same architecture, corresponding to the third column in Table 7.

### 4.2.2 Execution Time Analysis of the Algorithm in the GPU

By implementing the code in the GPU the searched objectives have been achieved, the selection function is no longer a bottleneck in the program. The program total execution time has been reduced, in this particular case, around 90% if compared to the CPU version with the same selection algorithm, and to a 97% if compared to the original algorithm.

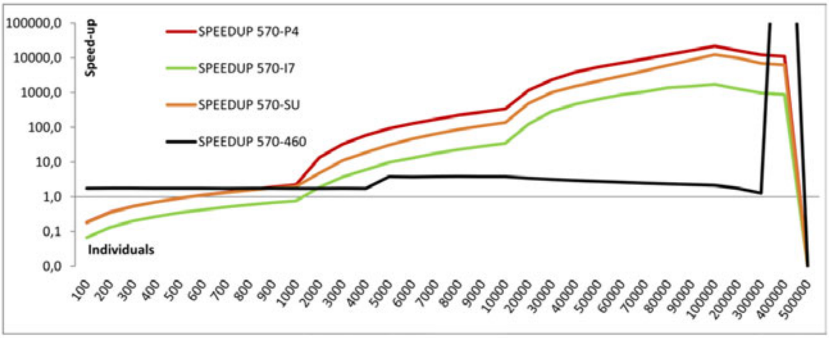


**Fig. 7** Speed-up on the GPU (y-axis)with variable number of individuals (x-axis) and 500 generations

Figure 7 represents the speed-up obtained with the GA execution in the graphic card. In this figure y-axis represents the speed-up for a certain number of generations. We can observe from negative speed-ups (below 1) to very high improvements, around 10000 units. We can observe the rise and sudden drop of the speedup of the 570GTX on 460GTX in the last two series of executions. The rise is due to the lack of memory in the 460GTX, and the drop is a consecuence of the lack of memory of the 570GTX.

The number of generations is the number of times that the algorithm will iterate. As no parallelization technique has been applied in this loop, its execution time would be in proportion to this parameter. This affirmation is not exact due to the fact that the size of the data also influences the program execution time. That is, the more GA generations, the more data will be stored by Matlab (not only the final results are stored). At each iteration, Matlab stores the new array of fitness and the best individuals. Storing a large amount of data cause the slowdown of Matlab with the advance of the GA step. Anyway, regardless the number of generations of the GA runnning on the CPU, the improvement margin among the different CPUs will be the same. However, this situation does not occur in GA execution over the GPU.

For this implementation, with this parallelization software and our strategy, the number of generations will influence the GPU speed-up due to memory overloads. Each additional iteration bears an extra cost, although sometimes small, what is true is that after a great number of generations it will have repercussions on the final execution time. To testify this event, another series of runs of the GA has been done. Figure 8 and Figure 9 summarize the results of this set of experiments.

Figure 8 represents a series of executions of 100 generations on the different GPUs. As it can be observed comparing both figures, the execution times on CPUs are proportional. As an example, we can look at the value of the execution time for the $i7-860$ with 6000 individuals. For this number of individuals and 100 generations the value of the execution time is approximately 150 seconds (exactly 154.3 seconds). On the other hand, the measure taken for the $i7-860$ with 6000 individuals for 500 generations is 775.92 seconds. If we multiply by five the execution time for 100 generations, we

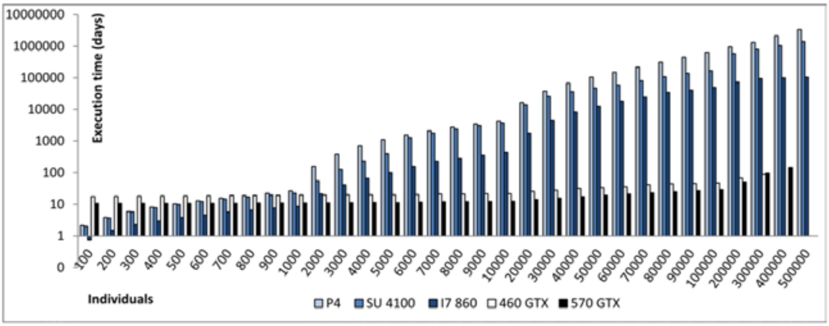


**Fig. 8** Execution times (y-Axis) for 100 generations and different number of individuals (x-axis)

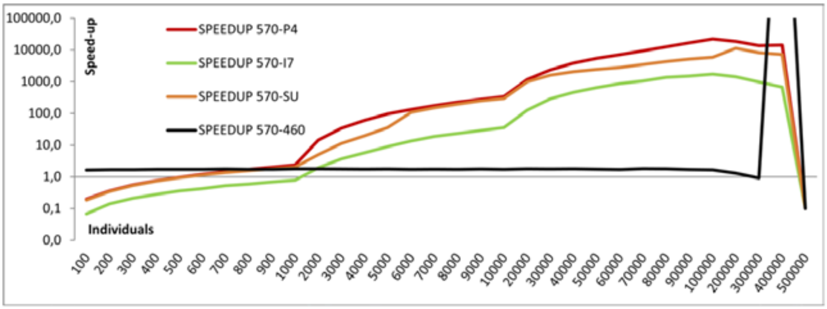


**Fig. 9** Speed-up : Speed-up on the GPU (y-axis) with variable number of individuals(x-axis) and 100 generations

obtain approximately the same measure than for 500 iterations. Nevertheless if we make the same operation for the GPU times, that is multiply 20.26 seconds (Figure 8) by 5, we obtain 100.15 seconds. However the execution time of the program is 219 seconds, which shows the non-proportionality of the execution time when changing the number of generations.

Figure 9 shows the speed-up of time for the series of executions of Figure 8. As previously explained it can be observed that the non-proportionality of the generations impacts in the improvement margin of the GPU if compared to the CPUs, thus remaining a much better margin of improvement for 100 cycles (generations). So eventually we conclude that for less number of generations and more than 2000 individuals the speed-up of the execution time on the GPU will be higher, and the higher the number of generations, the lower the improvement on time.

## 5 Conclusions

This sections depictes the conclussion of the chapter. First we expose some conclussions comparing the two parallel implementations of the trading system; Grid computing versus GPU computing, then we make a performance analysis. Finally we summarize some ideas that may be useful to the reader in future approaches to similar problems, which seek to improve both the quality of the solutions as the run time using parallel implementations.

### *5.1 Grid Computing versus GPU Computing*

We have presented two parallel implementations of the same program; a genetic algorithm for trading systems that can provide the investor a set of signals for buying and selling financial stocks. In this section we will compare the two alternatives Jacket and Boinc (i.e. the GPU and the grid).

- Speaking about the difficulty of developing a project in both platforms. On the one hand, developing an application for the Boinc platform requires a larger knowledge than doing the same development using Jacket. Settting up a Boinc project request a middle level of C language, SQL, Bash, etc. Nevertheless, the designer only needs a good knowdelege of Matlab language to develop an application with Jacket, that means that the learning curve in Jacket is softer than for Boinc.
- If we have an application that has not been developed with Matlab, we have access to a grid architecture, we have a system to attract volunteers and we want to reduce the performance in the execution times, surely we should develop for Boinc. Boinc allows us the parallelization in several programming languages, furthermore it includes some tools like the *wrapper* which enables the encapsulation of applications developed in not compatibles languages (see section 3.2).

- Boinc platforms are more stable and reliable for some specific aspects. For example, the execution in this system can be stopped at any time and back to the execution whenever the user needs. A GPU cannot stop the execution without the lost of partial results.
- GPUs and architetures like CUDA have been developed rapidly in recent years. The key advantage of using a GPU-CPU architecture as the one presented in this chapter, is that we could develop a realistic and independent investment environment, i.e. a professional in this area could get a computer system to have its own infrastructure for no more than 1000 $.

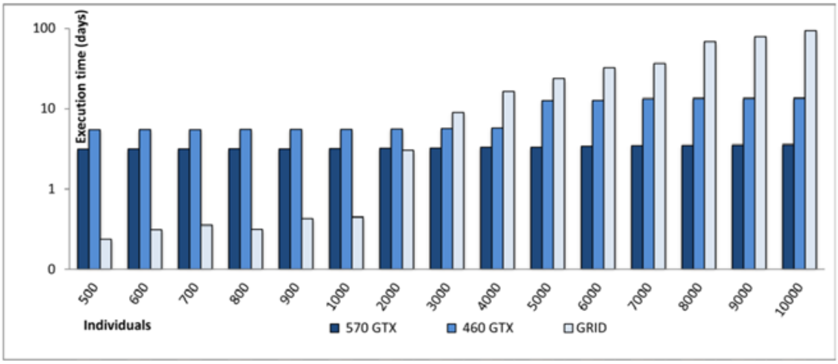


**Fig. 10** Execution times comparative (y- axis) for 500 generations and different number of individuals (x- axis)

## 5.2 Performance Analysis

Figure 10 shows a comparative graph between the different execution times of the platforms analyzed in the present chapter. Y-axis represents the execution time measure in days and, X-axis shows the different amount of individuals in each execution. In this graph we use fundamental analysis. It must be noticed that the grid Boinc version actually implements a different selection algorithm, so we must take in consideration that execution times on the GPU are taking benefit of that fact. We can observe that the grid system becomes inefficient around the 4000 individuals, because we consider that more than 10 days of execution starts to be too large times. Nevertheless, the GPU system remains with a little increments throughout the test (all the time bellow the 10 days).

With the same tests used in Figure 10, we have develop a speed-up graphic. Figure 11 shows the speed-up between the GPU and the grid, and also the reverse form. More specifically, the dark lines represent the relations:

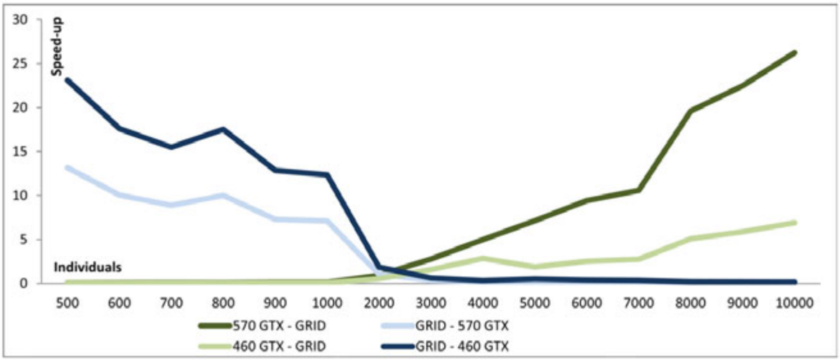


**Fig. 11** Speed-up on the differents plataforms (y-axis)with variable number of individuals (x-axis) and 500 generations

570GTX/grid and grid/570GTX. The clear lines represent the relations: 460GTX/grid and the grid/460GTX. X-axis represents the number of individuals and Y-axis shows the speed-up.

The first thing that takes our attention is that both platforms could be useful in differents periods. At the beginning, the Boinc system has a better behavior than the GPU, but this performance vanish when the number of individuals in a population rises. Beyond 2000 individuals the GPU/grid speed-up grows up constantly. Maybe the critical point (where both technologies have the same performance) should be shifted to the right because the GPU uses a different algorithm, however the trend to raise the speed-up by the GPU is clear. After, we have analyzed the above figures; we can conclude that the grid system works fine when the number of individuals is not too high.

## *5.3 Final Conclusions*

As we have already mentioned, the performance of investment decisions in stock markets is influenced by a huge number of variables related to macroeconomic, companies or market information that are difficult to analyze and even to follow since nowadays we have access to all this information (that is continuously changing) in real time. In addition, professionals making investment decisions suffer a high degree of stress due to the impressive amounts of money they manage. This factor may cause a bias in the analysis of the information by the trader. Next we summarize some of the ideas outlined in this chapter:

- The use of mechanical trading systems copes, at least partially, with these difficulties, since it avoids psychological reactions of traders while allowing managing a huge amount of realtime data.

- The exponential growing complexity of the investment problem related with the number of factors affecting the investment performance makes it necessary to count with powerful algorithmic tools to deal with the selection of indicators and parameters from the universe of existing economic and company variables and threshold values.
- GAs offer a powerful and fast search capacity due to its ability of processing information in a parallel way and the intelligent mechanism that is driving its functioning.
- For dealing with intra day or daily investment decisions for a big number of stocks is vital to speed up the GA process, to get in time good results. For this purpose, we carry out an innovative implementation of the GA that is fine-tuning the trading system, by means of using parallel computer architectures.
- *Boinc* from *Berkeley* is a mature platform that achieve great results using a lot of computers. We can use old computers or the volunteer system to cheapen the costs and we will obtain a powerful grid with few resources. The implementation of an application in Boinc requires a good knowledge of information technologies, this means that it is a tool with high curve of learning.
- *Jacket* of *Accelereyes* is a tool that brings results, recommended to decrease the difficulty when programming on GPU.
- By implementing the code in the GPU, the selection function no longer is a bottleneck and the total execution time has been reduced in a 90% if compared to the CPU version with the same selection algorithm, and to a 97% if compared to the original algorithm.
- There is a limit in the number of individuals implemented on the GPU of around 400000 individuals, where performance drops sharply. This figure is architecture dependant.
- The number of generations will influence the GPU execution time due to memory overload issues. Each additional generation suppose an extra cost, that after a great number of generations will have repercussions on the final execution time.
- For a reduced number of generations and more than 2000 individuals the speed-up of the execution in the GPU will be higher, and that for a higher number of generations the improvement margin will be reduce.

**Acknowledgements.** The authors thanks to Francisco Fernández-de-Vega and Daniel Lombraña from Universidad de Extremadura (Spain) for helping with the initial BOINC implementation. Diego Martín and Javier Villanueva (Felipe II University College of Aranjuez) help us with the configuration of the Boinc Server. This work has been partially supported by Spanish Government grants Avanza Competitividad I+D+i TSI-020100-2010-962, INNPACTO IPT-2011-1198-430000, and TIN 2008-00508.

## References

1. Allen, F., Karjalainen, R.: Using genetic algorithms to find technical trading rules. Journal of Financial Economics 51(2), 245–271 (1999)
2. Bali, T.G., Demirtas, O., Tehranian, H.: Aggregate earnings, firm-level earnings, and expected stock returns. JFQA 43(3), 657–684 (2008)
3. Banzhaf, W., Harding, S., Langdon, W.B., Wilson, G.: Accelerating genetic programming through graphics processing units. In: Genetic Programming Theory and Practice VI, pp. 1–19. Springer, Heidelberg (2009)
4. Basu, S.: The investment performance of common stocks in relation to their price-earnings ratios: A test of the efficient market hypothesis. Journal of Finance 32, 663–682 (1977)
5. Bodas-Sagi, D.J., Fernández, P., Hidalgo, J.I., Soltero, F.J., Risco-Martín, J.L.: Multiobjective optimization of technical market indicators. In: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, GECCO 2009, pp. 1999–2004. ACM, New York (2009)
6. Campbell, J.Y., Yogo, M.: Efficient tests of stock return predictability. Journal of Financial Economics 81, 27–60 (2006)
7. Cantú-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Norwell (2000)
8. Chan, L.K.C., Hamao, Y., Lakonishok, R.: Fundamentals and Stock Returns in Japan, 1739–1764 (December 1991)
9. Cole, N., Desell, T., Lombraña González, D., Fernández de Vega, F., Magdon-Ismail, M., Newberg, H., Szymanski, B., Varela, C.: Evolutionary Algorithms on Volunteer Computing Platforms: The milkyWay@Home Project. In: Fernández de Vega, F., Cantú-Paz, E. (eds.) Parallel and Distributed Computational Intelligence. SCI, vol. 269, pp. 63–90. Springer, Heidelberg (2010)
10. Contreras, I., Jiang, Y., Hidalgo, J.I., Núñez-Letamendia, L.: Using a gpu-cpu architecture to speed up a ga-based real-time system for trading the stock market. Soft Computing - A Fusion of Foundations, Methodologies and Applications, 1–13 (2011)
11. Ellenby, J.: (1979), http://www.thegridsystems.org/
12. Fama, E.F., French, K.R.: Business conditions and expected returns on stocks and bonds. Journal of Financial Economics 25, 23–49 (1989)
13. Fama, E.F., French, K.R.: The cross-section of expected stock returns. Journal of Finance 47(2), 427–465 (1992)
14. Fernández-Blanco, P., Bodas-Sagi, D.J., Soltero, F.J., Hidalgo, J.I.: Technical market indicators optimization using evolutionary algorithms. In: Ryan, C., Keijzer, M. (eds.) GECCO (Companion), pp. 1851–1858. ACM (2008)
15. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning, 1st edn. Addison-Wesley Longman Publishing Co., Boston (1989)
16. Lombraña Gonzalez, D., Ferná de Vega, F., Trujillo, L., Olague, G., Araujo, L., Castillo, P.A., Merelo Guervós, J.J., Sharman, K.: Increasing gp computing power for free via desktop grid computing and virtualization. In: El Baz, D., Spies, F., Gross, T. (eds.) PDP, pp. 419–423. IEEE Computer Society (2009)

17. Jiang, Y., Núñez, L.: Efficient market hypothesis or adaptive market hypothesis? a test with the combination of technical and fundamental analysis. In: Proceedings of the 15th International Conference on Computing in Economics and Finance, The Society for Computational Economics, University of Technology, Sydney, Australia (2009)
18. Krüger, F., Maitre, O., Jiménez, S., Baumes, L., Collet, P.: Speedups Between 70 and 120 for a Generic Local Search (Memetic) Algorithm on a Single GPGPU Chip. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcazar, A.I., Goh, C.-K., Merelo, J.J., Neri, F., Preuß, M., Togelius, J., Yannakakis, G.N. (eds.) EvoApplicatons 2010. LNCS, vol. 6024, pp. 501–511. Springer, Heidelberg (2010)
19. Langdon, W.B.: A fast high quality pseudo random number generator for nvidia cuda. In: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, GECCO 2009, pp. 2511–2514. ACM, New York (2009)
20. Maitre, O., Baumes, L., Lachiche, N., Corma, A., Collet, P.: Coarse grain parallelization of evolutionary algorithms on gpgpu cards with easea. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, GECCO 2009, pp. 1403–1410. ACM, New York (2009)
21. Miller, B.L., Goldberg, D.E.: Genetic algorithms, tournament selection, and the effects of noise. Complex Systems 9, 193–212 (1995)
22. Munawar, A., Wahib, M., Munetomo, M., Akama, K.: Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia cuda framework. Genetic Programming and Evolvable Machines 10, 391–415 (2009)
23. Núñez, L.: Trading systems designed by genetic algorithms. Managerial Finance 28, 87–106 (2002)
24. Núñez, L.: Fitting the control parameters of a genetic algorithm: an application to technical trading systems design. European Journal of Operational Research 179, 847–868 (2007)
25. Núñez, L., Pacheco, J., Casado, S.: Applying genetic algorithms to wall street. Int. J. Data Mining, Modelling and Management (2011) (forthcoming: in press)
26. Pospichal, P., Jaros, J., Schwarz, J.: Parallel Genetic Algorithm on the CUDA Architecture. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcazar, A.I., Goh, C.-K., Merelo, J.J., Neri, F., Preuß, M., Togelius, J., Yannakakis, G.N. (eds.) EvoApplicatons 2010 Part I. LNCS, vol. 6024, pp. 442–451. Springer, Heidelberg (2010)
27. Reinganum, M.: Selecting superior securities charlottesville. the tesearch foundation of the institute of chartered financial analysts. Technical report, The Tesearch foundation of the institute of Chartered Financial Analysts (1988)
28. Ritter, J.R.: Behavioral finance. Pacific-Basin Finance Journal 11(4), 429–437 (2003)
29. Zhang, S., He, Z.: Implementation of Parallel Genetic Algorithm Based on Cuda. In: Cai, Z., Li, Z., Kang, Z., Liu, Y. (eds.) ISICA 2009. LNCS, vol. 5821, pp. 24–30. Springer, Heidelberg (2009)

# A Knowledge-Based Operator for a Genetic Algorithm which Optimizes the Distribution of Sparse Matrix Data

Una-May O'Reilly, Nadya Bliss, Sanjeev Mohindra,
Julie Mullen, and Eric Robinson

**Abstract.** We present the Hogs and Slackers genetic algorithm (GA) which addresses the problem of improving the parallelization efficiency of sparse matrix computations by optimally distributing blocks of matrices data. The performance of a distribution is sensitive to the non-zero patterns in the data, the algorithm, and the hardware architecture. In a candidate distributions the *Hogs and Slackers GA* identifies processors with many operations – *hogs*, and processors with fewer operations – *slackers*. Its intelligent operation-balancing mutation operator then swaps data blocks between hogs and slackers to explore a new data distribution. We show that the *Hogs and Slackers GA* performs better than a baseline GA. We demonstrate *Hogs and Slackers GA*'s optimization capability with an architecture study of varied network and memory bandwidth and latency.[1]

## 1 Introduction

As processor speeds increase and dye sizes shrink, there is a growing demand to do increasingly complex computations on machines that were once extremely limited in terms of computational power. One area where this can be seen is in the high performance embedded computing imaging devices found in surveillance systems. Previously, these devices were only expected to handle the image processing capabilities, and the resulting images were post-processed by outside machines. A trend towards moving post-processing capabilites onto the imaging devices themselves now exists.

Una-May O'Reilly
Computer Science and Artificial Intelligence Laboratory,
Massachusetts Institute of Technology, Cambridge, USA

Nadya Bliss · Sanjeev Mohindra · Julie Mullen · Eric Robinson
Lincoln Laboratory, Massachuestts Institute of Technology, Cambridge, USA

[1] This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

F.F. de Vega et al. (Eds.): Parallel Architectures & Bioinspired Algorithms, SCI 415, pp. 219–241.
springerlink.com

Largely, this post-processing consists of entity extraction, a relatively efficient process on these devices, as well as determining the relations between entities and properties of those relations, a much more challenging problem. This second problem corresponds to a graph processing algorithm. Typically, these graphs can grow to be very large, but also remain very sparse. These types of computations present significant challenges to modern parallel architectures, which are typically not designed to perform them well. The irregular data access patterns of these computations being of primary concern.

This paper examines these challenging graph problems. Rather than using a typical graph structure, the duality between graphs and sparse adjacency matrices is exploited. The graph problems are recast as algebraic operations on sparse matrices. By studying and optimizing these operations, significant performance increases can be demonstrated that help close the gap between the desired and actual efficiency of parallel architectures on these algorithms.

In this contribution, a software optimization strategy is taken that maps the sparse matrix data in an efficient way across the processors to resolve, as much as possible, the irregular data access in local memory read and write accesses and across network communications. Given irregular data access as the main source of the poor performance, there are three interconnected factors that influence how the data should be mapped. These are: the location of the non-zero data in each matrix, the data movements dictated by the individual matrix operations (i.e. kernels) within the algorithm, and the parameters of the hardware which dictate the cost of computation operations and memory access (both local and network transfer operations).

The realization of this optimization necessitates the design and implemention of a framework, the Mapping and Optimization Runtime Environment (MORE), for measuring and optimizing the performance of sparse algebra kernels such as matrix multiply. As shown in Figure 1, at a high level, MORE enters the computation flow when, within a program, a sparse matrix operation is encountered and the matrix operands are tagged for distribution. The data mapping component within MORE then maps processors to continuous blocks it designates within each matrix and passes these to the operations analysis module. The operations analysis module, working from the program parse tree and maps, generates a dependency graph that expresses the dependencies between all network, local memory and compute operations comprising the computation. The dependency graph, being a sufficiently detailed description of the computation for estimating its execution time, becomes the input to MORE's final module, the simulator. The simulator, consulting a hardware model matched to the parameters of a target platform, processes the dependency graph and computes how long each parallel execution stage takes. It presents the sum of these times as MORE's output. In the language of signal processing, commonly used in high performance embedded computing, this simulated execution time is referred to as the "execution latency" of the kernel. The execution time can be used, along with a count of the number of operations in the kernel, to compute the overall OP/s, the rate of floating point operations per second.

MORE also has an optimization component. When this component is enabled, a genetic algorithm is engaged in lieu of the mapping module. The goal of the genetic

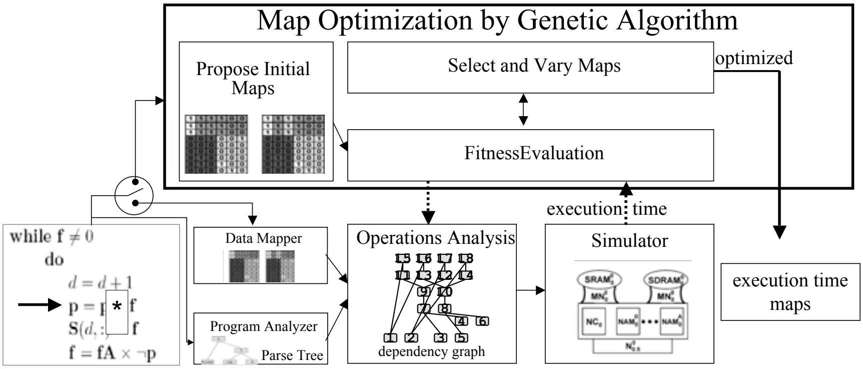


**Fig. 1** The flow and components of the MORE framework. See Introduction and Section 5 for details.

algorithm is to search for matrix maps that result in the minimum execution latency (or maximum OP/s) of the kernel. At a high level, the genetic algorithm module "wraps around" the analysis and simulator modules to repeatedly use them to test the execution time of the maps that the genetic algorithm encodes as a genome in lieu of the data mapping component. The execution time of a map is used as the fitness score of that map's genome. The genome representation of the genetic algorithm is a straightforward processor-to-block pairings. The output of the genetic algorithm optimization is the best set of maps it finds along with the execution time of the kernel.

While the artificial intelligence-based interpretation of a genetic algorithm positions it as a "weak method" ([21]) that is notable for its generality, it has long been recognized ([12]) that practitioners want to bring all available knowledge to bear in solving an optimization problem. In this domain, both the source of the performance gap – irregular data reference – and the factors that influence it – data locality, kernel, and hardware specifications – are well understood. Therefore the optimization module of the MORE framework employs the *Hogs and Slackers GA* and functions as a knowledge-based algorithm. It incorporates a deeper insight, shown graphically in Figure 2, searching for an ideal parallelization that suitably balances the entire computation across processors.

Drawing upon the domain knowledge in order to better optimize the mapping is not trivial. However, due to the fact that the MORE operations analysis module generates a dependency graph, it is possible to count how many operations are assigned to each processor and calculate their balance. This measurement is an explicit surrogate for the parallelization concept expressed in Figure 2. Intermediate results generated by the genetic algorithm help to show the relationship between operations balance and the genetic algorithm's search for maps yielding higher performance. A generation by generation analysis uncovers how the different candidate maps of the genetic algorithm explore balance of operations. It is apparent that, as the genetic algorithm continuously improves OP/s, its superior maps also hone in on a narrow balance interval.

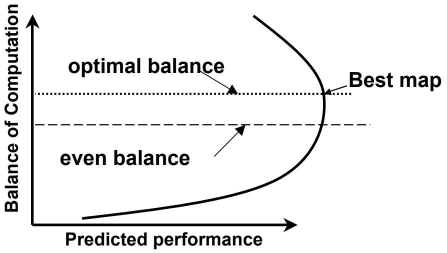


**Fig. 2** Knowledge of parallelization suggests a knee in the performance curve in terms of the balance of the computation across processors. The best maps lie at the point of *ideal* balance that, for a sparse matrix kernel, is *not* at the point where the computation is *evenly*, ie. *perfectly* balanced.

Determing that the operations balance "sweet spot" coincides with the best performance prompts the introduction of operations balance information into the genetic algorithm so that it can be exploited. How to properly introduce this is not obvious due to the fact that the genetic algorithm, via its genome encoding and genetic operators, works on the level of blocks and processor assignments, not operations. Changing the assignment of a block from one processor to another affects all the data in the block and this has a macroscopic and indirect effect on operations balance. Each block is also varies with respect to where its non-zero entries are and how they match up with the non-zero entries of other matrices involved in the kernel.

The *Hogs and Slackers GA* incorporates the insight that the search must find an ideal parallelization that suitably balances the entire computation across processors. The *Hogs and Slackers GA* references the computation's distribution of memory operations across processors in order to guide a genetic mutation operator, BALANCINGMU, to *intelligently* choose processors for block exchange. The BALANCINGMU operator ranks all the processors by either the number or cumulative size of their memory operations. High-ranked processors are "hogs" and low-ranked ones are "slackers". The mutation operator trades blocks assigned to hogs with those assigned to slackers. When tested against a naive block mutation operator, the BALANCINGMU operator is superior. This advantage is demonstrated and then the *Hogs and Slackers GA* is used to study a range of 3 hardware models with parameterized network bandwidth. The study shows that, on different architectures, the genetic algorithm provides consistent and significant optimization over conventional maps.

This contribution is organized as follows: In Section 2 more details of the motivation to study graph algorithms and expand on the graph matrix duality are provided. In Section 3 observations are presented that support the assessment that a performance gap between desired and present performance of sparse matrix computation exists. In Section 4 the source of the inferior performance is examined and the complexity involved in trying to understand the interaction of the three factors – nonzero data location, algorithm, and hardware parameters – that affect it is explored. In Section 5 the MORE framework for studying and optimizing a sparse matrix computations is described. In Section 6 a complete description of the *Hogs and Slackers GA* is provided. In Section 7 experimental results are presented related to the knowledge-based BalancingMu operator and a study of the optimized performance of a sparse matrix multiply kernel on a range of architectures. In Section 8 related work is presented. In Section 9 a conclusion is presented and future work is described.

## 2 Graph Algorithms for Decision Support Systems

High performance embedded computing is transitioning from solely serving the domain of signal processing to additionally serving the domains of knowledge extraction and decision support. This can be seen in a "system of systems" architecture where the lower level information streams from multiple computing systems with various sensing modalities and is integrated to provide unified object inference and global situation assessment. In many decision support systems, graphs play a role as important information elements. For example, graphs are used when examining social networks, determining sensor net coverage, and representing the Bayesian networks that are used to fuse images. Some general graph algorithms are edge (or vertex) betweenness centrality, Bayesian belief propagation, minimal spanning trees, and single source shortest path. Common real-world situations typically yield extremely large graphs with relatively few edges (on the order of thousands or millions of vertices and tens or hundreeds of edges, on average, per vertex).

It has previously been shown that it is efficient to cast graph algorithms as sparse matrix operations [18]. Using this graph-matrix duality, a graph is represented with a two-dimensional adjacency matrix. An entry in this matrix at row *i* and column *j* denotes an edge from vertex with id *i* to vertex with id *j*. Common operations on graphs can be implemented to take advantage of this matrix representation by using common linear algebra operations. For example, raising the adjacency matrix to some power *p* yields a resulting matrix where an entry *v* in row *i* and column *j*

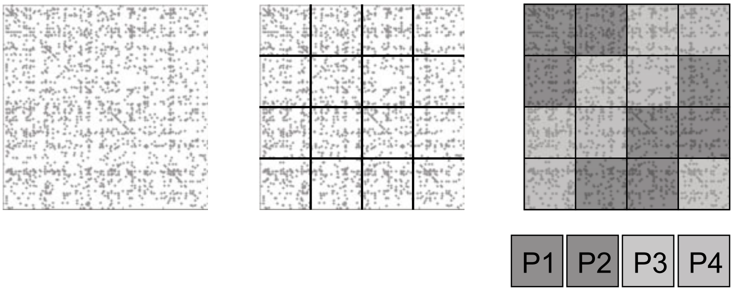


**Fig. 3** The matrix map concept. On the left a sparse matrix is shown. In the middle a grid is designated over the matrix to delineate blocks of matrix elements. On the right each block is assigned to a processor. Which processor is shown by coloring the block and providing a legend below the map. This is an example of an Anti-Diagonal Block Cyclic map. Along each diagonal line of the matrix, the processor assignment is cycled. This map is designed for efficient dense matrix maps. As with other conventional maps, its grid creates square blocks.

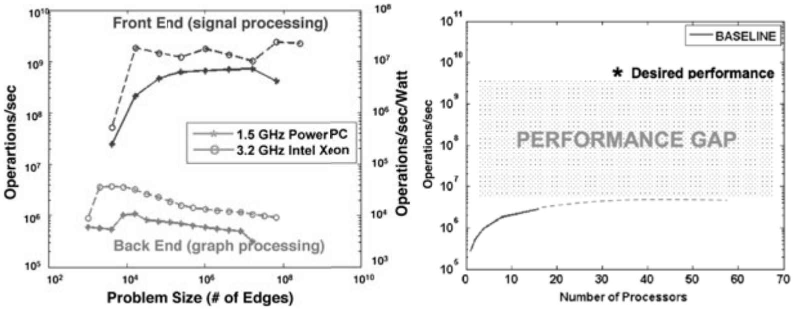
indicates that $v$ distinct traversals through $p$ edges exist between vertices $i$ and $j$. An example of the use of linear algrebra operations to compute vertex betweenness centrality can be found in [28].

In addition to the rigour and precision of its mathematical formalism, the matrix representation also supports a straightforward means of parallelization through the distribution of blocks of the matrix elements across processors of the distributed architecture. As a first order approximation, the efficiency of a distributed matrix computation depends on how its operands, the matrices, are mapped. Mappings such as anti-diagonal block cyclic, see Figure 3, have been established as practical and efficient.

## 3 The Sparse Matrix Performance Gap

Unfortunately challenges arise when the graphs (or matrices) are sparse. For example, see Figure 4(a), which compares sparse and dense performance for serial matrix multiply implementation. Dense data is associated with front end processing, i.e. signal processing, whereas sparse data is associated with back end processing, i.e. graph processing for decision support. On two different hardware platforms, a 1.5 GHz Power PC and a 3.2 GHz Intel Xeon, rgw number of edges is increased while the sparsity is held constant. After 10,000 edges the performance of both platforms (OP/s) with sparse data degrades. An efficiency differential of approximately $10^3$ is observed. Figure 4(b) shows that an edge betweenness centrality algorithm does not extract any performance advantage from being run on more than 30 processors. In addition, this algorithm's performance falls radically short of performance targets set by the decision support application in question.

There are a number of potential methods to resolve this performance gap. Specifications can be formed that will drive future hardware design. The specifications can consider the relative latencies and bandwidths within the memory hierarchy and may factor in payloads costs. An alterantive, and possibly a more powerful,



(a) (b)

**Fig. 4** (a) Comparison between dense and sparse matrix data with the same serial matrix multiply algorithm. (b) Scaling performance of an edge betweenness centrality algorithm when the number of available processors is increased.

approach is a hardware and software co-design process that supports the concurrent investigation of hardware design choices and their implications on sparse algebra computational performance.

These approaches have promise but entail a rather long development process. They do not address what can be done for architectures set be deployed in the short term and that will likely have tens to hundreds of processing nodes. In addition, new technologies for compute nodes, memory, and networks cannot be expected to be convergent so we should anticipate hardware that is a combination of technologies at different maturities. The short term will offer different architectural trade-offs as hardware designers explore design ideas in the relatively new space of 3D component packing and the relatively recent issues arising from power demands.

In the short term, software optimization holds the best promise. In the dense matrix case, the general strategy of mapping the data before the computation starts is sound. However, in the sparse matrix case, the mapping can not be solved in as straight forward a manner.

## 4 Problem Discussion: Mapping Sparse Matrices

At its root, poor sparse algebra computational performance stems from the internal representation of sparse matrices. Only the non-zero elements of sparse matrices are stored. They are usually stored in compressed sparse column (CSC), compressed sparse row (CSR), or tuple format. Sparse matrix kernels must compute only the interactions of non-zero elements. This makes them more computationally expensive. Each storage format incurs a particular cost for column and row access, and for the insertion, deletion, element transpose and a find-element primitives. See [7, 15, 29] for more details. More notably, the compressed storage formats leads to sparse matrix algorithms generating irregular data references, i.e read, write, or transfer operations. These operation may occur at any level of the memory hierarchy: either local to a processor or the inter-processor block transfers of data through the network. A memory reference can have a unit cost based on whether it is local or not. Each local or non-local reference cost can also be weighted by its time, i.e. “latency”, or the size of the data payload.

In a dense matrix mapping approach, where this irregularity does not emerge, it is sufficient to use mappings of the processors that cycle processors through the blocks. See Figure 5 for examples. The anti-diagonal block cyclic map shown in Figure 3 is one choice, along with the block cyclic map. Maps can be stored in an “atlas” that is referenced at run time ([32]).

In a sparse mapping approach, the intention is to distribute the data of the matrices over the processor architecture in a manner that reduces the irregularity of data reference. This goal makes the mapping problem much more complicated and conventional maps are not sufficient ([33]). It is necessary to be able to map any block to any processors. In addition, there are now three primary factors that influence how the data should be mapped – the location of the non-zero data in a matrix, the data movements dictated by the algorithm, and the parameters of the hardware which

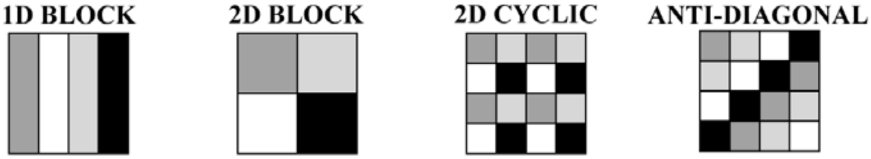


**Fig. 5** Conventional block mappings for dense matrices.

dictate the time cost of computation operations and data access, retrieval and network transfer operations. Studying any one of these factors in isolation is difficult. For example, there are multiple sparse data patterns that commonly arise. Figure 6 presents the patterns considered as part of this contribution. The difficulty of understanding the interactions of all three factors in order to determine an efficient map is very high despite knowing, in general, that some ideal balance of the computation across the processors is desired. The approach to this problem taken here involves the design and implementation of the MORE framework for evaluating and optimizing sparse matrix computations.

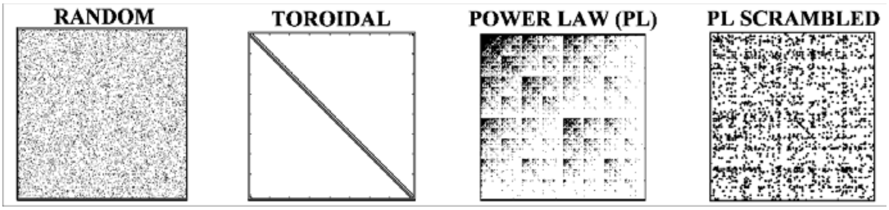


**Fig. 6** Different patterns of sparsity observed in the graph processing applications of our interest.

## 5 The MORE Framework

As introduced earlier (see also Figure 1), the Mapping and Optimization Runtime Environment (MORE) framework measures and improves the performance of a sparse matrix computation on a distributed system. The framework is implemented in pMatlab ([2]) and is supported by the software of the pMapper automatic matrix mapping project ([32]), both developed at Massachuesttes Institute of Technology Lincoln Laboratory. pMatlab is a parallel toolbox for Matlab developed that leverages MatlabMPI. pMatlab provides program mechanisms for tagging array data for distribution and distributing or aggregating such data to and from multiple processors. The pMapper project supports automatic mapping of the programmer-tagged dense arrays for a select set of overloaded matrix operations. It supports lazy evaluation of distributed array operations: execution is delayed until a result is required.

The framework has four principle components: a mapper, a program analyzer, an operations analyzer and a performance simulator. The *Hogs and Slackers GA* is an auxiliary map optimization component that can be switched on and which integrates with MORE's framework when maps need to be automatically generated rather than specified by a programmer. The *Hogs and Slackers GA* accesses the parse tree from the program analyzer and the dependency graph from the operations analyzer. It also uses the performance simulator.

## *5.1 The Mapper*

The mapping component facilitates better sparse matrix kernel performance with its implementation of fine-grained maps. It permits much smaller blocks of data than does the dense mapping case. It allows any processor to be assigned to any block because regularity in their distribution is likely to be detrimental to performance.

The mapper is able to support the fine grained maps without significant increase to index computation cost by using pMapper's underlying index representation scheme: Processor Indexed Tagged FAmiLy of Line Segments (PITFALLS)[26]. The advantage of using PITFALLS is that it allows for efficient redistribution of data by providing fast computation of indices local to each processor and a direct means of calclulating of messages that have to be sent between processors when changing maps.

A map, see Figure 3 for an example, consists of two components: the grid describing how the matrix is subdivided into blocks, and a list of processor assignments for the blocks. The genetic algorithm that maps matrix operands searches over a list of potential grid choices and all possible processor assignments for blocks. It repeatedly uses the program analyzer and performance simulator to obtain fitness scores for its candidate maps.

A map does not specify routing information. However, once maps are defined, the set of communication operations can be enumerated. The mapping component, when not in optimization mode, greedily routes by the shortest path in terms of communication operations. When in optimization mode, the genetic algorithm can run in nested mode where it enables an "inner" genetic algorithm to search for the best routes for the map. The performance of this nested genetic algorithm is reported in [33]. In this contribution, to focus on the knowledge-based aspects of mapping, all experiments are executed without nested-mode. Routes are chosen greedily.

The mapping component can either use maps supplied by the programmer or run a map optimization algorithm that performs search-and-test-based optimization implemented by a genetic algorithm.

## *5.2 The Program Analyzer*

The program analysis component converts the user code into a parse tree, T, to be analyzed. It uses a lazy evaluation strategy, not sending code for analysis until it is required by the user. This allows for the analysis to take in as much of the context of the code as possible and can lead to improved performance.

## 5.3 The Operations Analyzer

The operations analysis component performs fine-grained dependency analysis. It takes the maps supplied by the mapping component, the parse tree of the program to the current point of execution, and the hardware programming model and constructs a dependency graph. The dependency graph is a directed acyclic graph (DAG) of operations that are topologically sorted to identify parallel stages of the computation. Operations in the dependency graph are classified as either compute, network or local memory operations. Each dependency graph node identifies its origin in the program parse tree and its topological level in the dependency graph. Each node for a network and memory operation has a chunk size which lists the number and sizes of the accesses associated with it. Each memory operation node has a type identifier specifying whether it is a read or write.

## 5.4 The Performance Simulator

The performance simulator is designed to consult a parameterized model of a target architecture. By modifying these parameters, the effects of varying rates and latencies at various levels in the memory hierarchy on sparse computations can be examined. This model also permits experimentation with both heterogeneous or homogenous processor configurations and the exploration of power-performance trade-offs. The hardware model specification is represented logically through the use of a Kuck diagram ([19]). The Kuck notation provides a clear way of describing a hardware architecture along with the memory and communication hierarchy. It is can be used to provide both a high-level and a detailed physical description of the architecture. Figure 7 shows elements of a Kuck diagram for a system with a three level hierarchy.

The architecture model, for each processing element, records the computation rate and latency of the element, the operations allowed by the element and their efficiency, local memory capacity, bandwidth, and latency, and power requirements. For a network element it records the number of nodes, the inter-node bandwidths and latencies, routes, including paths and costs, and a routing policy. The shared memory network is recorded with its size, bandwidth, and latency to each element, routes, and a routing policy. See Table 1 for a brief example.

The performance simulator determines the performance of the program on the intended hardware. It takes a dependency graph of the program and network routes, R, along with a model of the hardware, H, and computes the time required to run the operations specified by the dependency graph on that hardware. Note that this allows one to easily interchange the hardware model and keep in place the remaining structure, providing easy transition to analysis of future hardware designs.

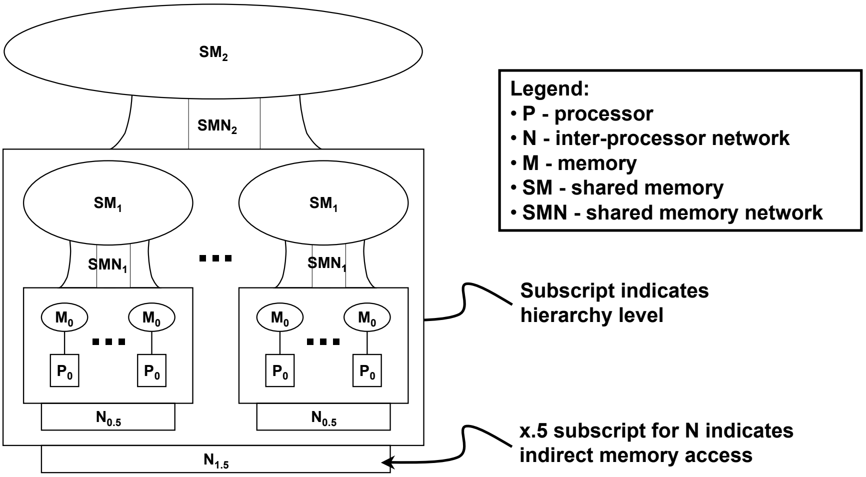

SM$_2$
SMN$_2$
SM$_1$
SM$_1$
SMN$_1$
SMN$_1$
M$_0$
P$_0$
N$_{0.5}$
N$_{1.5}$
Legend:
• P - processor
• N - inter-processor network
• M - memory
• SM - shared memory
• SMN - shared memory network
Subscript indicates hierarchy level
x.5 subscript for N indicates indirect memory access

**Fig. 7** Example 2-Level Memory Hierarchy in Kuck Notation.

**Table 1** Performance Simulator: Example parameters

| Parameter | Symbol | Unit |
|---|---|---|
| Number of processing cores | $N_P$ | N/A |
| Processor $P_i$ speed | $R_P$ | (FL)OPs/second |
| Processor $P_i$ latency | $L_P$ | second |
| Processor $P_i$ efficiency on operation $k$ | $E_{P_{i,k}}$ | N/A |
| Memory, including remote, bandwidth | $\mathbf{R}$ | bytes/second |
| Memory, including remote, latency | $\mathbf{L}$ | second |
| Size of a single data element of data type $T$ | $S_T$ | bytes |

For the purposes of this paper, a *topological simulator* is considered. In a topological simulator, the dependency graph is sorted topologically based on the dependences. This organizes the nodes in the graph into distinct levels. The simulation time for each level, $L$, can then be evaluated, where the operations within $L$ may be run concurrently according to $H$. The total simulation time is the sum of the simulation time for all $L$.

## 6 MORE Map Optimization: The *Hogs and Slackers GA*

MORE uses a genetic algorithm because it is simulation-based and little is known about the nature of the solution space: whether it is flat in terms of mapping options, rugged or multi-modal. The goal of the genetic algorithm is to return a set of maps such that the simulated execution time of the related dependency graph is minimized.

Formally, given the parse tree, $\mathsf{T}$, and the hardware model, $\mathsf{H}$, the genetic algorithm identifies a set of maps, $\mathsf{M}$ with corresponding set of routes, $\mathsf{R}$, such that the following objective function, $f$ is satisfied:

$$argmin_{\mathsf{M},\mathsf{R}} f(\mathsf{T},\mathsf{H},\mathsf{M},\mathsf{R}) \tag{1}$$

The function $f$ returns a duration of execution. Evaluation of $f$ is performed by simulating the dependency graph of $\mathsf{T}$ using the hardware model $\mathsf{H}$. Optimization results are reported in operations per second (OP/s) by dividing the operations executed by the duration of execution. The minimum size of search space of maps is:

$$S_M = N_P{}^{(B)} \tag{2}$$

where

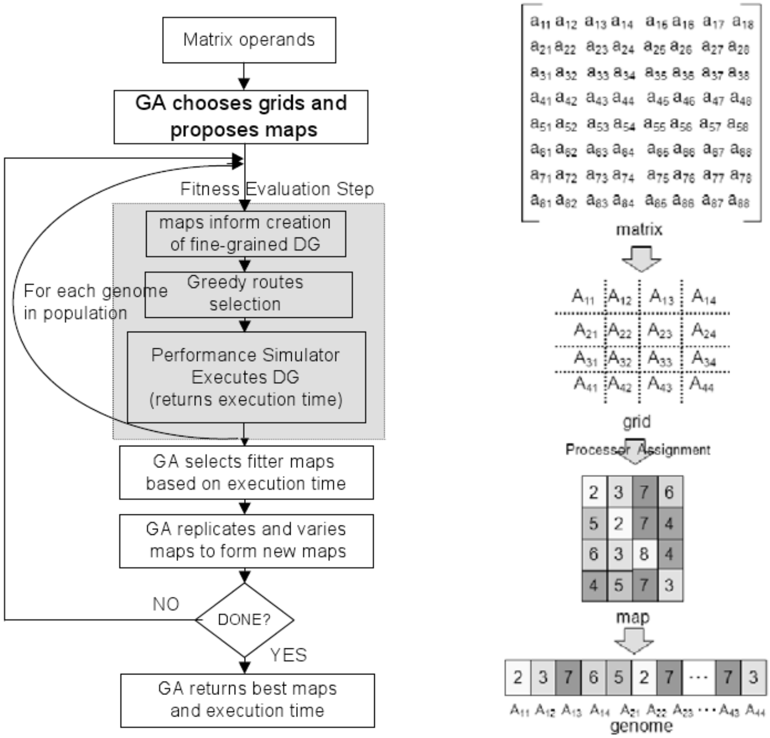
- $N_P$ = number of processing nodes
- $B$ = number of blocks, which is equal to $\sum_{i=1}^{N_M} B_i$ where $N_M$ is the number of matrices or arrays in $T$ and $B_i$ is the number of blocks in matrix $i$.

The genetic algorithm, see Figure 8(a), runs for a specified number of generations. When invoked, it receives the matrix operands of the sparse algebra operator. It starts by creating a random initial population of candidate genomes. Each genome encodes a map for each operand (see Figure 8(b)) and can optionally encode a map for the result. Each map is first set up with a grid which is determined randomly from prespecified options, such as the minimum number of blocks per row or column or some specific grid set. The grid dimensions are linked to the genome for reference. Then, for each grid, processors are assigned an approximately equal number of blocks.

To obtain the fitness score, the genome is passed to the operations analysis module which generates the fine-grained dependency graph (abbreviated as DG in Figure 8(a)) of the computation using the maps and parse tree. The dependency graph is then passed as input to the simulator, with a greedy route selection policy, which returns the execution time.

After the fitness evaluation of the population, the genetic algorithm uses tournament selection, with replacement, to select parents. A fraction of parents undergo both crossover and mutation. Another fraction undergo only mutation. Elitism propagates a small number of current best solutions, usually one or two, without any genetic variation. Upon completion, the genetic algorithm returns the maps of the fittest genome and its execution time.

When route selection is to be co-optimized with mapping, an inner genetic algorithm is triggered after the dependency graph has been generated. All possible routes for the given graph are determined and the inner genetic algorithm does a smaller scale search of them to return the best routes and execution time of the genome, as well as the associated dependency graph and route set. In this contribution, route selection is done greedily. This allows the focus to be placed on the impact of exploiting problem knowledge. As a result, the inner genetic algorithm is not invoked. For results on the nested genetic algorithm see [33]. The genetic algorithm can also


Matrix operands
GA chooses grids and proposes maps
Fitness Evaluation Step
maps inform creation of fine-grained DG
Greedy routes selection
Performance Simulator Executes DG (returns execution time)
For each genome in population
GA selects fitter maps based on execution time
GA replicates and varies maps to form new maps
DONE?
NO
YES
GA returns best maps and execution time
matrix
A11 A12 A13 A14
A21 A22 A23 A24
A31 A32 A33 A34
A41 A42 A43 A44
grid
Processor Assignment
2 3 7 6
5 2 7 4
6 3 8 4
4 5 7 3
map
2 3 7 6 5 2 7 ··· 7 3
A11 A12 A13 A14 A21 A22 A23 ··· A43 A44
genome

(a) Execution Flow

(b) Genome

**Fig. 8** Genetic Algorithm Overview

optimize multiple objectives. Once again, in this contribution, the capability is not used. See [24] for more details.

## 6.1 Parallel Implementation

The genetic algorithm is well suited to parallelization. In a parallel implementation, fitness evaluation of the population is divided across nodes of a cluster each generation. Each “worker” node receives a slice of the population. For each genome, it computes the dependency graph and runs the simulation. It stores the fitness scores of its sub-population locally. Upon completion of the entire sub-population’s evaluation, it sends the results to the master node where they are aggregated to reassemble a population fitness array. This requires minimal code changes using pMatlab – about 20 lines of code or 1% of the code base. The implementation provided by this contribution is executed on the Lincoln Laboratory computing cluster named LLGrid ([27]) and achieves near linear speedup. The time to pass each worker’s fitness results across the network is significantly lower than the time to evaluate the fitness of one individual.

## 6.2 Genetic Representation and Naive Operators

Figure 8(b) shows that the genome is a linear vector of processor identifiers, one per block. When the grid of a map is allowed to vary across genomes in the population, genomes have different numbers of blocks and vary in length.

The BLOCKXO operator is a uniform crossover operator. It aligns the blocks of the parents' genome and, at each block, probabilistically determines whether to swap the block between the parents. BLOCKXO is only permitted on maps with the same number of blocks because of the philosophical intent to exchange alleles of a gene. For more macroscopic recombination, MAPXO probabilistically exchanges the maps of a matrix operand between the parents. The "naive" RANDMU operator probabilistically changes the processor assignment of a block to a random processor.

## 6.3 Operations Balancing Mutation:BALANCINGMU

In the design of the knowledge-based BALANCINGMU, recall the basic hypothesis: the genetic algorithm will benefit from explicitly varying the computational balance as a means of exploring new maps. In the MORE framework, the available surrogate for computational balance is the "even-ness" or "balance" of operations across processors. These operations (in three types – CPU, memory or network) are countable from the dependency graph because each of its nodes corresponds to a hardware operation and is annotated with its type, the operation payload and what processor executes the operation. The operations balance measure expresses a normalized variation in the distribution of operations (of an operation type) across processors. The balance of a set of maps' memory operations is:

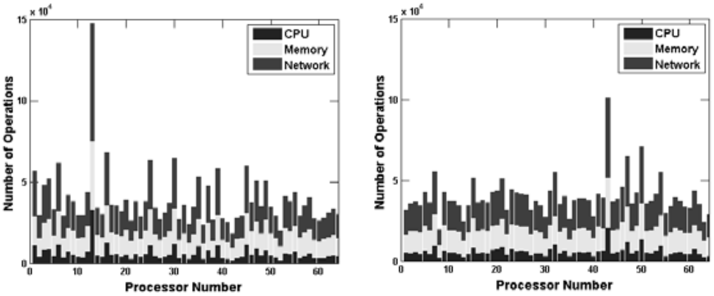
$$bal.memops = \frac{stdev(opcounts.memops)}{mean(opscounts.memops)} \tag{3}$$

CPU or network operations can be substituted for memory operations to similarly derive CPU or network operations balance. To test whether the measure is adequately reflective of computational balance, it was confirmed that there are differences in the CPU, memory and network operations balances in an optimized map when the pattern of sparsity in the matrices and the specific matrix multiply kernels are varied. Table 2 shows balances in a random and optimized map when twp different matrix types – power-law and scrambled power-law – and two different matrix multiply implementations – hybrid (see Section 7.1 for details) and inner product ([13]) – are paired for map optimization.

**Table 2** Balance of CPU, memory and network operations for matrix operands with different sparsity patterns

| Pattern | MatMult | Random<br>(CPU,MEM,NW) | Optimized<br>(CPU,MEM,NW) |
|---|---|---|---|
| Scrambled Power Law | Inner | (0.34,0.16,0.17) | (0.09,0.05,0.05) |
| Scrambled Power Law | Hybrid | (0.38,0.23,0.26) | (0.45,0.19,0.29) |
| Power Law | Inner | (0.91,0.42,0.46) | (0.70,0.30,0.28) |
| Power Law | Hybrid | (0.91,0.55,0.62) | (0.99,0.52,0.68) |

Figure 9(a) and Figure 9(b) show counts of all three operation types for a random (from the first generation) and best of run (from the final generation) map set with scrambled power-law matrices using the hybrid matrix multiply algorithm. In the initial generation, despite blocks being distributed evenly across processors, the operations are quite imbalanced due to the matrix sparsity pattern and the matrix multiply algorithm. This imbalance diminishes in the best individual but, as one would also expect, due to the complex interactions among the network, memory and CPU computations, perfectly even balance does not actually provide the best performance.



(a) Random map (b) Optimized Map

**Fig. 9** CPU, memory and network operation counts.

Despite its indirect influence, changing processor assignments to an entire block of a matrix can control operation re-distribution. A simplified intelligent operations balancing operator, called MICRO_BALANCING_MU, was tested that reassigns only a single block between one of the processors with the highest number of memory operations (the "hog") and one of the processors with the lowest number of memory operations (the "slacker"). On maps selected from later generations of different genetic algorithm runs, this reassignment frequently resulted in a more even operations balance. Accurate control was more frequent when consulting memory or CPU operations counts: 95% to 99% of changes changed balance in the appropriate direction) versus network operations counts (85%). When done in the opposite direction (block reassignment from slacker to hog), the operations balances also consistently became less even. In addition, MICRO_BALANCING_MU frequently generates offspring that are better than the parent (roughly 70% of the time). These results are better than the naive operators – BLOCKXO, MAPXO and RANDMU – whose success rates lie around 10% to 20%.

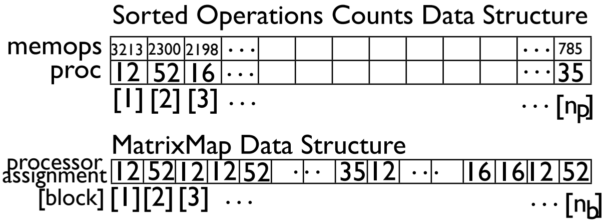
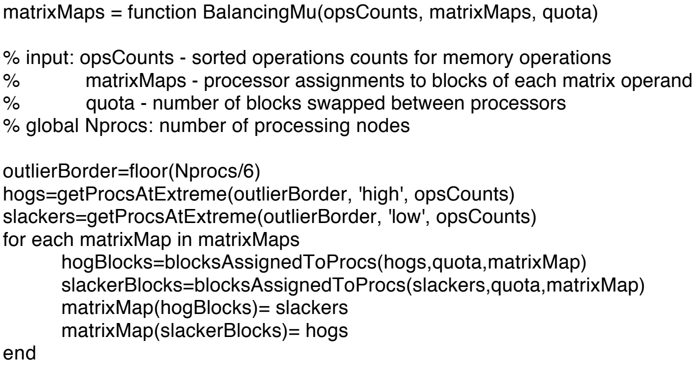


Fig. 10 BALANCINGMU inputs



```
matrixMaps = function BalancingMu(opsCounts, matrixMaps, quota)

% input: opsCounts - sorted operations counts for memory operations
%        matrixMaps - processor assignments to blocks of each matrix operand
%        quota - number of blocks swapped between processors
% global Nprocs: number of processing nodes

outlierBorder=floor(Nprocs/6)
hogs=getProcsAtExtreme(outlierBorder, 'high', opsCounts)
slackers=getProcsAtExtreme(outlierBorder, 'low', opsCounts)
for each matrixMap in matrixMaps
    hogBlocks=blocksAssignedToProcs(hogs,quota,matrixMap)
    slackerBlocks=blocksAssignedToProcs(slackers,quota,matrixMap)
    matrixMap(hogBlocks)= slackers
    matrixMap(slackerBlocks)= hogs
end
```

Fig. 11 Pseudocode of BALANCINGMU

Given these analyses, BALANCINGMU, a more general version of MICRO_BALANCING_MU, was designed. The operator's inputs and pseudocode are shown in Figures 10 and 11. BALANCINGMU is passed the sorted memory operations counts corresponding to the operation nodes of the dependency graph. It selects as a set of hogs or slackers $1/6$ of the total number of processors from the upper or lower extremes respectively. These processors are those whose operation counts are greater or less than approximately one standard deviation from the mean. From among each set, with uniform probability, a fixed number of their blocks are selected. These groups of blocks are then swapped between hogs and slackers.

# 7 Experimental Results

## 7.1 Experimental Setup

In this contribution, the kernel selected for optimization is matrix multiply, the associated codelet is shown in Table 2 (left). The genetic algorithm maps the operands $A$ and $B$. The result $C$ is assigned the map of $A$. Both $A$ and $B$ have either a power-law or scrambled power-law distributions with a non-zero element density of $8/N$ where $N = 1024$ is the number of vertices in the graph (the dimensions of the matrices). An R-MAT power law generator ([5]) was selected to produce $A$ and $B$. The implementation of "matrixMult" is a hybrid inner-outer product algorithm. It alleviates some of the communication load associated with an outer product algorithm and handles distributed sparse data better than Strassen matrix multiply. Similar to an outer product, the algorithm sends $A$'s entries to $B$'s column owners as their row position requires. However, rather than producing a local sum, the products generated are sent immediately to $C$, as with an inner product, and final values are summed once all of the products are gathered.

Because of the need for computational efficiency and fast optimization, the genetic algorithm was run for only 50 generations with a population size of 100. This amounts to searching a very tiny fraction of the search space. The tournament size is set to 5. Elitism preserves the two best genomes each generation. The genetic algorithm has 9 different grids it can randomly assign to a map when it initializes the population. Thus the number of possible combinations of grids for 2 matrices is 81. Each grid has 256 blocks. See Table 2 (right) for a list of possible grids. To select operator probabilities ranges were explored via experimental design. The probability of MapXO was decreased from 0.125 to 0.0 over the first half of the run to foster large exploration steps early. The probability of BlockXO was increased from 0.25 to 0.75 over the run with the likelihood of a block crossover decreasing from 0.1 to 0.01 to foster finer-grained exploration steps later on fewer blocks. RandMu was applied with 0.75 probability throughout the run on each genome while the likelihood of a block being reassigned linearly decreased from 1.0 to 0. BalancingMu was applied frequently at the beginning of a run (selecting a matrix map with 0.75 probability) then its application was decreased linearly (to 0.25 probability) over the course of the run. Over the same interval, the quota of blocks swapped was decreased from 10% of the maximum blocks in the grid to 1%.

**Table 3** Optimized Codelet (left), Grid Dimensions (right)

```
A=rand(N,N,p);
B=rand(N,N,p);
C=zeros(N,N,p);
C=matrixMult(A,B);
eval(C);
```

| Possible Grids | |
|---|---|
| 1 X 256 | 256 X 1 |
| 2 X 128 | 128 X 2 |
| 4 X 64 | 64 X 4 |
| 8 X 32 | 32 X 8 |
| 16 X 16 | |

## 7.2 *Evaluation of* BalancingMu

To evaluate BalancingMu, it was compared (see Table 4) to RandMu, or conventional Anti-Diagonal Block Cyclic (ADBC) maps, which provide the best advantage in dense computation. Fifty runs were executed, each with different matrix data but the same non-zero pattern and density, and their best maps in terms of OP/s were considered. Both mutation operators were teamed with the crossover operators. The mean performance of BalancingMu's best maps was approximately three times better than ADBC. Compared to RandMu, the mean performance improvement is statistically significantly (t-test, p=5.5e-38) with the best of all runs providing a 13% improvement (8.678e9 vs 9.778e9 OP/s). To test whether random swapping would further improve the smart swapping of BalancingMu, a RandSwap operator was added in that swapped a set of blocks chosen at random. The total applications of both swap operators and their blocks quotas were kept equal to the values set when BalancingMu is used alone. The unpaired t-test confirms there is a statistically significant benefit to adding the random swap (p=0.0035) though the best result is, on average, only 1% better.

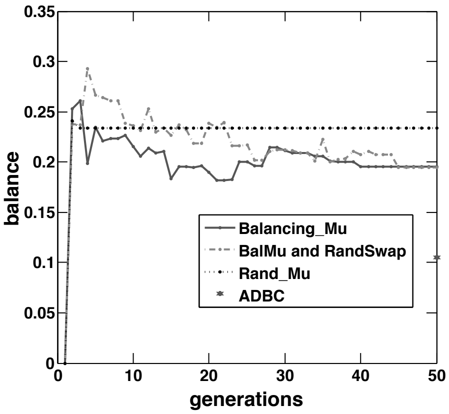
**Table 4** Comparison of BalancingMu to using RandMu or conventional Anti-Diagonal Block Cyclic (ADBC) maps, then RandSwap added.

| | Mean (SD) Best of Run ( OP/s) | Best of Runs (OP/s) | Relative to ADBC |
|---|---|---|---|
| ADBC | 3.17E+09 (6.39E+08) | 3.71E+09 | 1.0 |
| RandMu | 8.06E+09 (3.23E08) | 8.68E+09 | 2.54X |
| BalancingMu | 9.47E+09 (1.70E08) | 9.78E+09 | 2.99X |
| BalancingMu + RandSwap | 9.56E+09 (1.45E08) | 9.92E+09 | 3.01X |

Using the run which provided the best maps over 50 runs for ADBC, BalancingMu, and RandMu respectively, the memory operations balance of the best individual in each generation is presented in Figure 12 (left). While ADBC generates an evenly balanced map in terms of memory operations counts, the ideal distribution of counts (per the optimized map of the BalancingMu run) is slightly uneven. However, if the balance is too uneven, as is the case with the maps generated by the RandMu run, performance is not as good. In general, BalancingMu's behavior is consistent with the goals of its design: to significantly boost the efficiency of the genetic algorithm.

## 7.3 *Hardware Model Parametric Study*

In addition to providing optimization on a single architecture, the MORE framework enables comparative investigation of sparse matrix computation on different



| | |
|---|---|
| Net Latency | 50e-9 s |
| Net Bandwidth | 5e9 bps |
| Mem Latency | 50e-9 s |
| Mem Bandwidth | 12e9 bps |
| CPU Rate | 5e9 ops/s |

**Fig. 12** Left: Balance over a run Right: Baseline hardware model.

architectures. Here, the effects of varying network rate are examined. The baseline model, Figure 12 (right), corresponds to a nominal Cray-like machine with a 4X4X4 torus topology. Like the Cray, it has very high network provisioning. Two variations of it are created: flooding the system with a 10*X* improvement in bandwidth and restricting the bandwidth by $10^{-1}$X.

**Table 5** Comparison of Network Rate Parameterized Hardware Model

| Model | Mean of Runs<br>Absolute OP/s | Relative | Best of Runs<br>Absolute OP/s | Relative | Improvement<br>over ADBC |
|---|---|---|---|---|---|
| TenthX | 4.998E+09 | 0.53 | 5.575E+09 | 0.57 | 2.4X |
| nominal | 9.471E+09 | 1 | 9.777E+09 | 1 | 3.0X |
| TenX | 9.335E+09 | 0.99 | 1.023E+10 | 1.05 | 2.7X |

When the hardware model is constrained (row "TenthX" of Table 5), the best run reaches a solution of 5.575e+09 OP/s, which is only approximately 0.57 times the rate when the model is nominally configured. This is still over two times better than an ADBC map. The lower improvement ratio (2.4:1 vs 3.0:1) compared to the nominal model reflects the difference between the optimized balance point and balance point of the ADBC map. Across the 50 runs, the mean best solution is 0.53 times less efficient than solutions for the nominal model. As expected, the same relative differences between nominal and "flooded", i.e. "TenX", are not observed because the nominal model already has a relatively high network rate. The differences are still statistically significant but smaller. There is larger variance, but a lower mean, in the outcome of the TenX runs (mean best solution = 9.335E+09 OP/s with SD=4.40e08), while the best of runs of the "flooded" network is is better than the nominal: 1.02e10 vs 9.78e10 OP/s). Note that in all of the models the use of *Hogs and Slackers GA* maps provides better performance than the conventional ADBC mapping.

## 8 Related Work

Computational linear algebra has long considered parallel and distributed sparse matrix-dense vector multiplication, as discussed in [8, 22]. More recently sparse matrix-sparse matrix multiplication, such as in [3, 4], has been examined and the old, and previously inefficient, idea of representing a graph as an adjacency matrix has become plausible. Before, most of the work using an adjacency matrix representation revolved around formalizing proofs about graph computations as opposed to performing the computations themselves, e.g. [9, 31, 35]. In contrast, recent work has focused on using mathematical software and algebraic operations to solve various graph problems, e.g. [25, 36, 6, 11, 18]. This contribution is the first to optimize sparse matrix-sparse matrix algebra via a genetic algorithm. The idea of using a genetic algorithm to optimize a distributed algorithm traces back (at least) to [30] who compared it to hill climbing and simulated annealing for a mapping problem defined as "the optimal static allocation of communication processes on distributed memory architectures". Subsequently other work has widened this definition of mapping to include task allocation ([17, 1, 16, 20]), while also considering specific problems as varied as training set parallelism,([10]), and query ordering for sequence analysis, ([34]). [23] uses a GA for distributed sparse matrix Cholesky factoring.The contribution also includes a tree rotate mutation operator that references auxilliary knowledge. A recent example of exploiting domain specific intelligent genetic operators (with complementary genome representation) comes from the single vehicle pickup and delivery problem with time windows in [14].

## 9 Conclusions and Future Work

These results contribute to the study of genetic algorithms as well as optimization of distributed computations. They demonstrate again that knowledge gives a genetic algorithm an advantage. BALANCINGMU is a means by which the genetic algorithm can consider information on one representation level – operations balance, and make changes on another (processor and block assignment) with an effective improvement in performance.

With respect to the present challenge of optimizing a parallelized computation on increasingly complex architectures, the MORE framework serves as a concrete demonstration of a general methodology that will be more frequently necessary in high performance embedded computing. Plans exist to expand the optimization beyond that of one kernel to support entire graph algorithms. The plan is to use MORE's multiobjective genetic algorithm to examine power-performance trade-offs. While this current work focuses on distributed memory models, in the future shared memory models will also be considered.

## References

1. Biriukov, A., Ulyanov, D.: Simulation of parallel time-critical programs with the dynamo system. In: Proceedings of the Third IEEE Conference on Control Applications, pp. 825–829. IEEE Computer Society (1994), doi:10.1109/CCA.1994.381214
2. Bliss, N., Kepner, J.: pMatlab Parallel MATLAB Library. International Journal of High Performance Computing Applications (IJHPCA), Special Issue on High-Productivity Programming Languages and Models 21(3), 336–359 (2007)
3. Buluç, A., Gilbert, J.R.: Challenges and advances in parallel sparse matrix-matrix multiplication. In: The 37th International Conference on Parallel Processing (ICPP 2008), pp. 503–510. IEEE Computer Society (2008)
4. Buluç, A., Gilbert, J.R.: New ideas in sparse matrix-matrix multiplication. In: Kepner, J., Gilbert, J.R. (eds.) Graph Algorithms in the Language of Linear Algebra. SIAM Press (2008)
5. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-mat: A recursive model for graph mining. In: SIAM Data Mining SDM 2004 (2004)
6. D'Alberto, P., Nicolau, A.: R-kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. Algorithmica 47(2), 203–213 (2007)
7. Davis, T.A.: Direct Methods for Sparse Linear Systems. SIAM, Philadelphia (2006)
8. Filippone, S., Colajanni, M.: Psblas: a library for parallel linear algebra computation on sparse matrices. ACM Trans. Math. Softw. 26(4), 527–550 (2000)
9. Floyd, R.W.: Algorithm 97: Shortest path. Commun. ACM 5(6), 345 (1962)
10. Foo, S.K., Saratchandran, P., Sundararajan, N.: Genetic algorithm based pattern allocation schemes for training set parallelism in backpropagation neural networks. In: IEEE International Conference on Evolutionary Computation, pp. 545–550. IEEE Computer Society (1995), doi:10.1109/ICEC.1995.487442
11. Gilbert, J.R., Reinhardt, S., Shah, V.B.: A unified framework for numerical and combinatorial computing. Computing in Science and Engg. 10(2), 20–25 (2008)
12. Grefenstette, J.J.: Incorporating problem-specific knowledge into genetic algorithms. In: Davis, L. (ed.) Genetic Algorithms and Simulated Annealing, ch. 4, pp. 42–60. Morgan Kaufmann (1987)
13. Horn, R.A., Johnson, C.R.: Matrix Analysis. Cambridge University Press, Cambridge (1985)
14. Hosny, M.I., Mumford, C.L.: Single vehicle pickup and delivery with time windows: made to measure genetic encoding and operators. In: Proceedings of the 2007 Genetic and Evolutionary Computation Conference, GECCO 2007, pp. 2489–2496. ACM, New York (2007), http://doi.acm.org/10.1145/1274000.1274015, doi:10.1145/1274000.1274015
15. Du, I.S., Erisman, A., Reid, J.K.: Direct Methods for Sparse Matrices. Oxford University Press, Oxford (1986)
16. Jose, A.: An approach to mapping parallel programs on hypercube multiprocessors. In: Proceedings of the Seventh Euromicro Workshop on Parallel and Distributed Processing, PDP 1999, pp. 221–225 (1999), doi:10.1109/EMPDP.1999.746675

17. Kalinowski, T.: Solving the mapping problem with a genetic algorithm on the maspar-1. In: Proceedings of the First International Conference on Massively Parallel Computing Systems, pp. 370–374. IEEE Computer Society (1994), doi:10.1109/MPCS.1994.367057
18. Kepner, J., Bliss, N., Robinson, E.: Linear algebraic graph algorithms for back end processing. In: Proceedings of Workshop on High Performance Embedded Computing, HPEC 2008 (2008)
19. Kuck, D.: High Performance Computing: Challenges for Future Systems. Oxford University Press, New York (1996)
20. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys 31(4), 406–471 (1999), `http://doi.acm.org/10.1145/344588.344618`, doi:10.1145/344588.344618
21. Laird, J.E., Newell, A.: A universal weak method: summary of results. In: IJCAI 1983: Proceedings of the Eighth International Joint Conference on Artificial intelligence, pp. 771–773. Morgan Kaufmann Publishers Inc., San Francisco (1983)
22. Lee, S., Eigenmann, R.: Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems. In: ICS 2008: Proceedings of the 22nd Annual International Conference on Supercomputing, pp. 195–204. ACM, New York (2008)
23. Lin, W.Y.: Parallel sparse matrix ordering: quality improvement using genetic algorithms. In: Proceedings of the 1999 Congress on Evolutionary Computation, CEC 1999, pp. 2295–2301. IEEE Computer Society (1999), doi:10.1109/CEC.1999.785560
24. O'Reilly, U., Bliss, N., Mohindra, S., Mullen, J., Robinson, E.: Multi-objective optimization of sparse array computations. In: Proceedings of Workshop on High Performance Embedded Computing, HPEC 2009 (2009)
25. Rabin, M.O., Vazirani, V.V.: Maximum matchings in general graphs through randomization. J. Algorithms 10(4), 557–567 (1989)
26. Ramaswamy, S., Banerjee, P.: Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In: Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers 1995). IEEE Computer Society (1995)
27. Reuther, A., Kepner, J., McCabe, A., Mullen, J., Bliss, N., Kim, H.: Technical challenges of supporting interactive HPC. In: HPCMP Users Group Conference, pp. 403–409. IEEE Computer Society (2007)
28. Robinson, E.: Array based betweenness centrality. In: SIAM Conference on Parallel Processing for Scientific Computing (2008)
29. Sadd, Y.: Iterative methods for sparse linear systems. SIAM, Philadelphia (2007)
30. Talbi, E.G., Muntean, T.: Hill-climbing, simulated annealing and genetic algorithms: a comparative study and application to the mapping problem. In: Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences, pp. 565–573. IEEE Computer Society (1993), doi:10.1109/HICSS.1993.284069
31. Tarjan, R.E.: A unified approach to path problems. J. ACM 28(3), 577–593 (1981)
32. Travinin, N., Hoffman, H., Bond, R., Chan, H., Kepner, J., Wong, E.: pMapper: Automatic mapping of parallel matlab programs. In: HPCMP Users Group Conference, pp. 254–261. IEEE Computer Society (2005)
33. Travinin Bliss, N., Mohindra, S., O'Reilly, U.: Performance modeling and mapping of sparse computations. In: HPCMP Users Group Conference, pp. 448–456. IEEE Computer Society (2008)

34. Xiong, K., Suh, S., Yang, M., Yang, J., Arabnia, H.: Next generation sequence analysis using genetic algorithms on multi-core technology. In: International Joint Conference on Bioinformatics, Systems Biology and Intelligent Computing (IJCBS 2009), pp. 190–191 (2009), doi:10.1109/IJCBS.2009.104
35. Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., Catalyurek, U.: A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, SC 2005, p. 25. IEEE Computer Society, Washington, DC (2005)
36. Yuster, R., Zwick, U.: Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, pp. 254–260. Society for Industrial and Applied Mathematics, Philadelphia (2004)

# Evolutive Approaches for Variable Selection Using a Non-parametric Noise Estimator

Alberto Guillén, Dušan Sovilj, Mark van Heeswijk, Luis Javier Herrera,
Amaury Lendasse, Héctor Pomares, and Ignacio Rojas

**Abstract.** The design of a model to approximate a function relies significantly on the data used in the training stage. The problem of selecting an adequate set of variables should be treated carefully due to its importance. If the number of variables is high, the number of samples needed to design the model becomes too large and the interpretability of the model is lost. This chapter presents several methodologies to perform variable selection in a local or a global manner using a non-parametric noise estimator to determine the quality of a subset of variables. Several methods that apply parallel paradigms in different architecures are compared from the optimization and efficiency point of view since the problem is computationally expensive.

## 1 Introduction

In many real-life problems like finance, weather forecast, electricity load prediction, medical , etc. it is desirable to decrease the number of existing features (variables) in order to reduce the complexity. This is especially important when the number of input samples is not as large as the number of variables would require to have a good sampling of the space. If the sampling is not good enough, the ultimate task of designing a model that approximates the relationship between the inputs and the output will not succeed.

Alberto Guillén · Luis Javier Herrera · Héctor Pomares · Ignacio Rojas
Department of Computer Technology and Architecture, University of Granada, Spain
e-mail: aguillen@atc.ugr.es

Dušan Sovilj · Mark van Heeswijk · Amaury Lendasse
Department of Information and Computer Science,
Aalto University School of Science, Finland

F.F. de Vega et al. (Eds.): Parallel Architectures & Bioinspired Algorithms, SCI 415, pp. 243–266.
springerlink.com

In order to reduce the dimensionality, several approaches can be considered although not all of them are feasible to use due to the complexity in the evaluation of a solution. To overcome this drawback, information theory and noise estimation have been successfully applied to this problem.

The ultimate problem that this papers aim is to perform a regression, known as well as function approximation. Formally, this problem can be stated as, given a set of inputs/output $\left\{(\mathbf{x}_j;y_j);j=1,...,N\right\}$, a function $F(\mathbf{x}_j)\in\mathbb{R}$ must be designed so when a new vector $\mathbf{x}_{N+1}$ is given as input, the modelled function is able to compute the correct output $F(\mathbf{x}_{N+1})$.

In order to implement $F(\mathbf{x})$, there is a large variety of models that are universal approximators, that is, they can model any kind of relationship between the inputs and the output. The most common models that are used in the literature are Neural Networks [55, 33], Fuzzy Systems [45, 41], kernel methods [46, 60], etc.

Although they can be very accurate, they all suffer from the "Curse of Dimensionality" [40], which implies that, as the number of dimensions $d$ grows, the number of input vectors should be increased exponentially in order to perform a good sampling of the space. When the number of samples is small, the data set could be unbalanced and test results could be quite wrong even if the training results were quite accurate. This problem arises in many real life situations because of the difficulty of performing an exhaustive sampling.

This is not the only issue related to a high dimensionality, the fact of having a large number of variables can lead to models that are not easy to interpret [37] and, for some real life problems, it is important to understand the model and how the variables relate with each others.

Therefore, it is clear that there is a necessity of reducing the dimension of a data set before a model is designed, there are several papers that tackle this problem, but they are applied to the classification problems ([56, 53, 52, 57, 61]) instead of regression. There are several differences between these problems that make unsuitable apply the previous methods:

- Classification targets a finite number of classes meanwhile in regression, an infinite interval of numbers could be generated.
- To perform a miss-classification can be penalised in an equal way, however, in regression, if the output is not exactly the same, if it is "close" enough to the real output, it should not be penalised in the same way.

Due to these differences, specific algorithms for variable selection in regression problems should be designed. The complexity of the design and evaluation of the model in order to evaluate the quality of a subset of variables led researchers to study methods which are not dependent on the model and with as few parameters as possible. In [14] the Delta Test (DT) was proposed as a criterion to perform the variable selection and further studies have confirmed its adequacy. This chapter presents several techniques in order to find the best subset of variables using local and global optimization techniques. A novel aspect that is treated as well is how to speed the expensive computational costs of the process by parallelizing the algorithms using different paradigms.

## 2 Using Delta Test for Variable Selection

The Delta Test (DT) is a technique to estimate the variance of the noise, or the mean squared error (MSE), that can be achieved without overfitting. It was introduced by Pi and Peterson for time series [54] and proposed for variable selection in [14]. Given $N$ input-output pairs $(\mathbf{x}_i, y_i) \in \mathbb{R}^d \times \mathbb{R}$, the relationship between $\mathbf{x}_i$ and $y_i$ can be expressed as

$$y_i = f(\mathbf{x}_i) + r_i, \qquad i = 1, \ldots, N$$

where $f$ is the unknown function and $r_i$ are i.i.d. noise terms. The DT estimates the variance of the noise $r$.

The DT is useful for evaluating the nonlinear correlation between two random variables (i.e. the input and output pairs), and can be also applied to input variable selection: the set of input variables that minimizes the DT is the one that is selected. Indeed, according to the DT, the selected set of input variables is the one for which the relationship with the output variable can be represented in the most deterministic way. DT is based on a hypothesis coming from the continuity of the regression function. If two points $\mathbf{x}$ and $\mathbf{x}'$ are close in the input variable space, the continuity of regression function implies that the outputs $f(\mathbf{x})$ and $f(\mathbf{x}')$ will be close in the output space. Furthermore, if the corresponding output values are not close in the output space, this is due to the influence of noise.

The DT can be interpreted as a special case of the Gamma Test [44] considering only the first nearest neighbour. With the first nearest neighbour of a point $\mathbf{x}_i$ in the $\mathbb{R}^d$ space denoted as $\mathbf{x}_{NN(i)}$, the nearest neighbour formulation of the DT estimating $\mathrm{Var}[r]$ can be written as

$$\mathrm{Var}[r] \approx \delta = \frac{1}{2N} \sum_{i=1}^{N} (y_i - y_{NN(i)})^2,$$

where $y_{NN(i)}$ is the output corresponding to $\mathbf{x}_{NN(i)}$, the estimator is unbiased when $N \to \infty$.

### *2.1 Approaches to Compute the Delta Test*

This subsection presents two different approaches in order to efficiently compute the DT. The fact of having to recompute the distances between the input points makes any algorithm very slow since this computation is quite expensive. Therefore, to be able to optimize this aspect is crucial in order to be able to evaluate an acceptable number of solutions.

### 2.1.1 Computation of Delta Test Using Pre-calculated Distances

Computation of the nearest neighbour in the naive way involves calculating the distances between each pair of samples $\mathbf{d}_{i,j}^2 = \sum_{m=1}^{d}(x_i^{(m)} - x_j^{(m)})^2$ and returning the smallest $\mathbf{d}_{i,j}$ and the corresponding index $NN(i)$ for each sample. Since the focus is on examining non-empty subsets of variables which can share individual elements, a lot of time is wasted recomputing the squared differences to obtain $\mathbf{d}_{i,j}$. A simple solution to decrease running time is to store that information into a $N(N-1)/2 \times d$ matrix, where each row contains precomputed squared differences for a pair of samples $(x_i, x_j)$. Given this matrix, computing all pairwise distances for a given variable subset $I \subseteq \{1,2,\ldots,d\}$ involves summing precomputed values for those $I$ variables (i.e. the $I$-th columns of the matrix).

### 2.1.2 Computation of Delta Test on GPU

The computation of the $k$ Nearest Neighbours (KNN) requires a big computational effort since it has to compute the pairwise distances between all the points. In [17] an implementation of the $k$-NN algorithm on a GPU[1] was presented, showing very large speed-ups compared to CPU times. We use this algorithm to determine the nearest neighbours ($\mathbf{x}_{NN(i)}$) to all input points ($\mathbf{x}_i$).

Differently from the approach in the previous subsection, the pairwise squared differences between all points are not pre-calculated, since it would not be feasible to keep this entire matrix in memory. However, even though we do not make this optimization, computing the pairwise distances between the points can still be many times faster when using a fast GPU instead of the CPU.

Once all pairwise distances have been computed, a partial sort is performed in order to determine the nearest neighbour $\mathbf{x}_{NN(i)}$ (and its index $NN(i)$) to each of points $\mathbf{x}_i$. Finally, given these indices of the nearest neighbours, we can compute the Delta Test as explained in the beginning of this section.

## 3 Local Search Variable Selection Methodologies

This section presents commonly used methodologies to compute the optimization criteria in variable selection setting. These are heuristic approaches and can be easily modified for minimization of DT. Then, an adaptation of the Tabu search is presented which uses these local search algorithms.

[1] The code is available at: http://www.i3s.unice.fr/∼creative/KNN/

The main focus is on variable selection, such that encoding of a solution (subset of variables) is binary. The value indicates whether that variable is relevant or not.

### 3.1 Greedy Local Search Strategies

To obtain optimal solution in variable selection, all subset of variables should be checked and return the one solution that has the best value with respect to the criterion. This requires examining $2^d - 1$ solutions, which is infeasible as $d$ grows. To overcome the difficulties and the high computational time that an exhaustive search would entail, simple greedy algorithms are widely used to obtain a solution that is most promising at least locally. These strategies are affected by local optima because they do not test every input variable combination, but they are preferred over an exhaustive search if the number of variables is too large.

Among the typical search strategies, there are three that share similarities:

- Forward search
- Backward search (or pruning)
- Forward-backward search

The difference between the first two is that the Forward search starts from an empty set of selected variables and adds variables one at a time according to the optimization of a search criterion. The Backward search start starts from a set containing all the variables and removes those for which the elimination optimizes the search criterion.

The Forward-Backward search (FBS) is a combination of of the two previous strategies. Instead of exclusively adding or removing variables, FBS considers both operations and chooses the one most beneficial to the optimization process. If a variable is discarded during earlier stages of the search, it can later be introduced again if improves the search criterion. Another advantage is flexibility in the starting solution as FBS can be initialized with any variable set: empty set, full set, custom set or randomly initialized set.

Denoting with $O$ the optimization criterion, with $S$ the set of selected variables, and $S^C$ the complement of $S$ (all variables that are not in $S$), with $X^k$ the $k-$th variable in data set, the goal is to select set $S^*$ such that $O(S)$ is minimum or maximum, depending on the problem. $O(S)$ is the value of optimization criterion for a data set using only variables in the set $S$.

Given this notation, FBS can be described as follows:

1. Initialization:
   Choose starting set $S$ (empty set, full set, randomly chosen) and compute $O(S)$

2. Examining local solutions:
   Form set $Ne(S) = \{S \cup X^k | k \in S^C\} \cup \{S \setminus X^k | k \in S\}$. Choose a set $S' \in Ne(S)$ with the best $O(S')$.
3. Termination:
   Check if $O(S')$ is better than $O(S)$. If so, then replace $S$ with $S'$ and repeat step 2. Otherwise finish the algorithm and output set $S$ as the final solution.

In the above algorithm, the set of solutions $Ne(S)$ is called the *neighbourhood* of $S$ which plays key role in Tabu search, while the difference in solutions $S$ and $S'$ is a *single* variable. This difference can be seen as changing the status of that variables between binary values, and such change is sometimes referred to as a *move* between solutions.

In the case of variable selection using DT, the goal is to minimize function $O(S)$, which is replaced with $Var[r]$.

## *3.2 Tabu Search*

Tabu Search (TS) is a metaheuristic method designed to guide local search methods to explore the solution space beyond local optimality. The first successful application was by Glover ([18], [19], [20]) for combinatorial optimization. Later TS was successfully used in scheduling ([13], [47], [69]), design ([67], [68]), routing ([5], [63]) and general optimization problems ([21], [38], [1]). The TS has become a powerful method with different components tied together, that is able to obtain excellent results in different problem domains.

In the context of TS,the neighbourhood relationship between solutions, denoted $Ne(S)$, plays the central role. Compared to greedy local strategies, TS uses *memory* in order to influence which parts of the neighbourhood are going to be explored. A memory is used to record various aspects of the search process and the solutions encountered, such as recency, frequency, quality and influence of moves. Instead of storing whole solutions in memory, which is impractical in some problems, the common practice is to store attributes of solutions or moves/operations used to transition from one solution to the next one.

The most important aspect of the memory is to forbid some moves to be applied, or in other words, to prevent the search to go back to solutions that were already visited. This also allows the search to focus on those moves that lead toward unexplored areas of the solutions space. This part of the memory is called a *tabu list*, and the moves in this list are then considered tabu, and thus forbidden to use. The size of the tabu list as well as the time each move is kept in the list are important issues in TS. These parameters should be set up so that the search is able to go through two distinct, but equally important phases: *intensification* and *diversification*.

Another distinction between TS and greedy methods is that TS is not restricted by local optima, i.e. the search is guided towards the best solution in $Ne(S)$ which does not necessarily have to be better then the current solution $S$.

#### 3.2.1 TS for Variable Selection

For variable selection problem, a move is defined as a flip of the status of exactly one variable in the data set. The neighbourhood $Ne(S)$ is defined in the same way as described in the previous subsection, thus set $Ne(S)$ consists of solutions which have exactly one variable changed compared to $S$. With this setup, each solution has exactly the same amount of neighbours, which is equal to $d$.

The *tenure* for a move is defined as the number of iterations that it is considered as tabu. This value is determined empirically when the TS is applied to solve a concrete problem. For the variable selection problem, we propose a value which is dependent on the number of dimensions so it can be applied to several problems. In the experiments, only short-term recency based memory is used which remembers recently performed transitions and discards them after certain number of iterations. This memory keeps track when a variable $X^k$ changes state, and then prevents further change for this variable for $d/4+2$ iterations. This value has been found to be good through experimentation on data sets with different dimensionality.

## 4 Global Searches

Although local search techniques can perform satisfactory for some problems, the risk of falling into a local minimum is high. Genetic Algorithms [42] (GA) have been used for a long time successfully in global optimization problems although recent research suggests that the hybridization with local search procedures achieves better results. This section describes an algorithm of this kind that ameliorates the computational cost by parallelizing different functions.

### *4.1 Hybrid Parallel Genetic Algorithm*

#### 4.1.1 Encoding of the Individuals and Population Initialization

When designing a GA, the first step is to decide how an individual will encode a solution as this decision will condition the rest of the design [10, 51, 49]. For variable selection problems, the most straight forward encoding is to use a binary vector where 0 means that the variable is not selected and 1 that it is selected. There is another approach noted as scaling where the variable is given a weight, so values near 0 mean that the variable is not relevant and values near 1 mean that it is significant. To encode those solutions a vector of real numbers must be used and the GA will fall into the category of Real Coded Genetic Algoritms (RCGA). Nonetheless, the values of the real values should be discretised for, at least, two reasons: 1) the solution space diminishes significantly 2) it is possible to use the precomputed distance matrix.

### 4.1.2 Selection, Crossover and Mutation Operators

Once the individual is defined, it is possible to start choosing which operators will be used to evolve it. The three main aspects are: selection, crossover and mutation. The literature presents a large variety of operators based on heuristics that would require more computation than classical approaches but, as the design restriction demands efficiency, the classical ones were selected.

Regarding the selection, binary tournament selection [22] was shown to be adequate because the compromise it achieves between exploration and exploitation of the solution space instead of Baker's roulette wheel [4] and other complex operators [8]. The main benefit of the operator is that it does not require any extra computation apart from generating 4 random numbers.

The crossovers considered were the classical ones for the binary coded GA: one and two-point crossovers and the uniform crossover [9, 42, 64]. As the experiment section will show, the two-point crossover showed to perform slightly better than the other two.

Another crossover was implemented for the case where the scaling is considered. For this situation, an adaptation of the well known BLX-$\alpha$ [15] was coded in such a way that the offspring are discretised so they remain within the set of discretised values.

Regarding the mutation, it mutates a gene so, when it is applied, a variable is selected or unselected depending on its previous state.

### 4.1.3 Hybridization

The good behaviour of local search when starting from an appropriate initial solution contrasts with the not so satisfactory results when performing the local search from another initial solution. On the other hand, with global search, it is easy to find a good starting point although the time required to refine the search is large and not too robust. Therefore, it seems reasonable to merge both techniques in a hybrid algorithm as was already done in [43, 11, 29, 53]. The use of a local search to provide a good individual to the initial population is advisable, especially when the size of the population is small [58]. Once the GA has a good starting point, it can explore the solution space and provide a final set of solutions, then, a new local search is applied to refine the final result. Regarding the local search applied, TS was selected since it can escape from local minima unlike FBS.

## *4.2 Different Parallelization Approaches*

Even though the computation of the DT is much more efficient than the design of a model in order to determine the fitness of a subset of variables, it still requires a considerable amount of effort since all the distances between the input samples must be computed in order to determine the nearest neighbours. Therefore, in order to take advantage of the available architectures, the algorithm has been optimized and parallelized to provide better solutions in less time.

The parallelization has been done in several ways that can be further extended with more functionalities. The first way of optimization and parallelization consists in the computation of a big distance matrix where all the distances between each pair of points are stored. This computation was distributed among processors. Furthermore, the evolution process was also parallelized so the individuals of a single population are evaluated in different processors.

The second approach optimizes the memory consumption of the previous approach by optimizing the computation of the distances using Graphic Processing Units (GPUs). Moreover, the second stage of evolving the individuals was implemented in parallel so several populations are evolved simultaneously.

In the following subsections, these approaches are described in detail.

### 4.2.1 Using Precomputed Distance Matrix and One Population

Among the different approaches to parallelize a GA [7, 3, 2], for this first approach, the classical master/slave paradigm[26] would be the more suitable to classify the implementation.

Computing the distances and storing them in memory shows to be much more efficient that recomputing the distances using the CPU. Nonetheless, the evaluation of the population remains as the most time consuming part of the GA stages. As was described in the literature [7], the master/slave approach is the one where the sequential part of the algorithm is executed in one process (processor) and the evaluation of the individuals is distributed among all the processors available. Although this approach seems quite straight-forward, several aspects must be considered like the homogeneity of the processors, the variability in the computation of the fitness function, the topology of the communications, etc.

In this implementation, we can assume that the time to compute DT independently of the number of variables selected by the individual is very similar since the process only has to make several memory access which are allocated in contiguous positions in memory. Therefore, the number of individuals sent to the processors can be the same (assuming that the cluster has homogeneous processors) and equal to $size_o f_p opulation/number_o f_p rocessors$.

**Data Parallelism Paradigm**

According to the classification of parallel paradigms in data and functional, this implementation fits into the data parallelism since it considers the same data (a population) and chops it into several slices where it is processed independently. The main problem with this approach is the time spent distributing and collecting the chunk that correspond to each process.

In order to minimize the number of communications and the size of the packets sent, the implementation was optimized in such a way that it only requires one communication operation on each iteration. Since the number of processes is known in advance and we are assuming that the time of evaluation is constant, each process (mapped to a processor) knows in advance how many individuals will evaluate. The problem is that GAs have a random component that makes populations evolve in a non-determinist way so, a priori, the master should distribute the new individuals generated to the others. To skip this expensive step, all processes will execute the same code so all of them have the entire population at their disposal and, without any communication, will take care of their individuals. The trick consists in broadcasting the seed of the random number generator at the beginning of the execution so, when a process calls the function, it will obtain the same value as the other processes. Therefore, all the processes will perform selection, crossover and mutation with the same values. Thus the only communication that is needed is to send and receive the fitness of the individuals evaluated by the other processes.

**Hybridizing with Local Approaches**

A good way to obtain both benefits that the local and the global search provides is to merge them. This objective can be achieved by applying the local search at some stage of the GA: initialization, evolution and final solution. As it is known, the inclusion of good individuals in the initial population will lead to better populations and final solutions so it is quite reasonable to introduce at least an adequate individual generated using local search. The local search procedure chosen was TS because it escapes from local minima. Each process will start from a random point the local search procedure so the initial population will have $p$ individuals generated by TS, where $p$ is the number of processes.

The use of local search during the evolution has been applied in [29], however, since we already started from an initial population generated in that way, applying it again during the evolution might lead to a premature convergence.

Finally, once the GA is finished, TS is applied again to the best solutions. As we have several processes, several TS can be aplied in parallel so process 1 applies TS to the best individual, process 2 to the second best and so on.

The structure of the algorithm is depicted in Figure 1 showing the communications steps needed to obtain the final solution.

### 4.2.2 Multi-deme GA Using a Cluster of GPUs

The previous approach evolves a unique population using several computers to evaluate the individuals of that population, however, other approaches exist where several populations can be evolved at the same time. This kind of GAs usually include a migration step or operator that allow the isolated populations to communicate with others performing an exchange of individuals. This class of algorithms have shown a good behaviour outperforming the sequential approaches [3, 32, 30, 31].

**Island Model**

The paradigm of evolving isolated populations in parallel and communicating them periodically is known as the island model [6, 31, 39]. The classical mapping is to assign one island per processor so all populations can evolve at a similar speed.

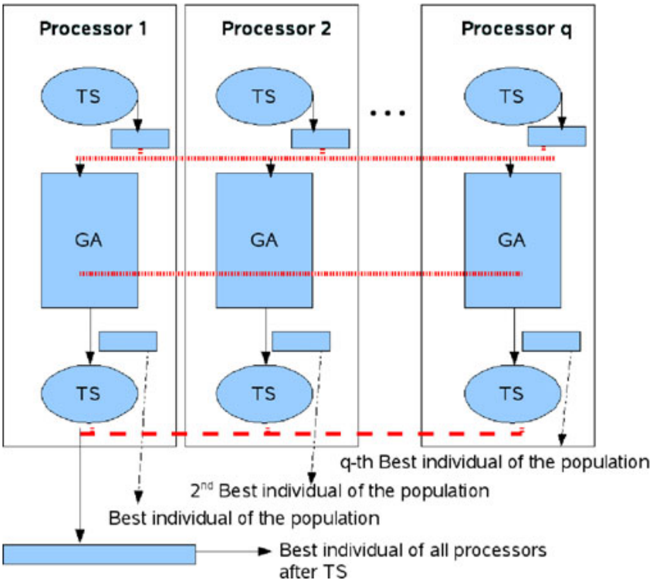

Processor 1
Processor 2
Processor q
TS
TS
TS
GA
GA
GA
TS
TS
TS
q-th Best individual of the population
2nd Best individual of the population
Best individual of the population
Best individual of all processors after TS

**Fig. 1** Algorithm scheme. Dashed line represents one to one communication, dotted lines represent collective communications.

The previous approach is very efficient for computing the DT. However, the need of precomputing the distance matrix forces to have a large memory. For big data sets, this might suppose a critical problem. In order to overcome this drawback, the computation of the DT was implemented so it can be computed using a Graphic Processing Unit (GPU). As the architectures improve, nowadays, it is possible to find a machine with several cores and several GPUs, therefore, the code was adapted so it can be executed in a machine like this. Furthermore, it is possible to connect several of these machines in a classical cluster manner so scalability is guaranteed. The mapping of the islands in the processors is recommended to be done using one pair of CPU/GPU to process one population (island). Otherwise, more synchronization and data distribution mechanisms, which could make the process less efficient, should be implemented.

The way in which the migration is done can be random [50], fixed [31], or using information about the population like diversity or convergence. In this work, a fixed migration scheme after a certain number of generations, like the one presented in [32] was implemented. It can be considered as an elitism operator because the best individuals of each population are sent to the others, replacing the worst individuals, in the same way the classical elitism keeps the best individual (or a few of them) in the following generations. The way in which the migration is performed is shown in 2.

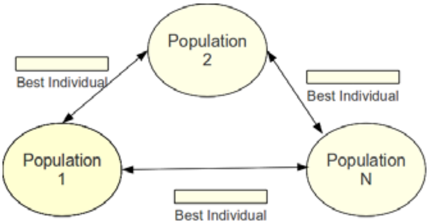


**Fig. 2** Island migration scheme.

**Population distribution in the cluster**

The architecture presented allows to have heterogeneous nodes with different CPU and GPU models as used in [36]. As the migration scheme is fixed, there is the possiblity of slowing the fastest machines that will have to wait for the slower ones to finish their generations. In order to fix this problem (in the case it arises), the decision of using different population sizes was taken in such a way that slower machines will have to process smaller number of individuals than the faster ones. The determination of the population size can be taken dinamically depending on the performance of the fastest machine so the other sizes are computed depending on this one: $popSize_w = popSize_0 * S_w$ where $w = 1...p$ and $S$ is division of the time required to evaluate one individual in the machine $w$ and the time required to evaluate the same individual in the fastest machine (processor 0).

## 5 Experiments and Results

This Section presents the results obtained by the different design stages of the algorithms as well as a comparison of several alternatives. The first experiment analyses the effect of computing the distance matrix and evaluating the individuals using several computers. The next experiment compares different design alternatives by hybridizing the local and global searches. Afterwards, the use of GPUs to speed up the evaluation of the individuals and the evolution of several islands is compared with the hybrid approach.

In order to set a stop criterion for global searches, a time limit of 600 seconds was chosen. The reason is because, as experienced with workers in the industry, 10 minutes is the maximum time that an operator wants to wait. This time limit has been used already in previous work in the literature [66, 70].

### *5.1 Computer Architectures Used*

The algorithms were implemented in MATLAB and, in order to communicate the different processes, the MPImex ToolBox with the capabilities to use GPUs presented in [34] was used.

### 5.1.1 Homogeneous Cluster of Computers

The experiments carried out for the evaluation of the data parallelism used a cluster of homogeneous nodes where each processor has the following characteristics:

**Table 1** Node specifications used in the the homogeneous cluster

| | |
|---|---|
| *Cpu family* | 6 |
| *Model* | 15 |
| *Model name* | Intel(R) Xeon(R) CPU E5320 @ 1.86GHz |
| *Stepping* | 7 |
| *Cpu MHz* | 1595.931 |
| *Cache size* | 4096 KB |
| *Cpu cores* | 2 |
| *Bogomips* | 3723.87 |
| *Clflush size* | 64 |
| *Cache alignment* | 64 |
| *Address sizes* | 40 bits physical, 48 bits virtual |

### 5.1.2 Heterogeneous Cluster of CPUs with GPUs

The cluster that was configured had the components described below that were interconnected as Figure 3 shows.

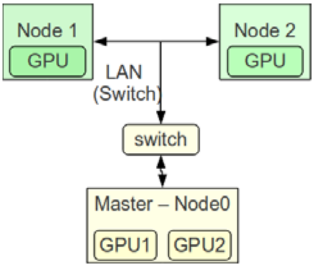


**Fig. 3** Cluster of GPUs used in the experiments

## 5.2 *Data Sets Used in the Experiments*

To compare the different approaches, several well-known data sets were used. These data sets are, as described in [35]:

1. The Housing data set[2]: The housing data set is related to the estimation of housing values in suburbs of Boston. The value to predict is the median value of

[2] http://archive.ics.uci.edu/ml/data sets/Housing.

**Table 2** Node specifications of the heterogeneous cluster of GPUs

| **Master node with 2 GPUs** | |
|---|---|
| **Processor** | |
| *model name* | (26) Intel(R) Core(TM) i7 CPU 930 @ 2.80GHz |
| *cache size* | 8192 KB |
| *cpu cores* | 4 |
| **2 GPUs** | |
| *Graphics Processor* | GeForce GTS 450 |
| *CUDA Cores* | 192 |
| *Memory* | 1024 MB |
| *Memory Interface* | 128-bit |
| *Bus Type* | PCIExpress x16 Gen1 |
| *PCI-E Max Link Speed* | 2500 |

| **Two local network nodes with 1 GPU** | |
|---|---|
| **Node 1 Processor** | |
| *model name* | (23) Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz — *cache size* 6144 KB |
| *cpu cores* | 4 |
| **Node 1 GPU** | |
| *Graphics Processor* | GeForce 9800 GTX |
| *CUDA Cores* | 128 |
| *Memory* | 512 MB |
| *Memory Interface* | 256-bit |
| *Bus Type* | PCIExpress x16 Gen2 |
| *PCI-E Max Link Speed* | 5000 |
| **Node 2 Processor** | |
| *model name* | (15) Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz |
| *cache size* | 4096 KB |
| *cpu cores* | 4 |
| **Node 2 GPU** | |
| *Graphics Processor* | GeForce 8400 GS |
| *CUDA Cores* | 16 |
| *Memory* | 512 MB |
| *Memory Interface* | 64-bit |
| *Bus Type* | PCIExpress x16 |
| *PCI-E Max Link Speed* | not available |

owner-occupied homes in $1000's. The data set contains 506 instances, with 13 input variables and one output.

2. The Tecator data set[3]: The Tecator data set aims at performing the task of predicting the fat content of a meat sample on the basis of its near infrared absorbance spectrum. The data set contains 215 useful instances for interpolation problems, with 100 input channels, 22 principal components (which will remain unused) and 3 outputs, although only one is going to be used (fat content).
3. The Anthrokids data set[4]: This data set represents the results of a three-year study on 3900 infants and children representative of the U.S. population of year 1977,

[3] http://lib.stat.cmu.edu/data sets/tecator.

[4] http://ovrt.nist.gov/projects/anthrokids.

ranging in age from newborn to 12 years of age. The data set comprises 121 variables and the target variable to predict is children's weight. As this data set presented many missing values, a prior sample and variable discrimination had to be performed to build a robust and reliable data set. The final set[5] without missing values contains 1019 instances, 53 input variables and one output (weight). More information on this data set reduction methodology can be found in [48].

4. The Finance data set[5]: This data set contains information of 200 French industries during a period of 5 years. The number of samples is 650. It contains 35 input variables, related to balance sheet, income statement and market data, and one output variable, called "return on assets" (ROA). This is an indicator of how profitable a company is relative to its total assets. It is usually calculated by dividing a company's annual earnings by its total assets.
5. The Santa Fe time series competition data set[6]: The Santa Fe data set is a time series recorded from laboratory measurements of a Far-Infrared-Laser in a chaotic state, and proposed for a time series competition in 1994. The set contains 1000 samples, and it was reshaped for its application to time series prediction using regressors of 12 samples. Thus, the set used in this work contains 987 instances, 12 inputs and one output.
6. The ESTSP 2007 competition data set[5]: This time series was proposed for the European Symposium on Time Series Prediction 2007. It is an univariate set containing 875 samples but has been reshaped using a regressor of 55 variables, producing a final set of 819 samples, 55 variables and one output.

All the data sets were normalized to zero mean and unit variance, so the DT values obtained are normalized by the variance of the output.

## 5.3 Parallelization of the Sequential GA

The first tests that should be done are the ones that confirm that a parallel implementation outperforms sequential ones keeping in mind that the improvement obtained depends in the measure used. As a time limit of 600 seconds was used to stop the algorithm, parallel implementations will not be faster than sequential ones so, instead of measuring the execution time, the variables to compare sequential and parallel approaches will be the number of generations computed by the algorithms and the quality of the final solution given by the algorithms.

The analysis of the performance between the sequential GA and the parallelization does not consider the hybridization with TS since this would distort one of the measures selected.

The parameters of the GA were set to the following values:

- Crossover Type: BLX-$\alpha$
- Crossover Rate: 0.85

[5] http://research.ics.tkk.fi/eiml/datasets.shtml

[6] http://www-psych.stanford.edu/~andreas/Time-Series/SantaFe.html

- Mutation Rate: 0.1 [7]
- Generational elitism: 10%

The sizing of the population is another critical issue when optimizing a genetic algorithm, and therefore there are many studies in that field [23, 24, 25, 59, 65]. Most recommended settings are established by empirical rules, and usually range from 30 to 100 individuals depending on the author [12, 27, 62]. The results of DT were obtained for population sizes of 50, 100 and 150 individuals, and averaged from 5 or 10 repetitions, depending on the case.

In this experiment, the crossover operator used is the BLX-$\alpha$ in order to weight the importance of the variables. The aim of using this operator instead of performing the classical binary approach is to obtain smaller DT values, therefore, the GA turns out into a parallel Real Coded Genetic Algorithm (pRCGA).

Table 3 shows the results for the sequential and the parallel approach for three data sets: Anthrokids, Tecator and ESTSP. The values correspond to the number of generations evolved (higher is better) and the final DT (lower is better).

The way in which the number of processors influences the number of generations is shown graphically in Figure 4 showing the same behaviour for the other data sets.

The results show how the bigger the population is, the less the number of generations is confirming that the evaluation function is the bottleneck for the GA.

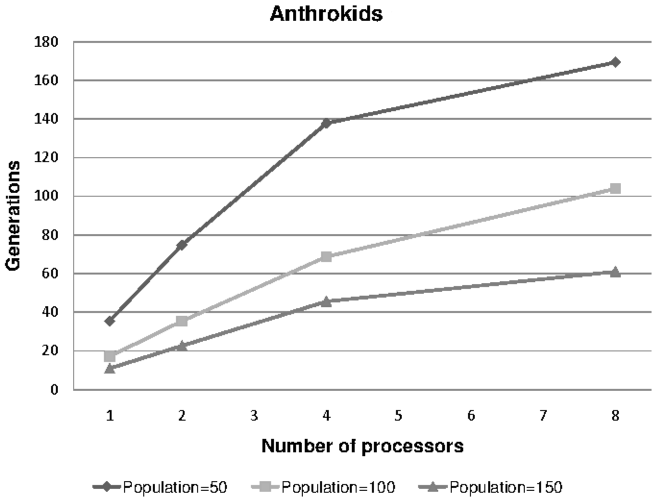


**Fig. 4** Generations evaluated by the GA vs the number of processors used. Anthrokids without scaling.

[7] This is the same rate used in [53], also for a feature selection application. This rate is higher than the recommended by most authors [12, 16, 27, 62], but it is motivated by the nature of the search: a wide space of solutions needs to be explored, sacrificing exploitation power.

As the parallel approach expands easily, the number of generations increase almost linearly with the number of processors, specially with large populations. Due to the increase in the number of generations, better solutions regarding the DT can be obtained as Table 3 showed.

**Table 3** Performance of RCGA vs pRCGA for Three Different data sets. Values of the Delta Test and Number of Generations Completed.

| Data set | Population | Measurement | RCGA | pRCGA (np=2) | pRCGA (np=4) | pRCGA (np=8) |
|---|---|---|---|---|---|---|
| Anthrokids | 50 | Mean (DT) | 0.01278 | 0.01269 | 0.01204 | 0.01347 |
| | | StDev (DT) | 11.5e-4 | 14.2e-4 | 12.6e-4 | 8.1e-4 |
| | | Mean (Gen.) | 35.5 | 74.8 | 137.8 | 169.3 |
| | | StDev (Gen.) | 1.9 | 4.1 | 6.8 | 13.8 |
| | 100 | Mean (DT) | 0.01351 | 0.01266 | 0.01202 | 0.0111 5 |
| | | StDev (DT) | 11.6e-4 | 86.4e-4 | 17.4e-4 | 5.6e-4 |
| | | Mean (Gen.) | 17.2 | 35.4 | 68.8 | 104 |
| | | StDev (Gen.) | 1 | 1.2 | 4.3 | 28.2 |
| | 150 | Mean (DT) | 0.01475 | 0.01318 | 0.01148 | 0.01105 |
| | | StDev (DT) | 12.1e-4 | 11.2e-4 | 9.9e-4 | 12e-4 |
| | | Mean (Gen.) | 11 | 22.7 | 45.6 | 61 |
| | | StDev (Gen.) | 0.8 | 0.9 | 0.6 | 4.2 |
| Tecator | 50 | Mean (DT) | 0.13158 | 0.14297 | 0.13976 | 0.1365 |
| | | StDev (DT) | 7.9e-4 | 7.7e-3 | 7.8e-3 | 3.7e-3 |
| | | Mean (Gen.) | 627 | 1129.4 | 2099.2 | 3369.5 |
| | | StDev (Gen.) | 39.5 | 55.4 | 119.6 | 256.7 |
| | 100 | Mean (DT) | 0.13321 | 0.13587 | 0.13914 | 0.13525 |
| | | StDev (DT) | 3.1e-3 | 2.4e-3 | 8.6e-3 | 3e-4 |
| | | Mean (Gen.) | 310.8 | 579.6 | 1110.4 | 1731 |
| | | StDev (Gen.) | 23.6 | 34.4 | 61.5 | 32.5 |
| | 150 | Mean (DT) | 0.13146 | 0.1345 | 0.13522 | 0.1303 |
| | | StDev (DT) | 8.5e-4 | 2.4e-3 | 6.9e-3 | 9.9e-4 |
| | | Mean (Gen.) | 195 | 388.1 | 741.2 | 1288 |
| | | StDev (Gen.) | 14.6 | 26.1 | 19.9 | 21.2 |
| ESTSP | 50 | Mean (DT) | 0.01422 | 0.01452 | 0.01444 | 0.01403 |
| | | StDev (DT) | 1.8e-4 | 3.7e-4 | 2.5e-4 | 2.9e-4 |
| | | Mean (Gen.) | 51 | 99.2 | 190.8 | 229 |
| | | StDev (Gen.) | 2.7 | 8.5 | 16.4 | 7.9 |
| | 100 | Mean (DT) | 0.01457 | 0.01419 | 0.01406 | 0.01393 |
| | | StDev (DT) | 2.5e-4 | 3.9e-4 | 2.9e-4 | 3.2e-4 |
| | | Mean (Gen.) | 24.8 | 50.5 | 93 | 128.7 |
| | | StDev (Gen.) | 1.4 | 2.8 | 2.9 | 2.1 |
| | 150 | Mean (DT) | 0.01464 | 0.01429 | 0.01402 | 0.0141 |
| | | StDev (DT) | 3.4e-4 | 2.1e-4 | 1.8e-4 | 1.4e-4 |
| | | Mean (Gen.) | 16.6 | 33.6 | 63.2 | 82.5 |
| | | StDev (Gen.) | 0.8 | 1.2 | 2 | 2.1 |

### 5.4 Combining TS and the pRCGA

The experiments also consider the incorporation of the TS with the pRCGA to obtain smaller DT values. In order to perform this hybridization, it is necessary to consider elements like: resource allocation for TS, when to apply it, and to which individuals.

Regarding the moment to apply the TS, several approaches can be taken:

- at the begging to initialise the population
- during the evolution, as a mutation operator
- at the end of the evolution

These three possibilities can be combined simultaneously [29] although, in order to avoid a premature convergence during the evolution and satisfy the time constraint, the TS was not applied during the evolution and only was considered at the beginning and at the end. As there is a time limit to execute the algorithm, it must be decided how much time will be spent on each of the three stages: TS start, GA and TS end.

Several experiments were carried out using different configurations of the time dedicated to the stages. However, the experiments always considered less time dedicated to the initial TS since highly refined individuals could lead to a fast convergence, making the global search not useful anymore.

Table 4 shows the results for the following alternative: pRCGA, pTSGA using TS at the end and pTSGA at the beginning and at the end. Following the results obtained in the previous experiments, the population was fixed to 150 individuals.

As the results show, the application of TS improves the performance of the algorithm in two ways: the DT values obtained are smaller and the robustness of these good results is increased.

**Table 4** Performance of pRCGA vs pTSGA, with the BLX-$\alpha$ Crossover Operator. The time distribution is $t_{TS_1}/t_{GA}/t_{TS_2}$ where $t_{GA}$ is the time (in seconds) dedicated to the GA, $t_{TS_1}$ is the time dedicated to the first TS, and $t_{TS_2}$ the time dedicated to the last.

| Data set | Measurement | pRCGA | pTSGA 0/400/200 | pTSGA 50/325/225 |
|---|---|---|---|---|
| Anthrokids | Mean (DT) | 0.0113 | 0.0084 | 0.0083 |
| | StDev (DT) | 11.5e-4 | 17.3e-5 | 5.8e-5 |
| Tecator | Mean (DT) | 0.13052 | 0.1180 | 0.1113 |
| | StDev (DT) | 25.8e-4 | 12.1e-3 | 88.9e-4 |
| ESTSP | Mean (DT) | 0.01468 | 0.01302 | 0.01303 |
| | StDev (DT) | 16.4e-5 | 8.4e-5 | 5.8e-5 |
| Housing | Mean (DT) | 0.0710 | 0.0710 | 0.0710 |
| | StDev (DT) | 0 | 0 | 0 |
| Santa Fe | Mean(DT) | 0.0165 | 0.0165 | 0.0165 |
| | StDev (DT) | 0 | 0 | 0 |
| Finance | Mean(DT) | 0.1498 | 0.1406 | 0.1406 |
| | StDev (DT) | 3.4e-4 | 7.9e-4 | 1.2e-4 |

## 5.5 *Experiments Using GPUs*

This section will compare the previous parallel approach with the parallelization that can be obtained using clusters of GPUs. Since it is desired to analyse the effect of parallelism in the evolution, the TS was not included in the algorithm.

### 5.5.1 Small Datasets

The same parameters for the GA were used as well as the same population sizes (50,100 and 150).Two data sets are used (Tecator and Anthrokids) since they have the largest number of features. Even though they are the largest of the previous experiments, the number of input vectors and variables to be selected is quite small in comparison with real life problems that can be highly monitorised.

As Table 5 shows, the results provided by the GPU implementation do not improve significantly the ones provided by the classical cluster approach. For these type of data sets, the precomputation of the distance matrix makes the computation of the DT much faster even for the use of GPUs.

**Table 5** Performance in tems of DT value of the cluster of GPUs (pGPU) against sequential and parallel approaches.

| Data set | Population | Measurement | seq. | parallel(np=2) | parallel(np=4) | pGPU(np=4,4GPUs) |
|---|---|---|---|---|---|---|
| Anthrokids | 50 | Mean (DT) | 0.01278 (11.5e-4) | 0.01269 (14.2e-4 ) | 0.01204 (12.6e-4) | 0.01587 (8.1e-3) |
| | 100 | Mean (DT) | 0.01351 (11.6e-4) | 0.01266 (86.4e-4) | 0.01202 (17.4e-4) | 0.014553 (5.6e-4) |
| | 150 | Mean (DT) | 0.01475 (12.1e-4) | 0.01318 (11.2e-4) | 0.01148 (9.9e-4) | 0.01556(12e-4) |
| Tecator | 50 | Mean (DT) | 0.13158(7.9e-4) | 0.14297 (7.7e-3) | 0.13976 (7.8e-3) | 0.123803 (3.7e-3) |
| | 100 | Mean (DT) | 0.13321 (3.1e-3) | 0.13587 (2.4e-3) | 0.13914 (8.6e-3) | 0.132501 (3e-4) |
| | 150 | Mean (DT) | 0.13146 (8.5e-4) | 0.1345 (2.4e-3) | 0.13522 (6.9e-3) | 0.13197 (9.9e-4) |

### 5.5.2 Large Datasets

As commented before, the data sets used so far can be considered small, if the number of input vectors or variables increases significantly, the optimization of computing the distance matrix in advance becomes infeasible due to the memory constraints. For these situations, the use of a cluster of GPUs becomes the only way to obtain some results.

The data to be used in this experiment consists of 19967 input vectors with 20 variables that was divided so 15385 input vectors were used for training and the rest for test. The input vectors contain information about the marital dissolutions in Spain and the data was provided by the Spanish Institute of Statistics (Instituto Nacional de Estadística, INE). The output desired to predict consists in how many months the dissolution process will take, so children damage can be reduced.

The algorithm was executed for several population sizes obtaining the results showed in Table 6.

**Table 6** Number of variables selected and DT values for the large size data set.

| Population size | DT value (std) | # vars |
|---|---|---|
| 50 | 0.00167 (1e-4) | 7.3 (0.57) |
| 100 | 0.00166 (2e-5) | 8.6 (1.52) |
| 150 | 0.00167 (3e-5) | 9 (1) |

To check if the variable selection performed by the algorithm is effective, as there are no other results to compare, several regression models were designed to see if they can achieve better approximation errors after performing the variable selection. Table 7 shows the Normalised Mean Squared Error (NRMSE) of the approximations using Radial Basis Function Neural Network designed with Improved Clustering for Function Approximation (ICFA) [33]. The approximation errors obtained after the variable selection are much better than without it both in test and training for both algorithms and different number of neurons.

**Table 7** Approximation errors (NRMSE) of the large data set with and without variable selection using a RBFNNs with 5 neurons and 7 variables.

| | Train Error | Test Error |
|---|---|---|
| without var. selec. | 0.5444 | 0.5689 |
| with var. selec. | 0.4869 | 0.5051 |

## 6 Conclusions

This chapter has covered several strategies to perform variable selection (including scaling) based on the non-parametric noise estimator, the Delta Test. Among the optimization strategies there are local and global searches that, combining them together, are able to provide better results.

From the implementation point of view, several parallel approaches have been compared leading to the conclusion that for small data sets, the precomputation of the input vector distances is efficient however, when the data size requires too much memory, the only way to obtain a good solution is by speeding up the algorithm using GPUs.

## References

1. Al-Sultan, K.S., Al-Fawzan, M.A.: A tabu search hooke and jeeves algorithm for unconstrained optimization. European Journal of Operational Research 103(1), 198–208 (1997)
2. Alba, E., Luna, F., Nebro, A.J.: Advances in parallel heterogeneous genetic algorithms for continuous optimization. Int. J. Appl. Math. Comput. Sci. 14, 317–333 (2004)

3. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Trans. on Evolutionary Computation 6(5), 443–462 (2002)
4. Baker, J.E.: Reducing bias and inefficiency in the selection algorithm. In: Grefenstette, J.J. (ed.) Proceedings of the Second International Conference on Genetic Algorithms, pp. 14–21. Lawrence Erlbaum Associates, Hillsdale (1987)
5. Brandao, J.: A tabu search algorithm for the open vehicle routing problem. European Journal of Operational Research 157(3), 552–564 (2004)
6. Cantú-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Massachusetts (2000)
7. Cantú-Paz, E.: Markov chain of parallel genetic algorithms. IEEE Trans. Evolutionary Computation 4, 216–226 (2000)
8. Chakraborty, U.K., Deb, K., Chakraborty, M.: Analysis of selection algorithms: A markov chain approach. Evol. Comput. 4(2), 133–167 (1996)
9. De Jong, K.A.: An analysis of the behavior of a class of genetic adaptive systems, Ph.D. thesis, University of Michigan (1975)
10. De Jong, K.A.: Evolutionary computation: Recent developments and open issues. In: Goodman, E.D., Punch, B., Uskov, V. (eds.) Proceedings of the First International Conference on Evolutionary Computation and Its Applications, Moscow, pp. 7–17 (1996)
11. Deb, K., Goel, T.: Controlled Elitist Non-dominated Sorting Genetic Algorithms for Better Convergence. In: Zitzler, E., Deb, K., Thiele, L., Coello Coello, C.A., Corne, D.W. (eds.) EMO 2001. LNCS, vol. 1993, pp. 67–81. Springer, Heidelberg (2001)
12. DeJong, K.A., Spears, W.M.: An Analysis of the Interacting Roles of Population Size and Crossover in Genetic Algorithms. In: Schwefel, H.-P., Männer, R. (eds.) PPSN 1990. LNCS, vol. 496, pp. 38–47. Springer, Heidelberg (1991)
13. DellÁmico, M., Trubian, M.: Applying tabu search to the job-shop scheduling problem. Ann. Oper. Res. 41(1-4), 231–252 (1993)
14. Eirola, E., Liitiäinen, E., Lendasse, A., Corona, F., Verleysen, M.: Using the delta test for variable selection. In: European Symposium on Artificial Neural Networks, ESANN 2008, Bruges, Belgium, pp. 25–30 (April 2008)
15. Eshelman, L.J., Schaffer, J.D.: Real-coded genetic algorithms and interval schemata. In: Darrell Whitley, L. (ed.) Foundation of Genetic Algorithms, vol. 2, pp. 187–202. Morgan-Kauffman Publishers, Inc. (1993)
16. Fogarty, T.C.: Varying the probability of mutation in the genetic algorithms. In: Schaffer, J.D. (ed.) Proc. of the Third International Conference on Genetic Algorithms, pp. 104–109. Morgan-Kauffman Publishers, Inc. (June 1989)
17. Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using GPU. In: CVPR Workshop on Computer Vision on GPU (2008)
18. Glover, F.: Future paths for integer programming and links to artificial intelligence. Comput. Oper. Res. 13(5), 533–549 (1986)
19. Glover, F.: Tabu search part i. ORSA Journal on Computing 1(3), 190–206 (1989)
20. Glover, F.: Tabu search part ii. ORSA Journal on Computing 2, 4–32 (1990)
21. Glover, F.: Parametric tabu-search for mixed integer programs. Comput. Oper. Res. 33(9), 2449–2494 (2006)
22. Goldberg, D.E.: Genetic algorithms in search, optimization and machine learning. Addison Wesley (1989)
23. Goldberg, D.E.: Optimal initial population size for binary-coded genetic algorithms, Technical Report TCGA 85001, Department of Engineering Mechanics, University of Alabama, Tuscaloosa, AL 35486 (November 1985)
24. Goldberg, D.E.: Sizing populations for serial and parallel genetic algorithms. In: Schaffer, J.D. (ed.) Proc. of the Third International Conference on Genetic Algorithms, pp. 398–405. Morgan-Kauffman Publishers, Inc. (June 1989)

25. Goldberg, D.E., et al.: Genetic algorithms, noise and the sizing of populations. Complex Systems 6, 333–362 (1992)
26. Grefenstette, J.J.: Parallel adaptive algorithms for function optimization, Technical Report TCGA CS-81-19, Department of Engineering Mechanics, University of Alabama, Vanderbilt University (1981)
27. Grefenstette, J.J.: Optimization of control parameters for genetic algorithms. IEEE Trans. Systems, Man and Cybernetics 16(1), 122–128 (1992)
28. Guillén, A., González, J., Rojas, I., Pomares, H., Herrera, L.J., Valenzuela, O., Rojas, F.: Output Value-Based Initialization For Radial Basis Function Neural Networks. Neural Processing Letters (2007)
29. Guillén, A., Pomares, H., González, J., Rojas, I., Herrera, L.J., Prieto, A.: Parallel multi-objective memetic rbfnns design and feature selection for function approximation problems. Neurocomputing, 3541–3555 (2009)
30. Guillén, A., Pomares, H., González, J., Rojas, I., Valenzuela, O., Prieto, B.: Parallel multiobjective memetic rbfnns design and feature selection for function approximation problems. Neurocomputing 72(16-18), 3541–3555 (2009)
31. Guillén, A., Rojas, I., González, J., Pomares, H., Herrera, L.J., Paechter, B.: Improving the Performance of Multi-objective Genetic Algorithm for Function Approximation Through Parallel Islands Specialisation. In: Sattar, A., Kang, B.-h. (eds.) AI 2006. LNCS (LNAI), vol. 4304, pp. 1127–1132. Springer, Heidelberg (2006)
32. Guillén, A., Rojas, I., González, J., Pomares, H., Herrera, L.J., Paechter, B.: Boosting the Performance of a Multiobjective Algorithm to Design RBFNNs Through Parallelization. In: Beliczynski, B., Dzielinski, A., Iwanowski, M., Ribeiro, B. (eds.) ICANNGA 2007. LNCS (LNAI), vol. 4431, pp. 85–92. Springer, Heidelberg (2007)
33. Guillén, A., Rojas, I., González, J., Pomares, H., Herrera, L.J., Valenzuela, O., Prieto, A.: Improving Clustering Technique for Functional Approximation Problem Using Fuzzy Logic: ICFA algorithm. Neurocomputing 70(16-18), 2853–2860 (2007)
34. Guillén, A., Garcia-Arenas, M., Herrera, L.J., Pomares, H., Rojas, I.: GPU Cluster with MATLAB. In: International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 37–46 (2011)
35. Guillén, A., Sovilj, D., Lendasse, A., Mateo, F., Rojas, I.: Minimising the delta test for variable selection in regression problems. Int. J. High Perform. Syst. Archit. 1, 269–281 (2008)
36. Guillén, A., van Heeswijk, M., Sovilj, D., Arenas, M.G., Herrera, L.J., Pomares, H., Rojas, I.: Variable Selection in a GPU Cluster Using Delta Test. In: Cabestany, J., Rojas, I., Joya, G. (eds.) IWANN 2011, Part I. LNCS, vol. 6691, pp. 393–400. Springer, Heidelberg (2011)
37. Guyon, I., Gunn, S., Nikravesh, M., Zadeh, A.: Feature extraction: Foundations and applications. STUDFUZZ (studies in fuzziness and soft computing). Springer-Verlag New York, Secaucus (2006)
38. Hedar, A.-R., Fukushima, M.: Tabu search directed by direct search methods for nonlinear global optimization. European Journal of Operational Research 170(2), 329–349 (2006)
39. Herrera, F., Lozano, M.: Gradual distributed real-coded genetic algorithms. IEEE Transactions on Evolutionary Computation 4(1), 43 (2000)
40. Herrera, L.J., Pomares, H., Rojas, I., Verleysen, M., Guilén, A.: Effective Input Variable Selection for Function Approximation. In: Kollias, S.D., Stafylopatis, A., Duch, W., Oja, E. (eds.) ICANN 2006. LNCS, vol. 4131, pp. 41–50. Springer, Heidelberg (2006)

41. Herrera, L.J., Pomares, H., Rojas, I., Guillén, A., Valenzuela, O.: The TaSe-NF model for function approximation problems: Approaching local and global modelling. Fuzzy Sets and Systems 171(1), 1–21 (2011)
42. Holland, J.J.: Adaption in natural and artificial systems. University of Michigan Press (1975)
43. Ishibuchi, H., Yoshida, T., Murata, T.: Balance between genetic search and local search in memetic algorithms for multiobjective permutation flowshop scheduling. IEEE Trans. on Evolutionary Computation 7, 204–223 (2003)
44. Jones, A.: New tools in non-linear modelling and prediction. Computational Management Science 1(2), 109–149 (2004)
45. Kosko, B.: Fuzzy systems as universal approximators. IEEE Transactions on Computers 43(11), 1329–1333 (1994)
46. Lee, S.-W., Verri, A. (eds.): SVM 2002. LNCS, vol. 2388. Springer, Heidelberg (2002)
47. Mantawy, A.H., Soliman, S.A., El-Hawary, M.E.: A new tabu search algorithm for the long-term hydro scheduling problem. In: 2002 Large Engineering Systems Conference on Power Engineering, LESCOPE 2002, pp. 29–34 (2002)
48. Mateo, F., Lendasse, A.: A variable selection approach based on the delta test for extreme learning machine models. In: Proceedings of the European Symposium on Time Series Prediction, pp. 57–66 (2008)
49. Michalewicz, Z.: Genetic algorithms + Data structures = Evolution programs, 3rd edn. Springer, Heidelberg (1996)
50. Hiroyasu, T., Miki, M., Negami, M.: Distributed genetic algorithms with randomized migration rate. In: Proceedings of the IEEE Conf. Systems, Man and Cybernetics, pp. 689–694 (1999)
51. Mitchell, M., Forrest, S.: Genetic algorithms and artificial life. Artificial Life 1(3), 267–289 (1995)
52. Oh, I.-S., Lee, J.-S., Moon, B.-R.: Local search-embedded genetic algorithms for feature selection. In: Proceedings of 16th International Conference on Pattern Recognition, vol. 2, pp. 148–151 (2002)
53. Oh, I.-S., Lee, J.-S., Moon, B.-R.: Hybrid genetic algorithms for feature selection. IEEE Trans. on Pattern Analysis and Machine Intelligence 26(11), 1424–1437 (2004)
54. Pi, H., Peterson, C.: Finding the embedding dimension and variable dependencies in time series. Neural Computation 6(3), 509–520 (1994)
55. Poggio, T., Girosi, F.: A theory of networks for approximation and learning, Tech. Report AI-1140, MIT Artificial Intelligence Laboratory, Cambridge, MA (1989)
56. Punch, W.F., Goodman, E.D., Pei, M., Chia-Shun, L., Hovland, P., Enbody, R.: Further research on feature selection and classification using genetic algorithms. In: Forrest, S. (ed.) Proc. of the Fifth Int. Conf. on Genetic Algorithms, pp. 557–564. Morgan Kaufmann, San Mateo (1993)
57. Raymer, M.L., Punch, W.F., Goodman, E.D., Kuhn, L.A., Jain, A.K.: Dimensionality reduction using genetic algorithms. IEEE Transactions on Evolutionary Computation 4(2), 164–171 (2000)
58. Reeves, C.R.: Using genetic algorithms with small populations. In: Forrest, S. (ed.) Proceedings of the Fifth International Conference on Genetic Algorithms, pp. 92–99. Morgan Kaufmann (1993)
59. Reeves, C.R.: Using genetic algorithms with small populations. In: Forrest, S. (ed.) Proc. of the Fifth International Conference on Genetic Algorithms, pp. 92–99. Morgan-Kauffman Publishers, Inc. (July 1993)
60. Rubio, G., Herrera, L.J., Pomares, H., Rojas, I., Guillén, A.: Design of specific-to-problem kernels and use of kernel weighted K-nearest neighbours for time series modelling. Neurocomputing 73(10-12), 1965–1975 (2010)

61. Saeys, Y., Inza, I., Larranaga, P.: A review of feature selection techniques in bioinformatics. Bioinformatics 23(19), 2507–2517 (2007)
62. Schaffer, J.D.: A study of control parameters affecting online performance of genetic algorithms for function optimization. In: Schaffer, J.D. (ed.) Proc. of the Third International Conference on Genetic Algorithms, pp. 51–60. Morgan-Kauffman Publishers, Inc. (June 1989)
63. Scheuerer, S.: A tabu search heuristic for the truck and trailer routing problem. Comput. Oper. Res. 33(4), 894–909 (2006)
64. Sywerda, G.: Uniform crossover in genetic algorithms. In: Proceedings of the Third International Conference on Genetic Algorithms, pp. 2–9. Morgan Kaufmann Publishers Inc., San Francisco (1989)
65. Thierens, D., Goldberg, D.E.: Mixing in genetic algorithms. In: Forrest, S. (ed.) Proc. of the Fifth International Conference on Genetic Algorithms, pp. 38–45. Morgan-Kauffman Publishers, Inc. (July 1993)
66. Wang, L., Kazmierski, T.J.: Vhdl-ams based genetic optimization of a fuzzy logic controller for automotive active suspension systems. In: Proceedings of the 2005 IEEE International Behavioral Modeling and Simulation Workshop, BMAS 2005, pp. 124–127 (2005)
67. Xu, J., Chiu, S., Glover, F.: A probabilistic tabu search for the telecommunications network design. Journal of Combinatorial Optimization, Special Issue on Topological Network Design 1, 69–94 (1996)
68. Xu, J., Chiu, S., Glover, F.: Using tabu search to solve steiner tree-star problem in telecommunications network design. Telecommunication Systems 6, 117–125 (1996)
69. Zhang, C., Li, P., Guan, Z., Rao, Y.Y.: A tabu search algorithm with a new neighborhood structure for the job shop scheduling problem. Computers & Operations Research 34(11), 3229–3242 (2007)
70. Zhang, J., Li, S., Shen, S.: Extracting minimum unsatisfiable cores with a greedy genetic algorithm. In: Proc. ACAI 2006, pp. 847–856 (2006)

# A Chemical Evolutionary Mechanism for Instantiating Service-Based Applications

Maurizio Giordano and Claudia Di Napoli

**Abstract.** Service Oriented Architecture (SOA) has become the *de facto* paradigm for the Internet of Services (IoS), i.e. a virtual space where information and content is stored, exchanged and manipulated by software and human entities through services. In this scenario, a Service Based Application (SBA) is a composition of a number of possibly independent services, that is software programs or interfaces for human entities connected through the network and performing a set of functionalities whose integration should fulfil the requirement of the SBA end-user. Therefore it becomes necessary to organize compositions of services on demand in response to dynamic requirements and circumstances. At this end the process of selecting service instances matching an SBA requested under certain conditions is modelled as an evolving chemical process that can react to environmental changes as they occur, so providing adaptability to non-functional characteristics changes. The chemical metaphor allows to approach the composition of services as a decentralized and incremental aggregation mechanism governed by local rules such that environmental changes affecting any part of SBA may be processed at any time.

## 1 Introduction

The Internet of Services (IoS) is becoming the substrate of a virtual space where information and content is stored, exchanged and manipulated by software and human entities through services. Service Oriented Architecture (SOA) has become the *de facto* paradigm for the IoS. Internet actors like enterprises and other organizations (social, educational, media, government, and so on.) take up the roles of both service providers and consumers, and services have to be composed in a way to fulfil their

Maurizio Giordano · Claudia Di Napoli
Istituto di Cibernetica "E. Caianiello" - CNR, Via Campi Flegrei 34, 80078 Pozzuoli, Naples - Italy
e-mail: c.dinapoli@cib.na.cnr.it,maurizio.giordano@cnr.it

F.F. de Vega et al. (Eds.): Parallel Architectures & Bioinspired Algorithms, SCI 415, pp. 267–286.
springerlink.com

needs. This view has changed the way enterprises do business, public and private organizations cooperate to solve and accomplish different tasks, people socialize and work together.

In this scenario, a Service Based Application (SBA) is a composition of a number of possibly independent services, that is software programs or interfaces for human entities, connected through the network and performing a set of functionalities whose integration should fulfil the requirement of the SBA end-user.

What is new in the SBA design and realization is that they should not rely on a centralized control mechanism. Moreover SBAs will be enacted in open and very dynamic settings, therefore their design should take this aspect into account: participant services of the composition may join/leave and fail/be-unavailable/degrade while the SBA is running. It is hard for an SBA designer to foresee all possible changes in service provisioning and status in order to perform at the right moment actions both to adapt services at runtime to guarantee SBA user requirements satisfaction, and to prevent possible SBA failures.

In the IoS scenario it is likely that more service providers can provide the same functionality at different conditions referring to non-functional characteristics of a provided service, like price, time to deliver, and so on. These may change in time depending on provider policies, and as such they cannot be advertised together with the service description nor planned at the composition design time. So, it becomes necessary to organize compositions of services on demand in response to dynamic requirements and circumstances.

SBA self-adaptability to unpredictable service provisioning and changes in service availability is a required feature service-based systems should support.

In the present contribution we address the problem of providing an *adaptive mechanism* to select service instances that match an SBA request specifying functionality of service components, their dependence constraints and some user's preferred non-functional characteristic values.

More specifically we use a chemical metaphor to model the problem of selecting service instances according to the required conditions as a decentralized and incremental aggregation mechanism governed by local rules [23]. In such a way environmental changes affecting any part of SBA may be processed at any time. Furthermore, applying this metaphor to the problem of selecting service instances allows to reduce the search space considerably, and more importantly, to model the service selection process as an evolving and always running mechanism that can adapt to environmental changes as they occur, so providing adaptability to non-functional characteristics changes.

## 2 The Chemical Computational Model

The $\gamma$-calculus [24, 12] is a formal definition of the chemical paradigm aimed at relaxing the artificial sequentializing of algorithms. The fundamental data structure of the $\gamma$-calculus is the *multiset*, i.e. a set that may contain multiple occurrences of the same element. Multisets are affected by so called reactions taking place independently and potentially simultaneously, according to local and actual conditions yielding a multiset rewriting system. There is no concept of centralized control, ordering, serialization, rather the computation is carried out in a not deterministic, inherently parallel, self-evolving way.

$\gamma$-terms (molecules) are:

1. constants, i.e. numbers, booleans or labels (strings)
2. variables $x$
3. $\gamma$-abstractions: $\gamma\langle x\rangle.M$
4. multisets of $\gamma$-terms: $M_1\ ,\ldots,\ M_m$
5. solutions: $\langle M\rangle$.

Juxtaposition of $\gamma$-terms is commutative ($M_1,M_2 \equiv M_2,M_1$) and associative ($M_1$, $(M_2,M_3) \equiv (M_1,M_2),M_3$). Commutativity and associativity are the properties that realize the 'Brownian-motion', i.e., the free distribution and unspecified reaction order among molecules that is a basic principle in the chemical paradigm [13].

$\gamma$-abstractions are the reactive molecules that operate on other molecules and replace them by reduction. Due to the commutative and associative rules, the order of parameters is not relevant; molecules, solutions participating in the reaction are extracted by pattern matching. The semantics of a $\gamma$-reduction is:

$$(\gamma\langle x\rangle.M),\langle N\rangle \rightarrow_\gamma M[x:=N] \tag{1}$$

where the two reacting terms on the left hand side are replaced by the body $M$ of the $\gamma$-abstraction where each free occurrence of variable $x$ is replaced by parameter $N$. Reactions may depend on certain conditions expressed as $C$ in $\gamma\langle x\rangle\lfloor C\rfloor.M$ that can be reduced only if $C$ evaluates to true. Reactions can capture multiple molecules in a single atomic step.

Besides associativity and commutativity, reactions are governed by: (i) law of locality, i.e. if a reaction can occur, it will occur in the same way irrespectively of the environment; and (ii) membrane law, i.e. reactions can occur in solutions containing sub-solutions separated by a membrane (*nested solutions*).

The $\gamma$-calculus is a *higher order* model, where $\gamma$-abstractions – just like any other molecules – can be passed as parameters or yielded as a result of a reduction.

The Higher Order Chemical Language (HOCL) [15] is a programming language based on the $\gamma$-calculus.

HOCL uses the self-explanatory **replace... by... if...** construct to express active molecules. **replace** $P$ **by** $M$ **if** $C$ formally corresponds to $\gamma(P)\lfloor C\rfloor.M$ with a major difference: while $\gamma$-abstractions are destroyed by the reactions, HOCL rules remain in the solution. **replace... by... if...** is followed by **in** $\langle ...\rangle$ that specifies the solution the active molecule floats in.

HOCL extends the $\gamma$-calculus with:

1. typed variables expressed by the notation $x ::< type >$, that can be used in patterns for matching the HOCL rules;

**Fig. 1** An HOCL program to compute the maximum number in a set of integers and its exeucution trace (bold molecules are the ones involved in the reaction)

```
1 let max = replace x :: int, y :: int
2           by x
3           if x > y
4           in ⟨max, 16, 5, 9, −6, 0, 3, 5, 0, 5, 9, −3, 1⟩
```

$$\begin{aligned}
&\textit{Step 1} && \langle \mathbf{max}, 16, 5, 9, -6, \mathbf{0}, 3, 5, 0, 5, 9, \mathbf{-3}, 1\rangle \\
&\textit{Step 2} \rightarrow && \langle \mathbf{max}, 16, 5, 9, \mathbf{-6}, 0, 3, 5, 0, \mathbf{5}, 9, 1\rangle \\
&\textit{Step 3} \rightarrow && \langle \mathbf{max}, \mathbf{16}, 5, 9, 0, 3, 5, \mathbf{0}, 5, 9, 1\rangle \\
&\quad \ldots && \\
&\textit{Step 10} \rightarrow && \langle \mathbf{max}, \mathbf{16}, \mathbf{9}\rangle \\
&\textit{Step 11} \rightarrow && \langle max, 16\rangle
\end{aligned}$$

2. tuples of molecules with the notation $M_1 : \ldots : M_n$,
3. names to identify and match active molecules in multisets and rules.

In Fig.1 the HOCL code defining an active molecule named *max* that captures a pair of numbers and replaces it with the greatest number is reported. The active molecule (rule) is floating in the solution specified at line 4.

## 3 Problem Formalization

It is assumed that users requiring SBAs submit their requests by specifying both the functionality of each component of the application, and the dependence constraints occurring among the components, i.e. the order of execution in which the components should be delivered. Users also provide a value representing a measure of some non-functional characteristics they would "prefer" the application to be delivered with. This value will be used to drive the selection of the suitable service components.

For example, the user may specify that he/she wants the composition to be delivered within a given deadline, or the money he/she is willing to pay to obtain the result of the application, or a measure of the reliability of the providers of the required services or more complicated features.

In the current approach, it is assumed that a single value is specified by the user at the time when the request is issued, representing a sort of Quality of Service (QoS) required for the application (all the dimensions required to develop a usable QoS model for a workflow of services are not investigated in this work).

It is also assumed that the QoS specified by the user for the entire composition can be related to a parameter for each component service in the abstract workflow. Of course it is not always possible to relate a global preference on a composition of services to each component service, but at the moment we refer to cases in which this exemplification is acceptable, i.e. where an additive property holds for the considered QoS.

The required functionalities together with their dependence constraints are expressed in the form of an *abstract workflow* (AW). The abstract workflow is a Directed Acyclic Graph (DAG) $AW = (S,E)$ where $S = \{s_i,\ldots,s_n\}$ is a set of nodes in the graph, and $E \subseteq S \times S$ is a set of directed edges in the graph. Each node represents a required *activity*, i.e. a service interface whose actual implementation can be provided by one or more services instances with different non-functional characteristics. Each directed edge represents a data, or a control (or both) dependence between two nodes it connects.

**Definition 1.** *Given an AW* $= (S,E)$*, a* ***path from*** $s_l$ ***to*** $s_p$ *is a set of AW nodes* $\{s_{m_1},\ldots,s_{m_k}\}$ *such that* $(s_{m_i},s_{m_{i+1}}) \in E\ \forall i \in \{1,\ldots,k-1\}$ *and* $s_l = s_{m_1}$ *and* $s_p = s_{m_k}$*; the nodes* $s_l$ *and* $s_p$ *are named respectively the* ***first*** *and* ***last*** *nodes of the path.*

AW nodes are ordered, i.e. for any path $\{s_{m_1},\ldots,s_{m_k}\}$, $m_i < m_{i+1}\ \forall i \in \{1,\ldots,k-1\}$.

It is assumed that for each node in the AW the corresponding in/out-degrees assume the values 0, 1, and 2. This is not a constraint, since nodes with outdegree greater than 2 may be either *or-nodes* or *and-nodes*; it is simply to transform an or(and)-node with an outdgree of $n$, in a sequence of $n-1$ or(and) nodes with and outdegree of 2. From the workflow execution perspective, a $n$-outdegree or-node is typically a *switch-case* construct, while a $n$-outdegree and-node is a *split* construct, where all branches may be concurrently executed.

In the AW there are four types of nodes: (1) a *start* node with an in-degree equal to 0; (2) a *stop* node with an out-degree equal to 0; (3) a *split* node with an out-degree equal to 2; (4) a *merge* node with an in-degree equal to 2.

Service providers able to provide the required AW activities make them available as *offers* specifying both the end point of a service implementing the required activity, and the value of the QoS parameter representing the non-functional characteristics they can provide the service with (see Fig.2). We denote with $\mathscr{O}_i$ the set of offers provided for the same activity $s_i \in S$, and with $\overline{o}_i \in \mathscr{O}_i$ a selected offer for the activity $s_i$.

Services matching the user's requirements have to be selected to obtain the actual workflow to be enacted, that we refer to as an *instantiated workflow* (IW) that is defined as follows:

**Definition 2.** *Given an AW* $= (S,E)$*, an* ***Instantiated Workflow of AW*** *is the graph* $IW = (\omega,\varepsilon)$ *where:*

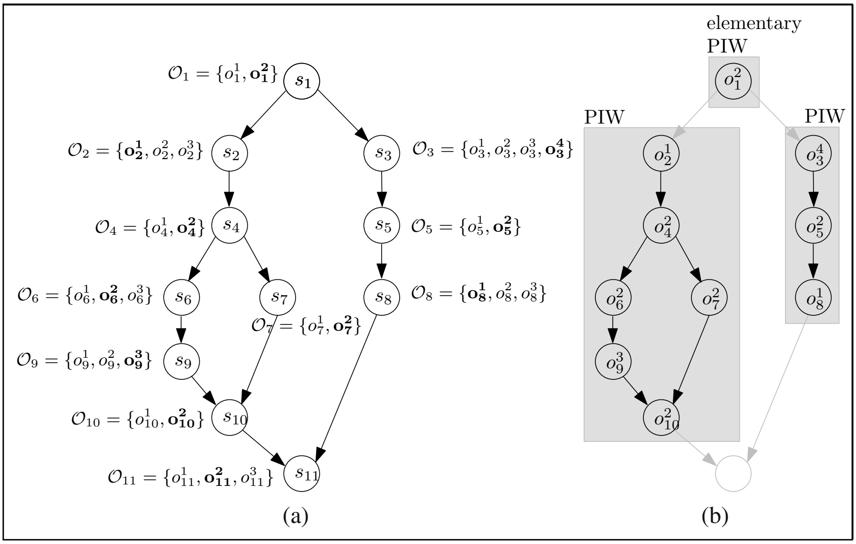
- $\omega = \{\overline{o}_1,\overline{o}_2,\ldots,\overline{o}_n\} \in \mathscr{O}_i \times \mathscr{O}_2 \times \ldots \times \mathscr{O}_n$*, such that each* $\overline{o}_i \in \mathscr{O}_i$ *is a selected and unique offer corresponding to the AW activity* $s_i$*, and*

- *$\varepsilon \subseteq \omega \times \omega$ and $(\overline{o}_k, \overline{o}_l) \in \varepsilon$ if and only if $(s_k, s_l) \in E$, i.e. there is a directed edge between two offers if and only if there is a directed edge between the corresponding activities.*

For example with respect to the AW reported in Fig.2(a), the $IW = (\omega, \varepsilon)$ corresponding to the selected offers (typed in bold) is:

$$\omega = \{o_1^2, o_2^1, o_3^4, o_4^2, o_5^2, o_6^2, o_7^2, o_8^1, o_9^3, o_{10}^2, o_{11}^2\}$$
$$\varepsilon = \{(o_1^2, o_2^1), (o_1^2, o_3^4), (o_2^1, o_4^2), (o_3^4, o_5^2), (o_4^2, o_6^2)(o_4^2, o_7^2), (o_7^2, o_{10}^2), (o_5^2, o_8^1),$$
$$(o_6^2, o_9^3), (o_9^3, o_{10}^2), (o_{10}^2, o_{11}^2), (o_8^1, o_{11}^2)\}$$

According to the number of service instances available for each node in the graph, and to the value of the QoS parameter of each instance, it will be possible to find zero or more instantiated workflows.



$\mathcal{O}_1 = \{o_1^1, \mathbf{o_1^2}\}$
$s_1$
$\mathcal{O}_2 = \{\mathbf{o_2^1}, o_2^2, o_2^3\}$
$s_2$
$s_3$
$\mathcal{O}_3 = \{o_3^1, o_3^2, o_3^3, \mathbf{o_3^4}\}$
$\mathcal{O}_4 = \{o_4^1, \mathbf{o_4^2}\}$
$s_4$
$s_5$
$\mathcal{O}_5 = \{o_5^1, \mathbf{o_5^2}\}$
$\mathcal{O}_6 = \{o_6^1, \mathbf{o_6^2}, o_6^3\}$
$s_6$
$s_7$
$s_8$
$\mathcal{O}_8 = \{\mathbf{o_8^1}, o_8^2, o_8^3\}$
$\mathcal{O}_7 = \{o_7^1, \mathbf{o_7^2}\}$
$\mathcal{O}_9 = \{o_9^1, o_9^2, \mathbf{o_9^3}\}$
$s_9$
$\mathcal{O}_{10} = \{o_{10}^1, \mathbf{o_{10}^2}\}$
$s_{10}$
$\mathcal{O}_{11} = \{o_{11}^1, \mathbf{o_{11}^2}, o_{11}^3\}$
$s_{11}$
(a)
elementary PIW
$o_1^2$
PIW
PIW
$o_2^1$
$o_3^4$
$o_4^2$
$o_5^2$
$o_6^2$
$o_7^2$
$o_8^1$
$o_9^3$
$o_{10}^2$
(b)

**Fig. 2** (a) Abstract Workflow with offers associated to its activities; (b) examples of partially intantiated workflows

A path in an IW is defined in the same way as for an AW:

**Definition 3.** *Assumed that $IW = (\omega, \varepsilon)$ is an instantiated workflow of AW and that $\{s_{m_1}, \dots, s_{m_k}\}$ is an AW path from $s_{m_1}$ to $s_{m_k}$, the corresponding* ***IW path*** *is $\{\overline{o}_{m_1}, \dots, \overline{o}_{m_k}\}$.*

Once at least one IW is computed, its execution can take place. An IW execution path is defined as follows:

**Definition 4.** *Given an IW* $= (\omega = \{\overline{o}_1, \ldots, \overline{o}_n\}, \varepsilon)$*, an* ***IW execution path*** *is any IW path from the offer* $\overline{o}_1$ *of the start node to the offer* $\overline{o}_n$ *of the end node.*

Since any path in the AW is ordered, also the corresponding IW path is ordered.

The process that leads from an AW to an IW is the *workflow instantiation process*, and it is computed in terms of chemical reactions [23]. The basic mechanism consists of the application of local chemical rules that aggregate service offers according to the AW structure in an incremental and recursive way. Each rule application produces the instantiation of an AW fragment so that it can be incrementally aggregated with other fragments leading to larger instantiated fragments.

An instantiated AW fragment is named a *partially instantiated workflow* (PIW) and its definition is the following:

**Definition 5.** *Given an AW* $= (S, E)$ *and for each AW node* $s_i$ *let* $\mathcal{O}_i$ *be the set of its offers, a* ***Partially Instantiated Workflow*** *of AW can be:*

1. *the graph PIW* $= (\{\overline{o}_i\}, \emptyset)$*, where* $\overline{o}_i \in \mathcal{O}_i$*, i.e. the graph containing no edges and a single node* $\overline{o}_i$ *which is a service offer of any AW node (****elementary PIW****),*
2. *the graph PIW* $= (\omega\prime, \varepsilon\prime)$*, where:*
   - $\omega\prime = \{\overline{o}_{m_1}, \ldots, \overline{o}_{m_k}\} \in \mathcal{O}_{m_1} \times \ldots \times \mathcal{O}_{m_k}$*, i.e. a set of service offers of the AW such that any AW path from node* $s_{m_1}$ *to* $s_{m_k}$ *contains only nodes in the set* $\{s_{m_1}, \ldots, s_{m_k}\} \subseteq S$*,*
   - $\varepsilon\prime \subseteq \omega\prime \times \omega\prime$ *and* $(\overline{o}_p, \overline{o}_l) \in \varepsilon\prime$ *if and only if* $(s_p, s_l) \in E$*.*

*The nodes* $\overline{o}_{m_1}$ *and* $\overline{o}_{m_k}$ *are named respectively the* ***first*** *and* ***last*** *node of the PIW.*

The PIW is the building block of the chemical-based instantiation process, and its formal definition corresponds in the control flow analysis theory [1] to the *basic-block* definition. In fact, a PIW is an instantiation of an AW fragment where the control flow enters only the first activity, and it comes out only from the last activity of the fragment.

For example, Fig.2(b) reports two possible PIWs of the AW of Fig.2(a): The leftmost PIW is the graph $(\omega\prime, \varepsilon\prime)$ where:

$$\omega\prime = \{o_2^1, o_4^2, o_6^2, o_7^2, o_9^3, o_{10}^2\}$$
$$\varepsilon\prime = \{(o_2^1, o_4^2), (o_4^2, o_6^2)(o_4^2, o_7^2), (o_6^2, o_9^3), (o_9^3, o_{10}^2), (o_7^2, o_{10}^2)\}.$$

where $o_2^1$ is the *first* node while $o_{10}^2$ is the *last* node. The node $o_4^2$ is a *split* node since it has an out-degree of 2, while the node $o_{10}^2$ is a *merge* node since it has an in-degree of 2. According to the PIW definition, every path from $o_2^1$ to $o_{10}^2$ contains nodes all in $\omega\prime$.

Of course an IW is a PIW whose first and last nodes correspond respectively to the start and stop nodes of the AW.

Three relations characterize nodes connectivity in the AW graph:

**Definition 6.** *Two AW nodes $s_p$ and $s_l$ are* ***chainable*** *if and only if $(s_p, s_l) \in E \wedge \neg(\exists(s_p, s_k) \in E : s_k \neq s_l \vee \exists(s_{k\prime}, s_l) \in E : s_{k\prime} \neq s_p)$, i.e. there is an edge from $s_p$ to $s_k$ but there is no other edge either outgoing from $s_p$ or incoming to $s_l$.*

The boolean function *chainable*$(s_p, s_l)$ is introduced and it evaluates to true if the AW nodes $s_p$ and $s_l$ are chainable according to the above definition.

**Definition 7.** *An AW node $s_p$ is a* ***split*** *to nodes $s_l$ and $s_m$ if and only if $(s_p, s_l), (s_p, s_m) \in E \wedge \neg(\exists(s_k, s_l) \in E : s_k \neq s_p \vee \exists(s_{k\prime}, s_m) \in E : s_{k\prime} \neq s_p)$, i.e. the node $s_p$ is the only source for nodes $s_l$ and $s_m$.*

**Definition 8.** *An AW node $s_p$ is a* ***merge*** *of nodes $s_l$ and $s_m$ if and only if $(s_l, s_p), (s_m, s_p) \in E \wedge \neg(\exists(s_l, s_k) \in E : s_k \neq s_p \vee \exists(s_m, s_{k\prime}) \in E : s_{k\prime} \neq s_p)$, i.e. the node $s_p$ is the only sink of both $s_l$ and $s_m$.*

The boolean functions *splitto*$(s_p, s_l, s_m)$ and *mergefrom*$(s_p, s_l, s_m)$ are introduced to check, respectively, if the AW nodes $s_p$, $s_l$ and $s_m$ satisfies the relations of Def.7 and Def.8.

## 4 Chemical Representation of the Problem

A chemical solution represents the environment in which active and passive molecules are inserted. The solution is the context of a computation: passive molecules represent the data of the computation, and active molecules represent the computation itself, i.e. reaction rules that apply on molecules that match the rule condition and that may consume molecules and/or produce new molecules in the same solution as outcome. Active molecules may remove/add new reaction rules in the solution thus changing the behaviour of the computation.

To represent the AW instantiation process in terms of chemical reactions, passive molecules are used to represent *service offers* associated to AW nodes. The QoS parameter of each offer is taken into account by the chemical-based selection process to *incrementally* and *recursively* build a partially instantiated workflow until an IW is computed, if possible. The IW will be computed if there are offers for all AW nodes and if the selection criteria allow to find at least one IW that satisfies the QoS requirements specified by the user requiring the service composition. In the chemical-based service instantiation process, when no more reactions can take place with the molecules available in the system, an inert state is reached with zero or more IWs produced, and PIWs.

### *4.1 Abstract Workflow and Service Offers as Chemical Molecules*

The AW structure provides the topological relations between AW nodes, i.e. edges between two nodes and in/out-degree of each node. This information is relevant to the workflow instantiation (and execution) phase and it drives the chemical rules applications while aggregating PIWs. In our approach the AW node connectivity is not explicitly expressed according to the chemical formalism. Instead, the condition part of the chemical rules computes this information by means of boolean functions with the same semantics of relations defined in Def.6, Def.7 and Def.8.

In the chemical notation two data structures are introduced: the service offers $o_i^{j_i}$ and the PIWs that are the outputs of the instantiation process.

The chemical notation of a service offer is the tuple:

$$o_i \equiv url_i : s_i : c_i \tag{2}$$

where:

1. $url_i$ is a string representing the url of a specific service implementation for the AW activity $s_i$,
2. $s_i$ is the identifier of the corresponding AW activity,
3. $c_i$ is the value of a QoS parameter representing the condition under which the service implementation is provided.

So, for each AW activity $s_i$ a set of actual service implementations may be available, $url_i^1, \ldots, url_i^{m_i}$.

Multiple service implementations, or the same service implementation with different QoS values, correspond to different service offers. For example, the same AW activity `hotelbooking` can be provided by different organizations with different costs, or the same service implementation can be provided by the same organization with different costs:

```
"http://provider1/htlbook":hotelbooking:5euros
"http://provider1/htlbook":hotelbooking:4euros
"http://provider12/hotelbk":hotelbooking:5euros
...
```

## 4.2 Instantiating Workflow as Chemical Reactions

To model the instantiation process in terms of chemical reactions, in addition to the molecules representing the offered services, also PIWs that are built during the chemical reactions are represented in the chemical formalism.

The chemical molecule representing a PIW is the following:

$$\langle \texttt{first} : url_i : s_i,\ \texttt{last} : url_j : s_j,\ node : url_k : s_k, \ldots,\ \texttt{qos} : c \rangle \tag{3}$$

where:

- `first` $: url_i : s_i$ is a tuple representing the service offer selected for the first node of the PIW,
- `last` $: url_j : s_j$ is a tuple representing the service offer selected for the last node of the PIW,

- $node : url_k : s_k, \ldots$ represents a set of intermediate nodes of the PIW (if any), where *node* can be one of three possible labels:
  - `split` for a split node;
  - `merge` for a merge node;
  - `seq` for a node that is not either a split or a merge node;
- $\texttt{qos} : c$ is the value of the considered non-functional parameter associated to the PIW obtained by combining the QoS values of the composed PIWs.

For an elementary PIW the notation is:

$$\langle \texttt{first} : url_i : s_i,\ \texttt{last} : url_i : s_i,\ \texttt{qos} : c_i \rangle \tag{4}$$

that contains a service offer for the single node $s_i$, and $c_i$ is the associated QoS value.

Let's consider the chemical representation of the (leftmost) PIW reported in Fig.2(b):

$$\begin{aligned}\langle &\texttt{first:url\_2\_1:s2, last:url\_10\_2:s10,}\\ &\texttt{split:url\_4\_2:s4, merge:url\_10\_2:s10,}\\ &\texttt{seq:url\_6\_2:s6,seq:url\_9\_3:s9,}\\ &\texttt{seq:url\_7\_2:s7,}\\ &\texttt{qos:12euros}\rangle\end{aligned} \tag{5}$$

where the `split` and `merge` labels identify the *split* and *merge* nodes of the PIW. In this example the node `url_10_2:s10` is represented twice since it is both a merge node and the last node of the PIW.

In Eq.5 the molecules `seq:url_6_2:s6` and `seq:url_9_3:s9` represent the left branch of the *split* node, while the molecule `seq:url_7_2:s7` represents the right branch of the same split. Again the `qos` parameter is a combination of the QoS values of the component nodes.

In the following subsections the chemical rules necessary to concatenate PIWs without branches and to aggregate PIWs containing *split/merge* nodes are described.

## 5 Workflow Instantiation as a Chemical Process

In order to obtain IWs to be enacted, a process to select the suitable service offers (and thus their actual implememtations) takes place. This process is expressed in terms of chemical reactions that occur when some conditions are satisfied, i.e. as long as there are molecules that match the rule conditions. When no more chemical reactions can take place an inert state is reached. IWs are created when service endpoints are assigned to all the corresponding nodes of the requested AW. All rules may apply concurrently. In fact, reactions are governed only by the availability of the suitable service offers, and they take place in a not deterministic way.

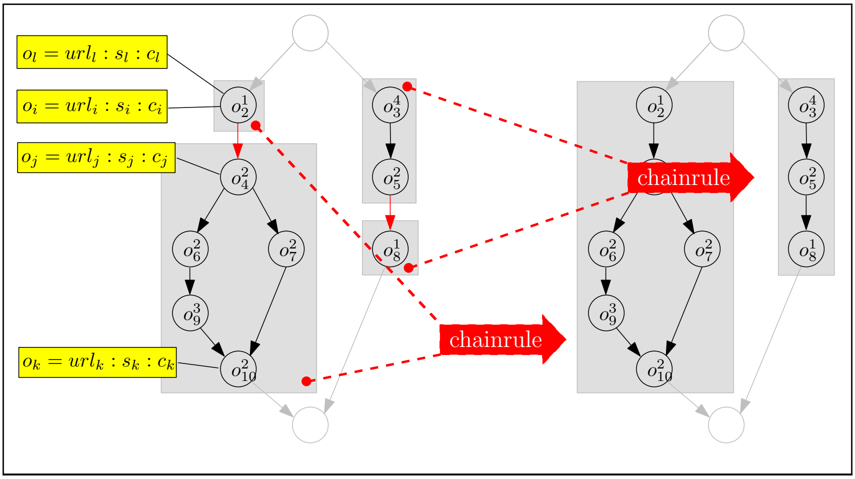


**Fig. 3** Two `chainrule` reactions

## 5.1 *Building Elementary PIWs*

To apply the reaction rules to incrementally build the PIWs, it is necessary to convert the molecules representing service offers into elementary PIWs. The chemical rule performing this transformation, called `prerule`, is:

$$\begin{aligned} \textbf{let}\ \texttt{prerule} = {} & \textbf{replace}\ url : s : c \\ & \textbf{by}\quad \langle \texttt{first} : url : s,\ \texttt{last} : url : s,\ \texttt{qos} : c \rangle \end{aligned} \tag{6}$$

where the **replace** part matches a molecule representing a service offer, and the **by** part, that is the action of the rule, has the effect of introducing in the chemical system a new molecule representing the elementary PIW corresponding to that offer. The `prerule` matches all service offers and thus it applies on all of them, i.e. no condition (**if** part) is specified to trigger the reaction.

After the transformation is done, all elementary PIWs are ready to be matched by other chemical rules to react, so that they can be composed in more complex PIWs, and finally in IWs if possible.

## 5.2 *Concatenating PIWs*

Elementary PIWs are the starting building blocks of the workflow instantiation process. Two PIWs can be chained together to form a new PIW containing a sequence of consecutive nodes. The sequences can be further concatenated producing longer sequences.

Two PIWs can be concatenated if the first one ends with a node that is not a *split* node, and the second one starts with a node that is not a *merge* node. The rule that concatenates PIWs is named `chainrule` and it is reported in Eq.7.

$$\begin{array}{lll}\textbf{let}\ \texttt{chainrule} = & \textbf{replace} & \langle \texttt{first}: url_l : s_l,\ \texttt{last}: url_i : s_i,\ \texttt{qos}: c_1,\ \omega_1\rangle, \\ & & \langle \texttt{first}: url_j : s_j,\ \texttt{last}: url_k : s_k,\ \texttt{qos}: c_2,\ \omega_2\rangle, \\ & \textbf{by} & \langle \texttt{first}: url_l : s_l,\ \texttt{last}: url_k : s_k, \\ & & \qquad \texttt{seq}: url_i : s_i,\ \texttt{seq}: url_j : s_j, \\ & & \qquad \texttt{qos}: \phi(c_1,c_2),\ \omega_1,\ \omega_2\rangle \\ & \textbf{if} & chainable(s_i,s_j) = \textbf{true} \wedge \psi(c_1,c_2) = \textbf{true}\end{array} \tag{7}$$

where the **replace** part of the rule matches two molecules representing the PIWs that can be chained through an edge if the conditions specified in the **if** part of the rule holds.

The **if** part of the rule is a conjunction of the following sub-conditions:

1. nodes $s_i$ and $s_j$ of the AW are "chainable" according to Def.6. If this condition holds, there is only one outgoing edge from the last node ($o_i = url_i : s_i : c_i$) of the first PIW to the first node ($o_j = url_j : s_j : c_j$) of the second PIW.
2. $\psi(c_1,c_2) =$ **true**, that is the QoS values of the PIWs satisfy a certain boolean condition expressed by the function $\psi$.

The **by** part of the rule is the action resulting in a new PIW that is the concatenation of the two input PIWs in the rule, and its QoS parameter is obtained by combining the QoS parameters of the two input PIWs according to a generic function $\phi$.

The input PIWs transformed by the `chainrule` are no longer in the chemical solution. The rule in Eq.7 may generate several PIWs starting from the same input AW nodes by matching other available service offers.

The $\omega$ symbols are wildcards matching anything inside the input molecules that is not relevant for the reaction to take place: all information matching the wildcards is reinserted in the new produced molecule, so intermediate nodes (if any) of the input PIWs are reinserted (following the output pattern) in the generated PIW. Two examples of `chainrule` applications are reported in Fig.3.

### *5.3 Processing Split/Merge Nodes*

To concatenate two PIWs where the first one ends with a *split* node and the second one starts with a *merge* node, a new chemical rule is introduced, called `splitrule`. This rule links together four PIWs, one ending with a *split* node, one starting with the corresponding *merge* node, and two ones representing the sub-graphs of the right and left branch of the split (see Fig.4).

The formula of the `splitrule` is the following:

$$
\begin{array}{lll}
\textbf{let} \ \texttt{splitrule} = & & \\
\quad \textbf{replace} & \langle \texttt{first} : url_i : s_i,\ \texttt{last} : url_k : s_k,\ \texttt{qos} : c_1,\ \omega_1 \rangle, & \\
& \langle \texttt{first} : url_l : s_l,\ \texttt{last} : url_j : s_j,\ \texttt{qos} : c_2,\ \omega_2 \rangle, & \\
& \langle \texttt{first} : url_p : s_p,\ \texttt{last} : url_q : s_q,\ \texttt{qos} : c_3,\ \omega_3 \rangle, & \\
& \langle \texttt{first} : url_r : s_r,\ \texttt{last} : url_t : s_t,\ \texttt{qos} : c_4,\ \omega_4 \rangle, & \\
\quad \textbf{by} & \langle \texttt{first} : url_i : s_i,\ \texttt{last} : url_j : s_j, & \\
& \quad \texttt{split} : url_k : s_k,\ \texttt{merge} : url_l : s_l, & \\
& \quad \texttt{qos} : \theta(c_1, c_2, c_3, c_4),\ \omega_1,\ \omega_2,\ \omega_3,\ \omega_4 \rangle, & \\
\quad \textbf{if} & splitto(s_k, s_p, s_r) = \textbf{true} \wedge mergefrom(s_l, s_q, s_t) = \textbf{true} & \\
& \wedge \psi(c_1, c_2, c_3, c_4) = \textbf{true} &
\end{array}
\tag{8}
$$

where the **replace** part of the rule specifies four molecules representing the PIWs to be linked around the *split* and *merge* nodes $s_k$ and $s_l$ (see Fig.4).

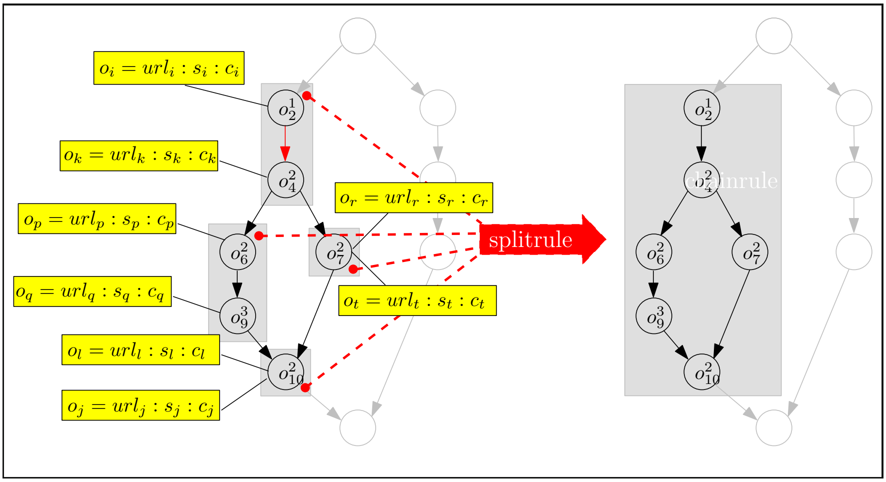

$o_i = url_i : s_i : c_i$
$o_2^1$
$o_k = url_k : s_k : c_k$
$o_4^2$
$o_r = url_r : s_r : c_r$
$o_p = url_p : s_p : c_p$
$o_6^2$
$o_7^2$
splitrule
$o_q = url_q : s_q : c_q$
$o_t = url_t : s_t : c_t$
$o_9^3$
$o_l = url_l : s_l : c_l$
$o_{10}^2$
$o_j = url_j : s_j : c_j$
$o_2^1$
chainrule
$o_4^2$
$o_6^2$
$o_7^2$
$o_9^3$
$o_{10}^2$

**Fig. 4** A `splitrule` reaction

More specifically, the four PIWs are in order: the PIW ending with the *split* node $s_k$, the one starting with the *merge* node $s_l$, the two PIWs representing the branches originating from the *split* and converging to the *merge* node. These conditions are verified in the **if** part by checking the boolean values of the functions $splitto(s_k, s_p, s_r)$ and $mergefrom(s_l, s_q, s_t)$ (see Def.7 and Def.8). The **if** part of the rule also checks if the input QoS values of the components (once combined) satisfy some general property (espressed by the $\psi$ function) to meet the QoS requirement specified in the user request.

The **by** part of the rule produces the PIW obtained by combining the four molecules around the split and merge nodes denoted by Eq.5.

# 6 Lazy Instantiation of Workflows

AW nodes may be characterized by additional attributes that are necessary for the execution phase once the nodes have been instantiated. In this section we introduce the possibility to exploit node attributes that are necessary for the execution phase, also for the AW instantiation process. In particular, we extend the proposed chemical model to make it possible to compute IWs also when there are missing offers in one of the branches of a *split* node that refers to if-then statements. In fact, in such cases it would be still possible to find an execution path for an IW although the IW is not yet fully instantiated.

## 6.1 Missing Services as Future Molecules

In the chemical notation a molecule named *service future* is introduced, that represents a "placeholder" for an actual service offer that is not yet available in the chemical system when the AW instantiation starts and that could enter the system during (or after) the instantiation finishes.

The chemical notation of a service future is the tuple:

$$o_i \equiv \texttt{future} : s_i : \texttt{undef} \tag{9}$$

In the chemical notation the `future` value will be replaced by actual service implementation when necessary. This new molecule is used to generate a PIW with some not instantiated parts.

Note that an `undef` QoS value is assigned to the placeholder since it does not represent a service implementation but just a *dummy* offer that will allow the composition process to continue in case of missing offers.

The elementary PIW notation that supports futures is the following:

$$\begin{gathered}\langle \texttt{first} : \texttt{future} : s_i,\ \texttt{last} : \texttt{future} : s_i, \\ \texttt{qos} : \texttt{undef},\ \texttt{canexec} : \textbf{false},\ \texttt{fixno} : 1\rangle\end{gathered} \tag{10}$$

In this notation the label `canexec` has a boolean value specifying if the PIW has at least one execution path. In case of elementary PIW the single node is a future placeholder, thus its execution is not possible. The label `fixno` is a counter for the number of future occurrences, i.e. how many placeholders (to be fixed) are in the molecule that represents the PIW.

In the case the chemical system is used to find IWs also with missing offers, then all the rules previously defined have to be modified to be compatible with the future mechanism. In particular, the prerule of Eq.6 has to generate elementary PIWs either with an available offer or with a placeholder for a missing offer with the same format. Therefore, two rules are introduced in Eq.11 and Eq.12.

$$\begin{aligned}\textbf{let}\ \texttt{prerule} = \ &\textbf{replace}\ url_i : s_i : c_i\\ &\textbf{by}\quad \langle \texttt{first} : url_i : s_i,\ \texttt{last} : url_i : s_i,\\ &\qquad\quad \texttt{qos} : c_i,\ \texttt{canexec} : \textbf{true},\ \texttt{fixno} : 0\rangle\\ &\textbf{if}\quad url_i \neq \texttt{future}\end{aligned} \tag{11}$$

$$\begin{aligned}\textbf{let}\ \texttt{prerule}\star = \ &\textbf{replace}\ \texttt{future} : s_i : c_i\\ &\textbf{by}\quad \langle \texttt{first} : url_i : s_i,\ \texttt{last} : url_i : s_i,\\ &\qquad\quad \texttt{qos} : c_i,\ \texttt{canexec} : \textbf{false},\ \texttt{fixno} : 1\rangle\end{aligned} \tag{12}$$

## 6.2 Aggregating Lazy PIWs

The chainrule of Eq.7 needs to be modified to compute an output PIW (i.e. the result of the chaining) with the new number of pending futures, and to determine the canexec attribute of the chain (i.e. the boolean AND). The new chainrule is reported in Eq.13

$$\begin{aligned}\textbf{let}\ \texttt{chainrule} = \ &\textbf{replace}\ \langle \texttt{first} : url_l : s_l,\ \mathit{last} : url_i : s_i,\ \texttt{qos} : c_1,\\ &\qquad\quad \texttt{canexec} : b1,\ \texttt{fixno} : f1,\ \omega_1\rangle,\\ &\qquad\quad \langle \texttt{first} : url_j : s_j,\ \texttt{last} : url_k : s_k,\ \texttt{qos} : c_2,\\ &\qquad\quad \texttt{canexec} : b2,\ \texttt{fixno} : f2, \omega_2\rangle,\\ &\textbf{by}\quad \langle \texttt{first} : url_l : s_l,\ \texttt{last} : url_k : s_k,\\ &\qquad\quad \texttt{seq} : url_i : s_i,\ \texttt{seq} : url_j : s_j,\ \texttt{qos} : \phi(c_1, c_2),\\ &\qquad\quad \texttt{canexec} : b1 \wedge b2,\ \texttt{fixno} : f1 + f2,\ \omega_1,\ \omega_2\rangle\\ &\textbf{if}\quad \mathit{chainable}(s_i, s_j) = \textbf{true} \wedge \psi(c_1, c_2) = \textbf{true}\end{aligned} \tag{13}$$

Since the new rule applies to PIWs with futures, it is assumed that the $\phi(c_1,c_2)$ returns an undefined value if some arguments are undefined.

In the same way, the splitrule has to be changed to process the canexec and fixno molecules included in each chemical solution representing the four arguments of the rule. So, the rule condition includes an additional check to verify that at least one branch can be executed, i.e. it is either fully instantiated or it contains nested split/merge constructs with at least one instantiated execution path.

The PIW produced by this rule, called *lazy PIW*, could be passed to an enactment engine for execution. The use of futures allows to postpone the problem of associating an actual service implementation to an activity of the AW at runtime, or during the chemical instantiation process as soon as the new service offers enter the chemical system.

$$
\begin{array}{lll}
\textbf{let}\ \texttt{splitrule} = \textbf{replace} & & \langle \texttt{first}: url_i : s_i,\ \texttt{last}: url_k : s_k,\ \texttt{qos}: c_1, \\
 & & \quad \texttt{canexec}: b1,\ \texttt{fixno}: f1,\ \omega_1\rangle, \\
 & & \langle \texttt{first}: url_l : s_l, \texttt{last}: url_j : s_j,\ \texttt{qos}: c_2, \\
 & & \quad \texttt{canexec}: b2,\ \texttt{fixno}: f2,\ \omega_2\rangle, \\
 & & \langle \texttt{first}: url_p : s_p,\ \texttt{last}: url_q : s_q, \texttt{qos}: c_3, \\
 & & \quad \texttt{canexec}: b3,\ \texttt{fixno}: f3,\ \omega_3\rangle, \\
 & & \langle \texttt{first}: url_r : s_r,\ \texttt{last}: url_t : s_t, \texttt{qos}: c_4, \\
 & & \quad \texttt{canexec}: b4,\ \texttt{fixno}: f4,\ \omega_4\rangle, \\
 & \textbf{by} & \langle \texttt{first}: url_i : s_i,\ \texttt{last}: url_j : s_j, \\
 & & \quad \texttt{split}: url_k : s_k,\ \texttt{merge}: url_l : s_l, \\
 & & \quad \texttt{canexec}: \textbf{true},\ \texttt{fixno}: f1 + f2 + f3 + f4, \\
 & & \quad \texttt{qos}: \theta(c_1, c_2, c_3, c_4),\ \omega_1,\ \omega_2,\ \omega_3,\ \omega_4\rangle, \\
 & \textbf{if} & splitto(s_k, s_p, s_r) = \textbf{true} \wedge mergefrom(s_l, s_q, s_t) = \textbf{true} \\
 & & \wedge (b3 \vee b4) \wedge \psi(c_1, c_2, c_3, c_4) = \textbf{true}
\end{array}
\tag{14}
$$

### 6.3 Fixing Future Services

Suppose that, when the instantiation process starts, some service offers are missing, so they are replaced by service futures. In this case, a new chemical rule is necessary to fix service futures when missing offers become available during the chemical system evolution. The formalization of this rule is reported in Eq.15

$$
\begin{array}{lll}
\textbf{let}\ \texttt{fixfuture} = & \textbf{replace} & url_i : s_i : c_i \\
 & & \langle node : \texttt{future} : s_i,\ \texttt{fixno}: n,\ \omega\rangle \\
 & \textbf{by} & \langle node : url_i : s_i,\ \texttt{fixno}: n - 1,\ \omega\rangle
\end{array}
\tag{15}
$$

The rule takes the new offer for the activity $s_i$ and it tries to match a PIW with a future placeholder associated to the same activity node. The rule rewrites the PIW by inserting the new service implementation, and decreasing the future occurrence counter.

To give an example of the `fixfuture` rule behaviour, let's suppose that an intermediate (not inert) state of the chemical system is:

$$
\begin{array}{l}
\langle \texttt{prerule},\ \texttt{chainrule},\ \texttt{splitrule},\ \texttt{fixfuture}, \\
\quad \mathbf{\texttt{url\_9\_1}} : \mathbf{\texttt{s9}} : \mathbf{\texttt{5euros}}, \\
\quad \texttt{url\_1\_3} : \texttt{s1} : \texttt{6euros}, \\
\quad \ldots \textit{other just-arrived offers} \ldots, \\
\quad \langle \texttt{first}: \texttt{url\_1\_2} : \texttt{s1},\ \texttt{last}: \texttt{url\_11\_7} : \texttt{s11},\ \texttt{seq}: \mathbf{\texttt{future}} : \mathbf{\texttt{s9}},\ \texttt{fixno}: \texttt{2}\ \ldots\rangle \\
\quad \langle \texttt{first}: \texttt{url\_3\_1} : \texttt{s3},\ \texttt{last}: \texttt{url\_8\_5} : \texttt{s8},\ \texttt{seq}: \texttt{url\_5\_4} : \texttt{s5},\ \texttt{fixno}: \texttt{0},\ \ldots\rangle \\
\quad \ldots \textit{other PIWs with (or without) futures} \ldots, \\
\rangle
\end{array}
\tag{16}
$$

where the node `s9` was replaced by a future placeholder since there were not associated offers when the instantiation process started.

At this intermediate stage the system is still active since a new service offer, that is the tuple `url_9_1:s9:5euros`, has entered the chemical solution, and it can be used by the `fixfuture` rule to replace the placeholder in the lazy IW.

We point out that the QoS parameter of the later incoming offer is not processed in the reaction. Currently we have not addressed the problem of combining QoS parameters in case of some missing instances, but we plan to investigate possible solutions in future works.

The future-fixing mechanism may reactivate the chemical system once it reached an inert state with only partially instantiated workflows[1] (with or without futures), as soon as new offers become available. So a complete instantiation of the abstract workflow that was not possible when the process started can be computed.

## 7 Related Works

Recently, the problem of dynamically selecting partner services composing a Service Based Application has gained wide attention since services are provided in highly changing and evolving environments like the Internet of Services. Also, several competing services may coexist implementing the same functionality, but with different QoS attributes (e.g., response time, reliability, cost, and so on) that are usually not static and so their values may change for several reasons.

Some research efforts address this problem by focusing on the development of automatic mechanisms to select appropriate services to build service compositions relying on languages and ontologies to represent service non-functional characteristics and providing selection algorithms that take them into account ([19, 18]).

Other research works have studied the development of frameworks to dynamically select service implementations. The Sword project [21] explores techniques for composing services using logical rules to express the inputs and outputs associated with services. A rule-based expert system is used to automatically determine whether a process could be implemented with the given services. It also returns a process plan that implements the composition. Maximilien et al. [20] propose a framework and ontology for dynamic Web Service selection based on software agents coupled with a QoS ontology. With this approach, participants can collaborate to determine each other's service quality and trustworthiness. Keidl et al. [22] propose the serviceGlobe environment that implements dynamic service selection using UDDI's notion of a *tModel*.

These approaches lack in providing self-adaptation techniques that may significantly improve service-oriented systems, because such techniques can help tackle the increased complexity of the systems themselves and of their environment. In fact, the variability of the number of providers available to provide the services corresponding to the required functionalities, and the dynamic nature of the values of QoS parameters they can offer, makes it necessary to rely on approaches that allow

[1] In fact the `splitrule` checks that both branches have at least one execution path, and this condition may inhibit workflow coverage completion.

to find sub-optimal solutions based on some heuristics in a reasonable time and to compute new solutions every time conditions in the system changes.

The rationale for self-adaptive service selection and composition is summarized in literature as: the evolving behavior of a service (mobility, quality, faults, etc.), uninformed evolution of external services, inadequacy of pre-deployment information [11], extreme dynamicity, unreliability, and large scale [10], and a highly complex task, already beyond the human capability to deal with [9].

Also, it has been argued and generally accepted, that such self-adaptable, evolvable and context-aware systems require innovative approaches that take inspiration from nature, by considering devices, data, and services interacting as individuals of an ecosystem [2] and they can effectively organize large numbers of unreliable and dynamically changing components (cells, molecules, individuals, etc.) into robust and adaptive structures [10].

Viroli et al. [2] constructed a conceptual architecture for clarifying the concepts expressed and framing the several possible nature-inspired metaphors that could be adopted. They follow a biochemical approach where above a common environmental substrate (defining the basic ”laws of nature”), individuals of different kinds interact, compete, and combine with each other, so as to serve their own individual needs as well as the sustainability and the evolvability of the overall service ecosystem.

Ding et al. [9], Sun et al. [4] take the neuroendocrine-immune (NEI) system as a metaphor to create a decentralized, evolutionary, scalable, and adaptive system for Web service composition and management. Bio-entities represent Web services and they are able to obtain the desirable characteristics by self-organizing, cooperating, and compositing.

Ardagna et al. [3] aimed at fulfilling different preferences and constraints so as selection dynamically identifies the best set of services available at runtime. A new modeling approach involves service selection problem formalized as a mixed integer linear programming problem, loops peeling is adopted in the optimization, and constraints posed by stateful Web services are considered. Similarly to our approach, the notion of speculative/predictive execution has been investigated by applying probability of execution of conditional branches [3] and service rankings using social network analysis [11].

## 8 Conclusions

The present contribution proposes to decouple the workflow instantiation from its execution, so that the first one can be modelled as an independent, autonomous, and always running system. In such a way it is possible to take into account environmental changes, i.e. new provider availability, or changes in the provided QoS, as they occur without discharging IWs already produced. In fact, in very dynamic environments like the service oriented ones, it is not known a priori if new offers may lead to better workflows in terms of their QoS.

This is easily realized with a chemical approach since it allows to change at runtime the state of the system (in this case the number and/or the attribute of offers), so allowing new compositions to be found because new chemical reactions may take place in a way that simulate an *adaptation* of the system to different configurations not planned in advance.

Furthermore, the proposed approach allows also to dynamically change the selection criteria coming from user requirements because they are represented in the chemical reactions that are manipulated in the same way as molecules; so reactive molecules can be removed from the system, and new ones can be inserted in it so changing the system behaviour.

In such a way the chemical-based mechanism provides adaptability from both the provider side, by giving the possibility to insert new offers and so to re-activate chemical reactions, and from the user side, by giving the possibility to change his/her preferences during the instantiation phase.

This approach has also the advantage to make it possible to generate an IW also when there are missing service offers for some AW nodes. In fact, once the IW enactment takes place, and a path with missing service instances has to be executed, the execution may be suspended to query the chemical system for the missing parts if available by instantiating the placeholders in the IW. This is because the chemical system may run concurrently with the enactment engine, so it is possible to exploit new offers made available in the system by new providers, if it is the case, allowing for an adaptive instantiation of the workflow that could not be possible before the enactment starts.

**Acknowledgements.** The present work is supported by the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-CUBE). The authors would like to thank Zsolt Nemeth from MTA-SZTAKI University for his contribution.

## References

1. Allen, F.E.: Control Flow Analysis. SIGPLAN Not. 5(7), 1–19; ISSN:0362-1340
2. Viroli, M., Zambonelli, F.: A biochemical approach to adaptive service ecosystems. Information Sciences 180(10), 1876–1892 (2010)
3. Ardagna, D., Pernici, B.: Adaptive Service Composition in Flexible Processes. IEEE Transactions on Software Engineering 33(6) (2007)
4. Sun, H., Ding, Y.: A scalable method of e-service workflow emergence based on the bio-network. In: Fourth International Conference on Natural Computation, vol. 5, pp. 165–169. IEEE Computer Society (2008)
5. Pandey, S., Wu, L., Guru, S.M., Buyya, R.: A Particle Swarm Optimization-Based Heuristic for Scheduling Workflow Applications in Cloud Computing Environments. In: Proc. of the 2010 24th IEEE International Conference on Advanced Information Networking and Applications, pp. 400–407. IEEE Computer Society (2010)
6. Olariu, S., Zomaya, A.Y. (eds.): Handbook of Bioinspired Algorithms and Applications. CRC Press (2005)

7. Lee, C., Suzuki, J.: An immunologically-inspired autonomic framework for self-organizing and evolvable network applications. ACM Transactions on Autonomous and Adaptive Systems 4(4), 22:1–22:34 (2009)
8. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
9. Ding, Y., Sun, H., Hao, K.: A bio-inspired emergent system for intelligent web service composition and management. Journal Knowledge-Based Systems 20(5), 457–465 (2007)
10. Babaoglu, O., Canright, G., Deutsch, A., Di Caro, G.A., Ducatelle, F., Gambardella, L.M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., Urnes, T.: Design Patterns from Biology for Distributed Computing. ACM Transactions on Autonomous and Adaptive Systems 1(1), 26–66 (2006)
11. Mei, L., Chan, W.K., Tse, T.H.: An Adaptive Service Selection Approach to Service Composition. In: Proceedings of the IEEE International Conference on Web Services (ICWS 2008). IEEE Computer Society Press (2008)
12. Banâtre, J.-P., Fradet, P., Radenac, Y.: Principles of Chemical Programming. In: Proceedings of the Fifth International Workshop on Rule-Based Programming, RULE 2004. Electronic Notes in Theoretical Computer Science (2004)
13. Banâtre, J.-P., Radenac, Y., Fradet, P.: Chemical Specification of Autonomic Systems. In: Proceedings of the 13th Int. Conf. on Intelligent and Adaptive Systems and Software Engineering, IASSE 2004 (2004)
14. Banâtre, J.-P., Priol, T.: Chemical programming of future service-oriented architectures. Journal of Software 4(7), 738–746 (2009)
15. Banâtre, J.-P., Fradet, P., Radenac, Y.: Generalised multisets for chemical programming. Mathematical Structures in Computer Science 16, 557–580 (2006)
16. Banâtre, J.-P., Priol, T., Radenac, Y.: Service Orchestration Using the Chemical Metaphor. In: Brinkschulte, U., Givargis, T., Russo, S. (eds.) SEUS 2008. LNCS, vol. 5287, pp. 79–89. Springer, Heidelberg (2008)
17. Champrasert, P., Suzuki, J.: Symbioticsphere: A biologically-inspired autonomic architecture for self-managing network systems. In: Computer Software and Applications Conference, vol. 2, pp. 350–352. IEEE Computer Society (2006)
18. Dong, J., Sun, Y., Yang, S.: OWL-S - Ontology Framework Extension for Dynamic Web Service Composition. In: Proc. of the Eighteenth International Conference on Software Engineering & Knowledge Engineering, pp. 544–549 (2006)
19. Mukhija, A., Dingwall-Smith, A., Rosenblum, D.S.: QoS-Aware Service Composition in Dino. In: Proc. of the Fifth European Conference on Web Services, pp. 3–12. IEEE Computer Society (2007)
20. Maximilien, E.M., Singh, M.P.: A Framework and Ontology for Dynamic Web Services Selection. IEEE Internet Computing 8(5), 84–93 (2004)
21. Ponnekanti, S.R., Fox, A.: SWORD: A Developer Toolkit for Web Service Composition. In: Proc. of the 11th World Wide Web Conference (2002)
22. Keidl, M., Seltzsam, S., Stocker, K., Kemper, A.: ServiceGlobe: Distributing E-services Across the Internet. In: VLDB 2002: Proceedings of the 28th International Conference on Very Large Data Bases, pp. 1047-1050 (2002)
23. Di Napoli, C., Giordano, M., Németh, Z., Tonellotto, N.: Using Chemical Reactions to Model Service Composition. In: Proc. of the Second International Workshop on Self-Organizing Architectures, pp. 43–50 (2010)
24. Bantre, J., Mtayer, D.L.: The GAMMA Model and Its Discipline of Programming. Sci. Comput. Program., 55–77 (1990)

# Author Index

Avilés, Carlos 171

Banzhaf, W. 51
Biscani, Francesco 151
Bliss, Nadya 219

Cámara, Mario 101
Codecasa, Daniele 125
Contreras, Iván 189

de Toro, Francisco 101
de Vega, Francisco Fernández 1, 77
Di Napoli, Claudia 267

Giordano, Maurizio 267
González, Daniel Lombraña 77
Guervós, Juan Julián Merelo 77
Guillén, Alberto 243

Harding, Simon 51
Herrera, Luis Javier 243
Hidalgo, J. Ignacio 1, 189

Izzo, Dario 151

Jiang, Yiyi 189

Lanchares, Juan 1
Langdon, W.B. 7
Laredo, Juan Luis Jiménez 77
Lendasse, Amaury 243

Mauri, Giancarlo 125
Mohindra, Sanjeev 219
Mullen, Julie 219

Nuñez-Letamendía, Laura 189

Olague, Gustavo 171
O'Reilly, Una-May 219
Ortega, Julio 101

Pomares, Héctor 243

Robinson, Eric 219
Rojas, Ignacio 243
Ruciński, Marek 151

Sossa, Humberto 171
Sovilj, Dušan 243

van Heeswijk, Mark 243
Vanneschi, Leonardo 125
Villegas-Cortez, Juan 171