Chapter 7

Kelvin K. Yue
Department of Computer Science

David J. Lilja
Department of Electrical Engineering
University of Minnesota
200 Union Street S.E.
Minneapolis, MN 55455

yue@cs.umn.edu
lilja@ee.umn.edu

Parameter Estimation for a Generalized Parallel Loop Scheduling Algorithm

Abstract

Algorithms that dynamically schedule parallel loop iterations in a shared-memory multiprocessor have been proposed to balance the processors' workload while maintaining low scheduling overhead. However, none of the existing strategies perform well for all types of loops on all types of system architectures. We present a generalized loop scheduling algorithm that can be adjusted to match the loop characteristics to the system environment. A new method of simulation using the Genetic Algorithm is developed to determine appropriate scheduling parameters. This approach allows us to quickly choose sets of scheduling parameters for different loops executing on different systems. Stochastic simulations show that our parameterized strategies perform at least as well as the best existing algorithms for different combinations of loop iteration characteristics and system assumptions. Our generalized strategy is thus more robust than existing strategies.

7.1 Introduction

Since the body of a loop may be executed multiple times, exploiting loop-level parallelism is an effective means of increasing performance in a shared-memory multiprocessor system [9]. Parallel loop scheduling algorithms, such as chunk scheduling [8], self-scheduling [3], guided self-scheduling [12], factoring [7], and trapezoid self-scheduling [13], have been proposed to evenly distribute the workload among the processors while maintaining low scheduling overhead. However, the performance of these scheduling algorithms is sensitive to the loop characteristics and the system architecture so that no single algorithm performs well for all types of loops on all types of system architectures [14].

In this chapter, we propose a generalization of the current parallel loop scheduling algorithms in which the scheduling characteristics are parameterized. By using this generalized algorithm, we can quickly adjust the scheduling strategy to match the loop characteristics to the system environment. As the combinations of scheduling strategies, loop characteristics, and system environments are enormous, a new simulation method involving the Genetic Algorithm is developed to estimate the scheduling parameters needed to achieve good performance.

The use of the Genetic Algorithm for multiprocessor scheduling has been previously proposed [6, 11], but these methods depend on knowing *a priori* precise task information, such as the order of the tasks' execution, the task arrival times, the *exact* execution times, and the dependences between tasks. These methods then generate a *schedule* specific to this set of tasks. These methods are not feasible for loop-level parallelism since the time needed for finding a schedule may be longer than the loop execution time and, in many cases, the loop characteristics are unknown until run-time. Instead of finding a specific schedule, our proposed method uses the Genetic Algorithm to find appropriate values for the parameters of the generalized scheduling algorithm to produce a specific scheduling strategy or algorithm. This algorithm, then, is used at run-time to dynamically generate the actual schedule for executing the loop iterations.

Two new scheduling strategies are found using this method, one of which is suitable for scheduling loops with small iteration execution time variances, while the other is suitable for loops with large variances. They perform as well as, or better than, existing algorithms. Since the scheduling parameters of our algorithms can be adjusted based on the changes in the loop characteristics or system environments, our generalized method is more robust.

This chapter is organized as follows: Section 7.2 provides background information on existing parallel loop scheduling strategies. Section 7.3 presents our methodology for finding scheduling parameters using the Genetic Algorithm, while Section 7.4 discusses the simulated results of applying this strategy to loop-level parallelism. Section 7.5 concludes the chapter.

7.2 Current Scheduling Algorithms

In this section, the current techniques for scheduling Doall loop iterations on a shared-memory multiprocessor system, such as that shown in Figure 7.1, are reviewed. A performance comparison of these algorithms is also presented.

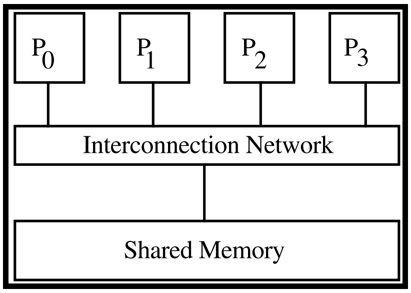

Figure 7.1: Shared memory multiprocessor architecture.

A Doall loop is the simplest form of parallelizable loop. In this type of loop each iteration is independent of the other iterations so that the iterations can be executed concurrently as independent tasks. An example of a Doall loop is:

```
DO i=I,N
     a(i) = b(i) + c(i)
END DO
```

The iterations of a Doall loop are assigned to the processors to execute based on some loop scheduling strategy. There are two main categories of scheduling algorithms: static and dynamic [9]. *Static scheduling,* or *prescheduling,* assigns iterations to the processors at compile time. Each processor *knows* exactly which iterations it should execute before the program is invoked and, therefore, there is no scheduling overhead. For example, the compiler could assign iterations to the processors based on the processor number so that processor 0 executes iterations 1, $P + 1$, $2P + 1$,..., processor 1 executes iterations 2, $P + 2$, $2P + 2$,..., and so on, where P is the number of processors. The main disadvantage of static scheduling is *load imbalance* [2]. This unequal distribution of work to the processors can be caused by differences in the iteration execution times, or by differences in the number of iterations each processor executes. Since the *schedule* of iteration execution is fixed at compile-time, it cannot be adjusted based on the dynamically varying workload of the processors.

Dynamic scheduling assigns iterations to processors at run-time and can therefore adjust the schedule to the processors' workload. *Self-scheduling* is the simplest form of dynamic scheduling. With self-scheduling, each idle processor obtains the index of the next iteration it should execute by accessing a shared work queue. By taking one iteration at a time, this algorithm balances the workload very well, but the scheduling overhead is large since the shared work queue must be accessed once for each iteration.

To reduce the scheduling overhead, *chunk scheduling* assigns groups of iterations as a single unit to the processors. Kruskal and Weiss [8] analyzed load imbalances with this strategy and proposed the optimal chunk size to be $\left[\left(\sqrt{2}Nh\right)/\left(\sigma P\sqrt{\log P}\right)\right]^{2/3}$, where N is the number of iterations, P is the number of processors, σ is the standard deviation of the distribution of iteration execution times, and h is the scheduling overhead. They assume that the central-limit theorem holds for the iteration execution times, which is valid only when N is large.

Another approach to reduce load imbalance while maintaining low scheduling overhead is to decrease the chunk size as the program executes. There are two strategies for decreasing the chunk size: *linear* decreases and *nonlinear* decreases. *Guided self-scheduling* (GSS) [12] decreases the chunk size nonlinearly by allocating iterations with a chunk size equal to $\lceil R/P \rceil$, where R is the number of iterations remaining to be executed. This algorithm allocates large chunk sizes at the beginning of a loop's execution to reduce the scheduling overhead. As the number of iterations remaining to be executed decreases, smaller chunks are allocated to balance the load.

The *factoring* scheduling algorithm (FS) [7] is similar to GSS except that it allocates iterations in batches of P equal-sized chunks. After a batch is scheduled, the new chunk size is calculated to be $\lceil R/(\mathrm{x}P) \rceil$, where R is the number of iterations remaining, and x typically is chosen to be 2. The initial chunk size for FS is smaller than GSS. As a result, it has more iterations remaining at the end of the loop's execution to balance the load. However, FS requires many more scheduling steps than GSS. To reduce the number of scheduling steps, *safe self-scheduling* [10] proposes to use an x factor smaller than 2 so that more iterations will be allocated per chunk. However, the calculation of the x factor for safe self-scheduling requires knowing not only the maximum and minimum iteration execution times, but also the probability of branching for the conditional statements in the loop. Safe self-scheduling may be less robust than factoring or guided self-scheduling since these characteristics typically are not known until runtime.

Trapezoid self-scheduling (TSS) [13] decreases the chunk size linearly to achieve a better tradeoff between the scheduling overhead and the distribution of the processors' workload compared to the nonlinear strategies. The number of chunks, C, is equal to $[2N/(f + l)]$ and the chunk size is decreased by a factor of $(f- l)/(C - 1)$ at each scheduling step, where typically $f = N/(2P)$ and $l = 1$. TSS does not allocate chunks as large as GSS in the beginning, and it does not require as many scheduling steps as FS. However, the linearly decrementing chunk size may create large load imbalances if the execution time differences between the last few chunks are large.

To summarize, one-iteration-at-a-time self-scheduling can perfectly balance the workload but it generates a large scheduling overhead that adds directly to the overall execution time. Chunk scheduling, on the other hand, requires minimum overhead, but it produces greater load imbalance. Guided self-scheduling, factoring, and trapezoid self-scheduling use a variable chunk size to tradeoff load imbalances with the scheduling overhead. However, the performance of these algorithms is sensitive to the characteristics of the loop and the system environment so that no single algorithm performs best in all cases [14, 15]. For instance, if the variance in iteration execution times is large, GSS may not balance the workload well since it does not save enough single-iteration chunks until the end [7, 13]. Factoring saves enough single-iteration chunks to balance the load, but with small variances in iteration execution times, these chunks cause extra scheduling overhead [9]. Trapezoid self-scheduling assigns small initial chunks, as does factoring, and it requires fewer scheduling steps than GSS [15], but the difference in execution time between the last few chunks might be large due to the linear decrement in the chunk size. This large difference may create correspondingly large load imbalances [7].

7.3 A New Scheduling Methodology

In the previous section, we reviewed five dynamic scheduling algorithms and concluded that no single algorithm produces the best performance in all cases. To match the scheduling algorithms to the loop characteristics and system environments, one can exhaustively try all of the strategies for all types of loops on all types of systems. However, this is obviously infeasible, if not impossible.

We propose a generalization of all of these scheduling algorithms in which the scheduling characteristics are parameterized and, therefore, can be easily adjusted to match the scheduling algorithm to both the individual loop and the system architecture. We also develop a new simulation methodology that uses the Genetic Algorithm as a heuristic search engine to choose appropriate parameters for the generalized scheduling strategy.

This section details the generalization of the scheduling algorithms and presents the simulation methodology. The implementation of the Genetic Algorithm is also described.

7.3.1 A Generalized Loop Scheduling Algorithm

As discussed in Section 7.2, there are two primary types of dynamic scheduling algorithms: those that use a fixed chunk size based on the total number of iterations, and those that use a variable chunk size based on the remaining number of iterations. The first step of the generalization is to define a pargreeter X which is equal to N, the total number of iterations, if the scheduling strategy uses a fixed chunk size. Otherwise, X is equal to R, the remaining number of iterations, if the strategy uses a variable chunk size.

Notice that the chunk size for the current scheduling algorithms is related to the total number of processors, P. For instance, the chunk size for chunk scheduling is $\lceil N/P \rceil$, for GSS it is $\lceil R/P \rceil$, for FS it is $\lceil R/2P \rceil$, and the initial chunk size for TSS is $\lceil N/2P \rceil$. Therefore, the chunk size for our generalization is in terms of aX/fP. The parameter f is used to represent the factoring size giving $f = 1$ for CS and GSS, and $f = 2$ for FS. We also introduce another adjustment factor, a, to make our generalization more versatile by not limiting the scheduling algorithm to only integer factors.

To include all of the possible chunk sizes while allowing the chunk size to be decremented either linearly, as in TSS, or nonlinearly, as in GSS and FS, the parameter ℓ is introduced and the generalization is refined to $aX/fP - \ell$ For fixed-sized scheduling algorithms, or for variable-sized scheduling algorithms with a nonlinearly decreasing chunk size, ℓ is used as a *refining* factor. For instance, if chunk scheduling is used where the chunk size is determined to be some integer value that cannot be calculated with only aX/fP, then ℓ is set to a constant value to adjust the chunk size to the desired value. On the other hand, if a linearly decrementing chunk size strategy is used, ℓ is a function of the scheduling step. In TSS, for example, $\ell = i\times\left(\frac{N}{2P}-1\right)\Big/\left\lceil\frac{2N}{2P+1}-1\right\rceil$ where i is the current scheduling step. As the execution proceeds, the number of the scheduling step is increased, which causes the chunk size to decrease linearly.

In our generalization, we also include a parameter, m, for a minimum chunk size feature as suggested in [12]. If the calculated chunk size is smaller than m, a chunk size of m is used instead. Also, the parameter C is the number of chunks with the same size that are scheduled before the chunk size is recalculated. In FS, C is equal to P, while for the other scheduling algorithms, C is always 1. Note that C can take on any value in our generalization.

The generalization of loop scheduling algorithms is summarized as follows:

The number of iterations per chunk, K, is determined by:

$$\begin{cases} K = \left\lceil \frac{a}{f}\frac{X}{P} - \ell \right\rceil & \text{if } K > m, \\ K = m & \text{otherwise}, \end{cases}$$

and C batches of the same chunk size, K, are scheduled before K is recalculated.

In the above expression, a and f are the adjusting factors, X is equal to N, the total number of iterations, if the strategy uses a fixed chunk size, or X is equal to R, the number of iterations remaining to be executed, if the strategy uses a variable chunk size, P is the number of processors, ℓ is the linear decrement factor, and m is the minimum chunk size allowed. The following table shows the parameter values that will produce specific scheduling algorithms:

Algorithm	C	a	f	X	ℓ	m
Self Scheduling	1	P	N	N	0	1
Chunk Scheduling	1	#	a	N	0	1
Guided Self-Scheduling	1	#	a	R	0	1
Factoring	P	#	$2a$	R	0	1
Trapezoid Self-Scheduling	1	#	a	N	δ	1

The symbol # represents any positive integer and $\delta = i\left(\frac{N}{2P}-1\right)\Big/\left\lceil\frac{2N}{2P+1}-1\right\rceil$ where i is the current scheduling step. For CS, GSS, FS, and TSS, we can choose any positive integer for parameter a by properly choosing the corresponding parameter f. In addition to the values shown in the table, self-scheduling can also be represented with the parameters $a = f = 1$, $X = N$, $\ell = (N/P) - 1$, $C = 1$, $m = 1$.

Based on this generalization, we have shown that we can parameterize the different existing loop scheduling strategies. This generalization allows us to select the desired scheduling algorithm, and it allows us to produce completely new scheduling algorithms, by choosing the appropriate parameters. As a result, we can easily adjust the scheduling algorithm to match the loop characteristics to the system environment.

7.3.2 Parameter Estimation

To utilize the generalized scheduling algorithm, a quick and simple method for matching the scheduling parameters to the loop characteristics and the system environment is needed. As previously mentioned, it is infeasible, if not impossible, to exhaustively test all the parameter combinations to determine which one generates the best performance. Therefore, we develop a new simulation methodology that uses the Genetic Algorithm (GA) as the means to determine appropriate parameters.

Our simulation consists of two modules: the GA engine and the multiprocessor simulator (Figure 7.2). The GA engine generates possible scheduling strategies and then sends them to the multiprocessor simulator for evaluation. The multiprocessor simulator simulates a shared memory multiprocessor environment executing a Doall loop based on the given scheduling strategies. It returns a measure of the performance of each strategy to the GA engine, which then creates new strategies based on the simulated performance of the previous strategies. In the following subsections, the implementations of the GA engine and the multiprocessor simulator are presented in detail.

7.3.2.1 GA Engine

The Genetic Algorithm (GA) [5] has been applied to a wide variety of areas, ranging from artificially intelligent machine learning to gas pipeline control systems, since it was first introduced

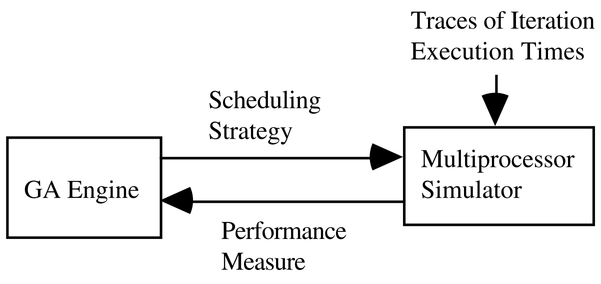

Figure 7.2: Simulation environment for estimating the parameters of the generalized scheduling algorithm.

approximately twenty years ago. It has been proven to be a robust and efficient algorithm for searching and optimization problems [4]. GA is based on the concept of natural selection and adaptation and the idea of *survival of the fittest.* GA is different from other optimization and search algorithms in four important characteristics [4]:

1. GA works with a coding of the parameter set, not the parameters themselves.

2. GA searches from a population of points, not just a single point.

3. GA uses payoff (objective function) information, not derivatives or other auxiliary knowledge.

4. GA uses probabilistic transition rules, not deterministic rules.

These characteristics of the Genetic Algorithm combine both exploration and exploitation in the searching process [1]. Unlike the hillclimbing search, which is simply exploitation, GA explores new domains in the search space and will not be limited to local maxima. Unlike random search, GA uses the known results to guide it to a better solution, thereby making the search process more efficient. Moreover, GA is more feasible than a brute-force trial-and-error method since it does not try every possible parameter combination in the search space. Therefore, we think that the Genetic Algorithm can be used in our simulation to find estimates of the scheduling parameters based on the system environment.

In the following subsections, the genetic operations of GA are reviewed and the representation and implementation of our generalized scheduling algorithm in the Genetic Algorithm framework is presented.

Implementation The parameters for the scheduling strategy are represented in the chromosome format shown in Figure 7.3. The parameters ℓ, f, a, C, and m are described in Section 7.3.1. Their binary representations are decoded into integer values in the simulator. The bits X and ℓ are condition bits. If bit X is one, N, the total number of iterations, is used. Otherwise, R, the remaining number of iterations, is used. If bit ℓ is one, the linear decrement strategy is used. GA does not impose any specific rules in designing or coding of a chromosome, and the quality of the resultant solutions does not depend on the arrangement of the parameters within the chromosome due to the robustness of the Genetic Algorithm [4].

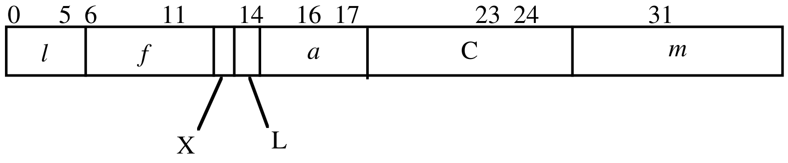

Figure 7.3: Chromosome representation for the generalized loop scheduling strategy.

The fitness function for a chromosome is the parallel execution time efficiency of a Doall loop executed using the scheduling strategy encoded in the chromosome. It is calculated as:

$$E = \frac{Speedup}{P} = \frac{Sequential\ Run - time}{P \times Parallel\ Run - time}$$

The sequential run-time is the sum of all iteration execution times, excluding the scheduling overhead, which is equivalent to the execution time for the Doall loop when it is executed on a sequential machine. The parallel run-time is the total execution time of the last processor to finish executing.

7.3.2.2 Multiprocessor Simulator

After a set of scheduling parameters is generated by the GA engine, its performance then needs to be evaluated. In this experiment, we use a simple stochastic simulation model. It is possible, however, to use a more complicated simulation, or even a real multiprocessor system, for the performance evaluation.

This module simulates a shared memory multiprocessor system with *P* processors, all of which execute at the same speed. When a processor is idle, it locks the loop index variable to determine the next chunk of iterations it will execute. It then unlocks the loop index variable and begins executing the iterations. The number of iterations a processor assigns itself at each scheduling step, i.e., the size of a chunk, is determined by the scheduling strategy. When two or more processors attempt to simultaneously access the loop index, the one with the smallest processor identification number is allowed to go first. The delay introduced by this contention adds directly to the execution time of the stalled processors. The specific scheduling strategy used in the simulator is dynamically configured according to the information sent from the GA Engine.

The execution times of the iterations are generated by a random number generator with a normal (Gaussian) distribution. The mean and variance of the iteration execution times are specified based on the types of the loops [14]. Again, it is possible to use more complicated methods to generate traces of the iteration execution times, but we use the simplest method to demonstrate our scheduling strategy. At the end of the simulation, the efficiency of the scheduling strategy is calculated and returned to the GA engine where it is used as the fitness value of the chromosome that defines the given scheduling strategy. The following algorithm summarizes the simulation environment:

```
/* Initialization */
randomly generate the initial population


FOR each chromosome in the population
   /* begin multiprocessor simulation */
    simulate the scheduling strategy
    measure the efficiency
   /* end multiprocessor simulation */
   use the efficiency value as the fitness of the chromosome
```

```
ENDFOR

DO until (population converges) or (no. of generations >
predefined value)
   /* Selection Phase */
   select the chromosomes with the highest fitness values

   /* Reproduction Phase */
   generate the new chromosomes using crossover operator
   and mutation operator.

   /* Evaluation Phase */
   FOR each chromosome in the new population
      /* begin multiprocessor simulation */
       simulate the scheduling strategy
       measure the efficiency
      /* end multiprocessor simulation */
      use the efficiency value as the fitness of the
      chromosome
   ENDFOR
ENDDO
```

7.4 Results

We use the simulation methodology discussed in the previous section to match scheduling algorithms to the loop characteristics while varying the number of processors, the number of iterations, the scheduling overhead, and the variance in iteration execution times. The GA engine found that scheduling algorithms that use a fixed chunk size, and algorithms that use a variable chunk size with linear decrement, do not perform as well as the other algorithms. The GA engine discovered two new scheduling algorithms that perform as well as, or better than, existing scheduling algorithms. We call these two new algorithms *CS-2* and *FS-alt* due to their similarity to chunk scheduling and factoring, respectively. CS-2 is similar to chunk scheduling in that it uses N/P iterations per chunk, except it saves $2P$ single-iteration chunks to balance the load at the end. FS-alt is similar to factoring except that it uses a larger *factor* and it allocates fewer chunks per batch. In this section, these two new scheduling algorithms are presented and their performance is compared with the other scheduling algorithms.

7.4.1 New Scheduling Algorithms

CS-2 allocates iterations with two different chunk sizes it has $2P$ chunks with 1 iteration per chunk and P chunks with $\lceil N/P - 2 \rceil$ iterations per chunk. When the loop execution begins, each processor acquires a chunk with $\lceil N/P - 2 \rceil$ iterations and starts executing. Near the end of the execution, the single-iteration chunks are used to dynamically balance the workload among the processors. The total number of scheduling steps, that is, the number of accesses to the shared work queue, for CS-2 is $3P$. CS-2 is suitable for loops with small variances in iteration execution times. Standard chunk scheduling cannot balance the variation well for this type of loop while other dynamic scheduling algorithms require too many scheduling steps and, thus, are too costly for effectively balancing this small variation. For loops with large variances in execution times, CS-2, similar to standard chunk scheduling, does not perform well compared to other dynamic scheduling algorithms.

FS-alt, on the other hand, performs well for loops with large variances. It is similar to factoring, except that it uses a factor of 5/6 instead of 1/2, and it uses *P*/2 chunks per batch instead of *P* chunks per batch. FS-alt improves on the performance of factoring by allocating larger chunks at the beginning of the execution and, therefore, reducing the number of scheduling steps. To compensate for the larger chunk sizes, FS-alt allocates *P*/2 chunks per batch allowing it to save enough small chunks to balance the processors' workload at the end of the loop's execution. The following table shows the parameter values for CS-2 and FS-alt from our generalization of the loop scheduling algorithm.

Algorithm	C	a	f	X	ℓ	m
CS-2	P	#	a	R	2	1
FS-alt	$P/2$	5	6	R	0	1

7.4.2 Performance Comparisons

Figure 7.4 shows the speedup values of the two GA-generated scheduling strategies compared to the current algorithms. The speedup is measured on a P = 16 processor system executing a Doall loop with an average iteration execution time of 100 cycles. The overhead for scheduling a chunk of iterations is 10% of the mean iteration execution time, or 10 cycles. The total number of iterations, N, is set to 500 iterations and 5000 iterations while the variance in iteration execution times is changed from 5 cycles to 70 cycles.

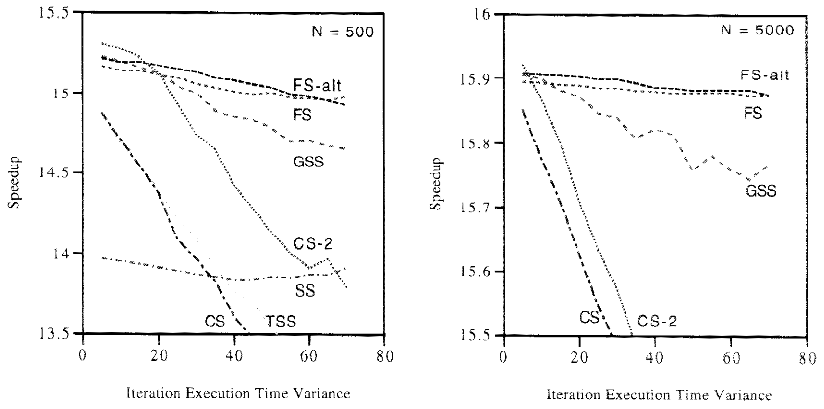

Figure 7.4: Speedup comparisons of the different scheduling algorithms.

As shown in Figure 7.4, CS-2 performs better than the other scheduling algorithms when the iteration execution time variances are small. As the variance increases, CS-2's performance decreases with the same rate as standard chunk scheduling. FS-alt performs slightly better than FS in both cases, but the difference between the two in the N = 5000 case is quite small. The speedup of SS and TSS are less than 15.5 when N = 5000 and are not shown in the figure. The large scheduling overhead and poor load balancing are the causes of the poor performance for SS and TSS, respectively.

The comparisons shown in Figure 7.4 are based on the assumption that all the scheduling algorithms have the same scheduling overhead. It suggests that the two GA-generated scheduling algorithms slightly improve the overall performance. The scheduling overhead for some algorithms is much lower than for the others, however. For instance, SS requires only a `Fetch&Add` operation to obtain the next iteration while FS requires a more complicated calculation. To eliminate this factor, Figure 7.5 compares the total number of scheduling steps for all of the algorithms with different values of N, the total number of iterations, on a 16-processor system. The total number of scheduling steps for SS is not shown in the figure since it is simply the total number of iterations. This figure shows that FS requires the most scheduling steps, and the number of scheduling steps increases at a faster rate than the others as the total number of iterations increases. FS-alt schedules iterations in a fashion similar to FS, but it requires fewer total scheduling steps. CS-2 requires at least twice as many scheduling steps as CS, but, as shown in Figure 7.4, it balances the workload more evenly than CS.

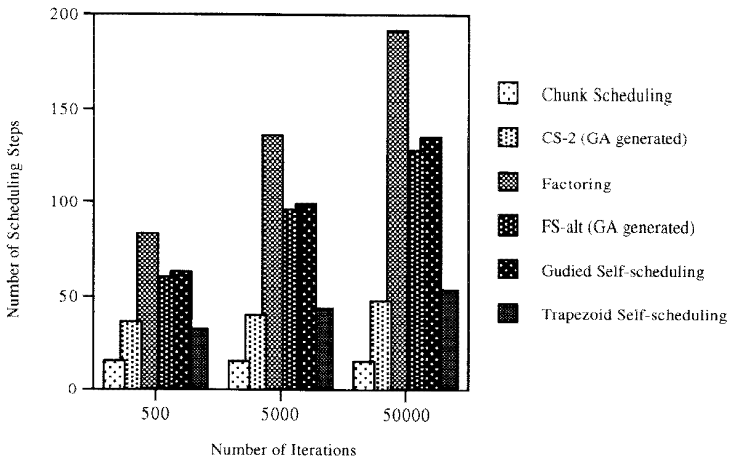

Figure 7.5: Comparison of the number of scheduling steps.

The total number of scheduling steps not only directly contributes to the scheduling overhead, but it also relates to the network and memory contention since, as the number of scheduling steps increases, the chance of two or more processors trying to access the shared loop index at the same time increases as well. When one processor is accessing the loop index, all the other processors which need to obtain additional work at the same time must wait. Figure 7.6 compares the processor execution times divided into three different categories: the *execution time,* which is the time the processor spends executing the iterations, the *overhead,* which is the time the processor spends calculating the chunk size and accessing the shared loop variables, and the *contention time,* which is the time the processor is idle waiting to access the shared variables or waiting for synchronization. The sum of these three times is equal to the parallel execution time of the Doall loop using the specific scheduling algorithm. We set the average iteration execution times to 100 cycles and vary the number of processors (*P*), the total number of iterations (*N*), the scheduling overhead (*O*), and the iteration execution time variance (*V*).

In Figure 7.6(a), we have a 500-iteration Doall loop with a variance of 10 cycles executing on a 16-processor system with a scheduling overhead of 10 cycles. SS produces the largest scheduling overhead while, as expected, CS has the smallest. The GA-generated algorithms, CS-2 and FS-alt, both have a small scheduling overhead compared to the others. CS-2, FS, FS-alt, and GSS have similar contentions and, therefore, the algorithms with lower scheduling overhead, i.e., CS-2 and FS-alt, have the lower total parallel runtime. We use Figure 7.6(a) as the comparison baseline as we alter the system parameters. In Figure 7.6(b), the total number of processors (*P*) is doubled. The average execution time for all of the algorithms is halved as more processors share the same amount of work. The scheduling overhead per processor is decreased since the processors do not need to obtain work from the shared work queue as many times as in the baseline case. The contention, or the processor idle time, is increased, however, since more processors are competing to access the shared work queue. An opposite effect occurs when the number of iterations (*N*) is doubled, as shown in Figure 7.6(c). In this case, the processors spend more time executing the iterations since the workload per processor is increased. The scheduling overhead is also increased since more iterations need to be assigned, but the contention time is decreased since there is more work for the processors and the processors are less likely to wait idle.

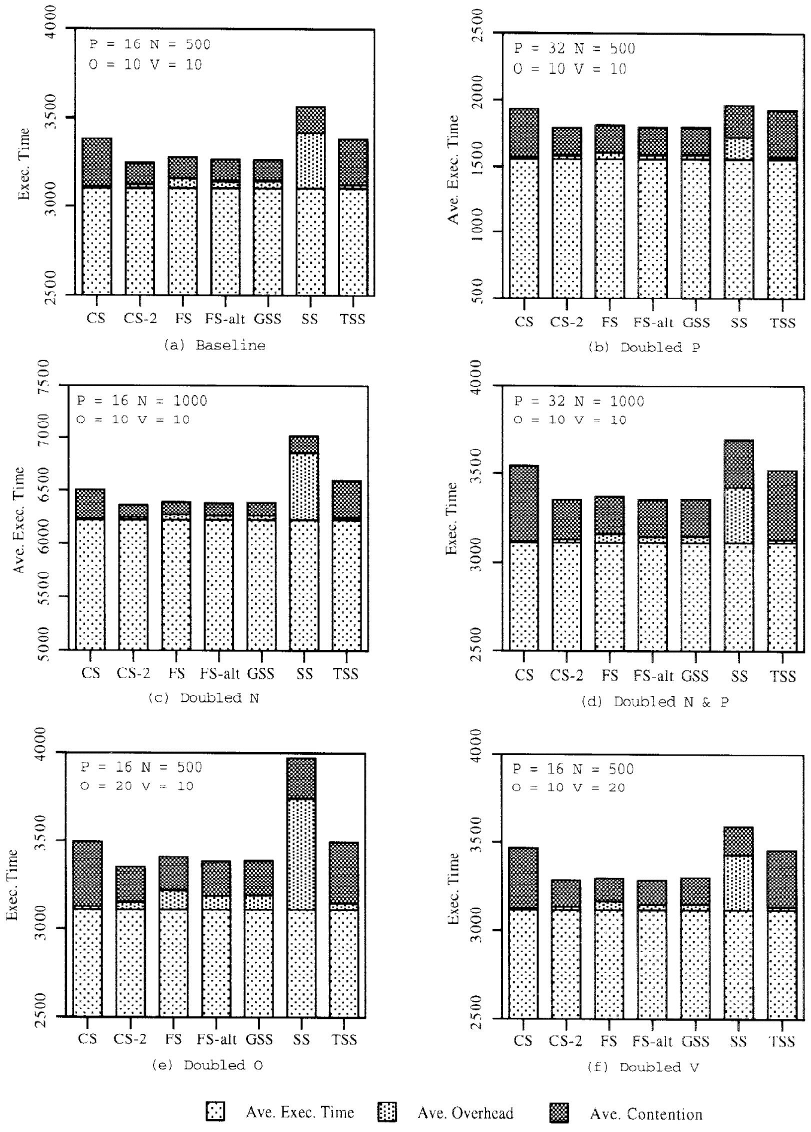

Figure 7.6: Breakdown of processor execution times.

To compare the scalability of the scheduling algorithms, both the number of processors (*P*) and the number of iterations (*N*) are doubled. The contention of all of the algorithms is increased from the baseline since more processors are sharing the single shared work queue. Self-scheduling (SS) suffers the most as it has to access the shared work queue for each iteration and there are more iterations to be executed, and more processors to compete for the work queue. CS-2, FS-alt, FS, and GSS also have increased contention time, but they still outperform the other algorithms. CS-2 and FS-alt have relatively little scheduling overhead comparing to FS and GSS. As a result, these two scheduling algorithms have the shortest overall parallel execution times.

From Figure 7.5, it is seen that FS requires more scheduling steps than any of the other algorithms except SS. The effect of this factor on the overall performance becomes more obvious when the overhead per scheduling step (*O*) is doubled, as shown in Figure 7.6(e). Both the scheduling overhead and the contention time for all of the scheduling algorithms are increased compared to the baseline in Figure 7.6(a), since it takes longer to access the shared work queue, and since the competing processors must wait idle longer. The performance of FS degrades more than FS-alt and CS-2 because of its much greater number of scheduling steps. CS-2 requires fewer scheduling steps than FS-alt, and, therefore, the overall parallel execution of CS-2 is less than that of FS-alt.

To compare the algorithms' sensitivity to variances in the iteration execution times, we measure the execution time after doubling the variance (*V*). Both CS and TSS produce longer parallel execution times due to the larger load imbalance induced by the larger variance. The performance of SS remains the same since it always balances the workload perfectly. The increases in contention times for CS-2 and GSS are larger than that of FS and FS-alt because they do not have enough single-iteration chunks to balance the workload at the end of the loop's execution. However, the overall execution times for CS-2 and FS-alt are still less than the others, as shown in Figure 7.6(f).

In this section, we have compared the performance of the GA-generated algorithms, CS-2 and FS-alt, with the current scheduling algorithms. The results show that CS-2 has smaller scheduling overhead than the other algorithms and that it outperforms the others when the iteration execution time variance is small. FS-alt, on the other hand, has a larger scheduling overhead than CS-2, but it produces better load balance when the iteration execution time variance is large.

7.5 Conclusion

In this chapter, a generalized scheduling algorithm is proposed in which the scheduling strategy is parameterized and so can be adjusted to match the loop characteristics and the system environment. A new simulation methodology using the Genetic Algorithm is developed to find appropriate parameters for this generalized scheduling. Two new scheduling algorithms, CS-2 and FS-alt, were discovered using this simulation methodology. CS-2 is similar to CS, but it improves the load balancing capability while maintaining a low scheduling overhead. It is suitable for loops with small iteration execution time variances. FS-alt, on the other hand, performs well for loops will large variances. It reduces the scheduling overhead of FS by using a larger factor and a smaller batch size.

Based on simulated performance comparisons, the newly discovered algorithms perform as well as, or better than, the existing algorithms. Since we can further *fine tune* the parameters of our generalized scheduling algorithm by interfacing the GA engine to a real multiprocessor system, or to a system of some other architectural design, our scheduling algorithm is more robust than current algorithms. Another possible use of the generalized scheduling algorithm is to have a dedicated processor executing the GA engine to adjust the scheduling parameters dynamically based on the status of the system and the loop execution. A variety of other techniques can be used to determine appropriate values for the parameters of this generalized loop scheduling algorithm.

References

[1] David Beasley, David R. Bull, and Ralph R. Martin. *An Overview of Genetic Algorithms: Part 1, Fundamentals,* volume 15 of *University Computing,* pages 58-69. Inter-University Committee on Computing, University of Cardiff, Cardiff, *CF2* 4YN, UK, 1993.

[2] Carl J. Beckmann and Constantine D. Polychronopoulos. The effect of scheduling and synchronization overhead on parallel loop performance. In *International Conference on Parallel Processing,* volume II: Software, pages 200-204, 1989.

[3] Zhixi Fang, Pen-Chung Yew, Peiyi Tang, and Chuan-Qi Zhu. Dynamic processor selfscheduling for general parallel nested loops. In *Proc. 1987 International Conference in Parallel Processing,* August 1987.

[4] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, Massachusetts, 1989.

[5] John Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, Michigan, 1975.

[6] Edwin S.W. Hou, Nirwan Ansari, and Hong Ren. A Genetic Algorithm for Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems,* 5:113-120, February 1994.

[7] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring - a method for scheduling parallel loops. *Communciations of the ACM,* 35:90 101, August 1992.

[8] Clyde P. Kruskal and Alan Weiss. Allocating independent subtasks on parallel processors (extended abstract). In *International Conference on Parallel Processing,* pages 236 240, 1984.

[9] David J. Lilja. Exploiting the parallelism available in loops. *Computer,* pages 13-26, February 1994.

[10] Jie Liu, Vikram A. Saletore, and Ted G. Lewis. Scheduling parallel loops with variable length iteration execution times on parallel computers. In *ISMM 5th International Conference on Parallel and Distributed Computing Systems,* pages 83-89, October 1992.

[11] Hirak Mitra and Parameswaran Ramanathan. A genetic approach for scheduling nonpreemptive tasks with precedence and deadline constraints. In *26th Hawaii International Conference on System Sciences,* volume 2, pages 556- 564, 1993.

[12] C. Polychronopoulos and D. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers,* C-36:1425-1439, December 1987.

[13] Ten H. Tzen and Lionel M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems,* 4:87-97, January 1993.

[14] Kelvin K. Yue and David J. Lilja. Categorizing parallel loops based on iteration execution time variances (submitted for publication). 1994.

[15] Kelvin K. Yue and David J. Lilja. Scalability analysis for parallel loop scheduling algorithms (submitted for publication). 1994.