Table of Contents

Introduction

This is an investigation into the application of concurrent programming and object technology in developing distributed systems with specific interest in concurrent scheduling and planning algorithms. The research paper begins with a brief presentation of the concepts of object technology and C++. Then, as a preface to the presentation of the concepts of concurrent programming and object frameworks, the investigation of the Common Object Request Broker Architecture (CORBA), and the prototyping results, the next section gives some background information to set the tone for this research and to support the following statement.

Distributed object technology and concurrent/parallel processing are part of an abrupt evolution in information systems. Object technology will be the vehicle that allows information system providers to reap the benefits of distributed processing. Distributed object technology will affect the IS industry at all levels of hardware, operating system and software design.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Introduction

This is an investigation into the application of concurrent programming and object technology in developing distributed systems with specific interest in concurrent scheduling and planning algorithms. The research paper begins with a brief presentation of the concepts of object technology and C++. Then, as a preface to the presentation of the concepts of concurrent programming and object frameworks, the investigation of the Common Object Request Broker Architecture (CORBA), and the prototyping results, the next section gives some background information to set the tone for this research and to support the following statement.

Distributed object technology and concurrent/parallel processing are part of an abrupt evolution in information systems. Object technology will be the vehicle that allows information system providers to reap the benefits of distributed processing. Distributed object technology will affect the IS industry at all levels of hardware, operating system and software design.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Object Technology

The fundamental characteristic of object oriented technology is the combination of data structure and behavior.1 This combination of data and function is more understandable to people and most resembles real life. As operating systems, programming languages, and frameworks evolve through object technology to allow a more efficient coupling between processing and networking, distributed processing will be as common place as simple data structures like linked lists.

Many of the sections in this paper do not require the reader to have an in depth knowledge of object technology and C++; an understanding of C programming is required. However, to fully understand the topics on object frameworks, CORBA, and the prototyping results, the reader would benefit from an introduction to object oriented design and implementation in C++. The experienced C++ programmer may skip to the next section.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Object-Oriented

Object-Oriented refers to the design and implementation of information systems which, as mentioned before, combines data structure with the behavior relevant to the data structure. More specifically, object-oriented exhibits the following characteristics: encapsulation; classification; inheritance; and polymorphism.

Encapsulation

Encapsulation is the technique of separating the implementation details of an object from the external interface. The external interface is available for use by other objects. Another name for encapsulation is "information hiding". Information hiding is not restricted to object-oriented but it is available in most object implementations. The concept of external or "public" interfaces on objects is crucial to distributing objects in a system. We will see that it is really the "public" interfaces of objects that are distributed in a system. The actual object implementations are hidden by these distributed interfaces.

Classification

Classification refers to the difference between an "object" and a "class" in an object-oriented system. An "object" is a unique, identifiable instance of information such as a named employee. Classification is the grouping of objects with like attributes (data) and operations (behavior). These groupings are called "classes". Objects are instances of classes. Classes are types whereas objects are identities (e.g. STUDENT is a class and JOE_SMITH is an object which is of type STUDENT). Using the OMT notation 1, classes are diagrammed as a titled box with two sections for the attributes and behaviors. See Diagrams in Appendix.

Inheritance

Classes can be organized such that specialized classes can inherit from more general classes. This technique is called inheritance and it is more easily demonstrated by an example. The classes `car`, `airplane`, `train`, and `bus` could all inherit from the general class `vehicle`. The attributes (data) and operations (behavior) are inherited from one class to a more specialized class. Inheritance is represented in an object model as in Diagrams in Appendix .

Polymorphism

A more abstract object-oriented characteristic is polymorphism. Polymorphism describes the capability of applying the same operation to different objects and having the resulting behavior vary depending on

the class type. As an example, the operation `move` on an object of type `vehicle` might behave differently if the object's specific type is `airplane` or `car`.

Functional vs. Object-Oriented

The discussion of polymorphism sheds light on the real distinction between object-oriented and functional techniques. It's all in the name, "object-oriented". A functional programming language is centered around a hierarchical, functional flow. Object-oriented implementations are centered around object interactions. The semantics of object-oriented programming languages vary, however, the code "reads" the same. As an example, a functional code stub "reads"

```
LAND (AIRPLANE); PARK(CAR);
```

where as the object oriented code would "read"

```
AIRPLANE(LAND); CAR(PARK);
```

In the functional case, the "object" or data is the argument to the function where as the opposite is true for the object-oriented case; the function is the argument of the object. In fact, functions are "members" of classes just like attributes are. In a class they are referred to as "member functions".

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

C++

In the first versions o1, a pre-compiler front-end called CFront was used to translate C++ code into C. Now there are direct to binary C++ compilers. However, the output of CFront is informative because it shows how the C++ language constructs can be implemented in C. Although it is an over-simplification, it can be very instructive for the experienced C programmer to view C++ classes as C structures. Using C, member functions are implemented as structure members that are of type pointers to functions. This is seen in the following structure declaration.

```
struct vehicle {

int max_speed;

int weight;

int length;

void move(int distance);

void change_direction(float angle);

}
```

the above structure declaration is defined and used as follows in a simple program:

```
main() {

struct vehicle V1; // allocate memory

V1.max_speed = 113;

V1.weight = 2000;

V1.length = 10;

V1.change_direction(45.0);
```

C++

```
V1.move(50);

}
```

Notice that the above code segment shows that member functions are neatly engaged just like a classic structure member. The C++ language constructs and semantics hide these complex structures. The real C++ constructs are more "elegant". In C++ the key word `struct` can be replaced with the key word `class`. Inheritance can be demonstrated using the above declaration as follows:

```
class car : vehicle {

char make[25];

char model[25];

void park(void);

}
```

The `car` class inherits from class `vehicle` and all the data members and member functions of `vehicle` are found in `car`.

Encapsulation and polymorphism are demonstrated in the following re-declaration of `vehicle` and `car`. C++ allows for information hiding through the key words `public` and `private` which are modifiers on class members. These modifiers specify the scope of the class' members where the public members constitute the external interface and the private members constitute the internal interface. C++ provides polymorphism with the key word `virtual` which is a modifier of member function. A member function specified as `virtual` will cause the compiler to build a table of pointers to functions for the class called the virtual function table. All classes which inherit from a base class with virtual functions must re-implement the virtual functions locally. The virtual function table will contain all unique, specialized definitions for the original virtual function. When a virtual function is called, the virtual function table provides a level of indirection so that the appropriate function is called.

C++ uses the keywords `new` and `delete` for run-time memory allocation and de-allocation. `new` and `delete` are operators for all objects. Notice in the example below that a pointer to `vehicle` can be assigned an address of a `car` object because `car` is a `vehicle` through inheritance. Therefore, the call to the virtual `move` function uses the virtual function table to engage the `car::move(int)` function.

```
class vehicle {
```

C++

```
private:

int max_speed;

int weight;

int length;

public:

virtual void move(int distance);

void change_direction(float angle);

}

class car : public vehicle {

private:

char make[25];

char model[25];

public:

void move(int distance);

void park(void);

}

main() {

vehicle *vptr1,*vptr2;

car *cptr1;

vptr1 = new vehicle;

vprt2 = new car;
```

C++

```
cptr1 = new car;

vptr1->move(55); // vehicle::move(int)

vptr2->move(55); // car::move(int)

cptr1->move(55); // car::move(int)

delete vptr1;

delete vptr2;

delete cptr1;

}
```

Code Block 1 - Example of Vehicle & Car Class

In C++ there are special functions that handle object initialization and clean-up. These functions are called "constructors" and "destructors". A class constructor is called when an object is defined or explicitly "instantiated" with the `new` operator. A class destructor is called when a previously allocated object goes out of scope or is explicitly freed with the operator `delete`. Constructors and destructors are provided either by default in the compiler or they can be included in the class declaration by the programmer. The constructor must be a part of the "public" interface of the class for an object to be instantiated. Therefore, an "abstract" class can be implemented by restricting the constructor to the "private" interface of the class. The "abstract" class allows powerful class designs as will be shown in the discussion of object frameworks in section 1.5.1. Object frameworks are class hierarchies which use abstract classes and virtual functions to create common interfaces for the framework user.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Distributed Processing

Distributed processing is not as abstract to the newcomer as object technology. For decades engineers have used distributed processing to improve performance of information systems. In general, the more processors you have to perform a task, the faster it will complete. If a computing process can be decomposed into smaller, concurrent tasks then in theory a performance increase can be achieved. The technique of deploying concurrent tasks on either multi-processor hardware or on a multi-processing operating system is called parallel processing.[1] The research presented here concentrates on implementing parallel/distributed processing on multi-computers.[2] A multi-computer a grouping of single processor computers capable of message passing using a high speed network. A small local area network (LAN) of UNIX and/or MS-Windows workstations is an example of a multi-computer. Another popular parallel processing schemes uses symmetric multiprocessors (SMP) which refers to a single block of memory shared by many processors.

In practice, the application of concurrent processing has been mostly limited to the well funded, high performance assets of government agencies and military contractors. The complication and intense training it takes to develop distributed systems has kept them on the shelf and off the drawing board until recently. Now there are thousands of organizations with the appropriate hardware and operating systems. These organizations could be deploying distributed solutions today.

Although implementations are hard to find, there is support for distributed solutions in current CPUs, operating systems, and high speed networks. This investment from the hardware and operating systems manufacturers would probably not be without the advent of object-oriented frameworks for distributed processing. Market forces are driving the manufactures to develop new equipment capable of parallel processing. The market demand is due in part to the advances in object-oriented frameworks for distributed processing. There are already companies investing in object technology and distributed processing by developing object oriented operating systems such as Sun MicroSystem's NEO OS, DOE (Distributed Objects Environment), and Next Computers NextStep/OpenStep OS. [3],

---

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Undergraduate Research

My interest in object-oriented, C++, and distributed processing started during my undergraduate research work at Villanova University, For my undergraduate thesis, I built a prototype software system for the automated design and optimization of simple optical lens systems. It was designed and built on a IBM compatible PC and later ported to a Silicon Graphics (SGI) UNIX workstation. The prototype was developed in C++. The techniques of object-oriented design and the clean semantics of C++ allowed for a quick turn around when producing the prototype. After developing in object-oriented/C++ I was convinced that it was the best technique for writing high performance software.

The prototype results were limited by a lack of both computing resources and a good understanding of simulation techniques. I started thinking about distributed processing as I sat waiting for ray tracing results and waiting to run out of memory. I wondered why the hundreds of workstations networked with my computer could not be used to speed up the prototype. In fact, the other networked computers formed a multicomputer which could be used to distribute processing and share resources. However, even though the hardware was there, a framework of tools for building distributed systems was missing. This research paper presents the requirements for such a framework and shows how object oriented techniques support distributed processing. The following explanation of my prototype design demonstrates why simulation/optimization algorithms consume a lot of computing resources; this explanation will demonstrate to requirement for distributed processing.

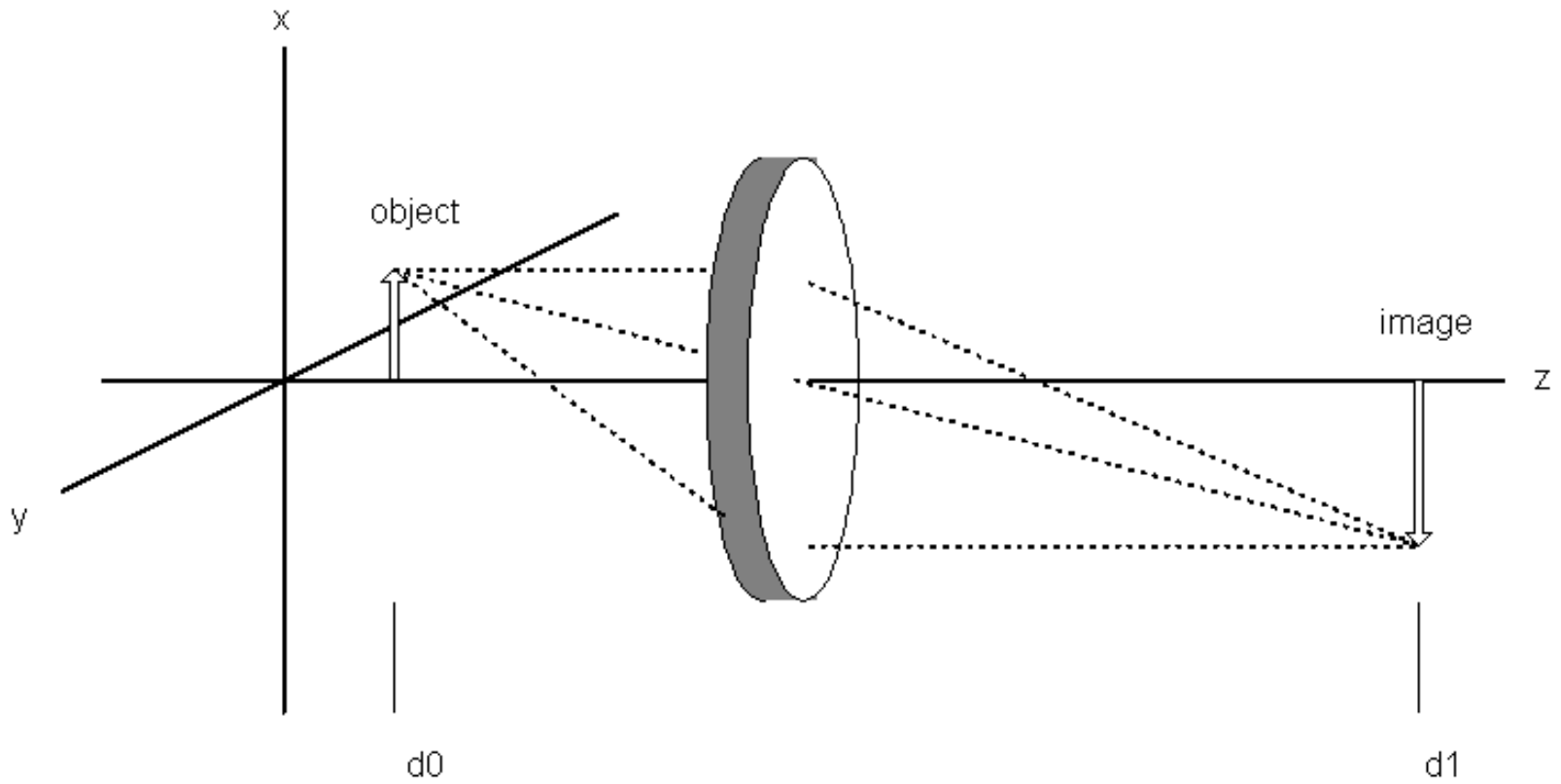
This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Optical System Design

Consider a simple optical lens system such as a microscope. This lens system exhibits optical aberrations that limit the quality of the resulting images. Optical aberrations, like spherical and chromatic aberration and astigmatism, can be reduced and in some cases, removed from the system using various techniques that have been developed for centuries.[4] During my undergraduate work on the prototype I was limited by the infinite search space that resulted when calculating the optimal result in an optical element's configuration.

Take the example of the simplest optical system with the most trivial constraints: a magnifying glass that can be translated in one (1) dimension. The magnifying lens is a first order (circular) convex-convex lens. The lens is perpendicular to the optical axis 'z'. The magnifying lens can be translated along this optical axis a distance 'k' from the object point. Other variables include the desired location of the image point (focal point), the curvature (radius) of the lens surfaces, the thickness of the lens, the material of the lens and environment. See Figure 1. The unknown value is the location of the lens along the optical axis. The goal is to find this unknown value so that the image is brought into focus. Even though this simple problem is usually solved using high school physics equations, a search algorithm is applied for demonstration purposes.

The domain of the lens location is from d0 to d1. The algorithm will be a search over the domain of [d0,d1]. This domain is the search space. The size of the search space depends upon the precision of the solution or rather the delta Z along the optical axis. Given a function Q(z) which evaluates the quality of the image at location d1 over the domain Z, the optimal location of the lens z can be determined. Again, ignoring the obvious solutions using physics laws and calculus to find inflection points of the function, it follows that all points along Z must be evaluated and sorted to find the optimal Q(z). This extreme example uses only simulation. A solution provided by physics equations, geometry and/or calculus is an analytical solution.



W2WTHUMBFigure

1 - Simple Optical System

In my undergraduate prototype I used analytical techniques to solve problems. I attempted simple search algorithms but these techniques consumed my computational resources too quickly. In the real life case, the simulation technique is not longer trivial because the search space becomes huge. This is seen when all the real world factors to the simple lens design problem are considered such as attitude and position of lens elements in three dimensional (3D) space, multi-spectral radiation, gradients in temperature, gradients in material, nth order optical surfaces, etc.. The simple search space would be the cross multiplication of all the domains of the said factors. The inability to solve this problem during my undergraduate work sparked an interest in optimization algorithms.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Work Experience

After I completed by undergraduate degree, I worked in the Edison Engineering Program at GE Aerospace. For three years I worked on large government information and aerospace systems. As a member of the Edison Program I attended Villanova University again for my masters. In Villanova's Intelligent Engineering curriculum, I was introduced to the computer engineering topics of expert systems, fuzzy logic, thinking algorithms, searching algorithms, recursion, dynamic programming, image processing, neural networks and more.

Throughout my work and studies in the Edison Program I was presented with software requirements that were well suited for object technology, distributed processing, and the new techniques I was learning in Villanova's Intelligent Engineering curriculum.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

1.4.1

As one example, I worked on a project that required a Global Information System (GIS) which specialized in generating and displaying terrain maps in 3D. At the time it was not possible to acquire commercial off the shelf (COTS) map databases easily. However, we were able to obtain the U.S. Geological Survey's Digital Chart of the World (DCW) product which contained over (1.5) gigabytes (GB) of mapping data. The maps were displayed on high speed SGI IRIS/VGX UNIX workstations.

The DCW described map data in 2D latitude and longitude coordinates. The elevation data was separate. The map data was arranged as points, lines, and polygons for cities, roads/rivers, and areas respectively. There were many polygons in the DCW (islands, countries, states, lakes, etc.) that contained more than 256 points. Also, many of the polygons were not convex but rather concave. A convex polygon is a three or more sided shape where a line drawn between any two points does not intersect an edge. A triangle is a convex polygon. [5]

The SGI Graphics Language (GL) we used required that graphics data be compiled into vectors of 4D vertices (x, y, z, and 24-bit color value). The vectors could be of length one, two, three or more which allowed points, lines, triangles, or polygons to be drawn. The vectors had to be less than 256 and represent convex polygons

The minority of polygons in our database which represented larger than acceptable polygons and/or concave polygons had to be "fixed" - split into convex polygons with less than 256 points. In the end, we manually "fixed" the few hundred polygons. At the time I wanted to employ some algorithms that would increase the quality, and performance of the mapping system.

The graphics engine in the SGI VGX is optimized for vectors with three (3) points. The graphics pipeline renders all polygons as triangles. If a >255 point vector enters the graphics pipeline it is decomposed into adjacent triangles optimized for rendering. It is possible to achieve a performance increase of orders of magnitude if data is submitted as optimized triangles. Other hardware venders have graphics engines that are optimized for quadrilaterals.

I investigated algorithms that would decompose any polygon into triangles or quadrilaterals that are optimized for a given set of constraints. I had trouble finding solutions because of combinational explosion of the search space. Exploring only geometric decomposition by hand, it can be shown that there are only a few ways to decompose a square or pentagon. However, one learns that a black board will not be big enough to decompose even simple shapes like hexagons and octagons let alone polygons with hundreds or thousands of vertices. In order to insure that all possible decomposition's are known (insure an accurate search space), recursion can be used to determine the number of possible paths to a

solution.

P(n) = paths per n	n-2	solution space
P(3) = 1	1 triangle	1
P(4) = 2 * P(3) = 2	2 triangles	2
P(5) = 5 * P(4) = 10	3 triangles	5
P(6) = 6 * P(5) = 60	4 triangles	14
P(7) = 7 * P(6) = 420	5 triangles	?

Table 1: The recursive determination of search paths.

Consider a fifteen (15) sided polygon. P(15) = 108,972,864,000. This polygon would call for over 100 billion state calculations to find a search space of (I am guessing) hundreds of thousands if not millions of unique states. If heuristics and rules were employed to reduce the search space size to the square of the vertex count [$P(n) = n^2$] or even to [P(n) = n * log(n)], then a solvable algorithm could be developed. However, a polygon with hundreds of points would still be time consuming to decompose. I became determined to learn more about the real application of intelligent search algorithms. I also considered the use of distributed processing to calculate associated search paths concurrently.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Airport Traffic Incursion Detection

On another project, I helped develop software to detect and prevent incursions (potential collisions) of airplanes and ground vehicles at airports. The airport incursion detection system used Global Positioning Satellites (GPS), radar, lasers, and other sensors to create a real-time digital map of the vehicles at an airport. The numerous of channels of sensor data were fused at a central location where incursion detection would took place. To test the data fusion and incursion detection algorithms vehicle simulators were required to produce test scenarios.

We choose an object oriented design and an implementation in C++ for the simulator. The object-oriented characteristics made the simulator easy to prototype. As an example, the following entities of the simulator were all objects: `command`, `point`, `vector`, `unit`, `vehicle, airplane`, `I/O`, `time`, `queue`, `list`, and `display`. This allowed us to build the simulator and get it running without having to flesh out all the details. We could test the command interpreter and dispatch flight commands to the vehicles even if the flight command implementation was an empty stub. In functional designs this quality would be called modular. In object-oriented it is more than modular because the object abstracts its data and functionality behind the object's interface.

We considered making the simulator distributed by encapsulating or "hiding" a interprocess communication (IPC) BSD socket framework in the lower level objects. As an example, a command dispatcher object had a reference (pointer) to a command stream object. The dispatcher did not need to know that the real command stream was coming over a TCP/IP socket from another process. The concept of information hiding has been around for a while and used with success. However, in object-oriented/C++ the interface typing is tighter and in fact the methodology calls for information hiding. It is the definition of classes: clearly defined/typed interfaces to functions and data.

The simulator's success proved that object-oriented and C++ was very powerful. Also, our success proved that well designed object systems can be made distributed with less re-coding than more common functional designs. Another important lesson learned from this project was that the engineering team, which we (the systems developers) had working on the critical methods for vehicle dynamics, was able to understand the software design during the development cycle. The qualities of object-oriented design allowed for more fluid communication between the engineers and programmers. One need not translate from reality into C functions, structure, and global/local variables. The discussions were about vehicles, commands, time, velocity vectors, etc. - the same names for the objects that modeled them.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Graduate Research

Two years after I had completed my undergraduate optics project, I was confident in my original idea that object-oriented technology was a viable alternative to common functional models. I was convinced that distributed systems, especially those designed object-oriented, were not limited to hardware intentionally designed for parallel processing like Cray, Thinking Machines, and Intel Paragon. Given a robust development environment, high speed network, and powerful operating system like UNIX, or Windows NT, any organization could harness distributed processing.

Villanova has many powerful UNIX workstations, hundreds of 486/Pentium PCs, and a campus wide area network (WAN) integrating various local area networks (LAN). If a framework for building distributed systems could be found that coupled heterogeneous systems at the application level, Villanova could support distributed systems. The goal of the research in this report is to prove that practical access to parallel processing using commonplace hardware and object oriented design is possible. The first part of my graduate independent research was an investigation of distributed object frameworks. The following section introduces the reader to object frameworks and my research results.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Object Frameworks

In the case of object-oriented/C++, a software development framework is a combination of pre-defined class hierarchies, libraries, design patterns, and tools that allow for re-useable and consistent software. Frameworks are similar to programming standards like IBM's Standard Query Language (SQL), SGI's Graphics Language (OpenGL), BSD Sockets, X11, etc. Common object frameworks in this report refer to those which are used for general software development, not just database, graphics, user-interface, or communication systems.

In the past, as more complicated client-server and peer-to-peer systems were developed, the expertise needed to overcome the intricacies of network programming was a major problem. In the late 1980s many companies started investing in common frameworks for application programming. The two leading C/C++ compiler venders for PCs, Microsoft and Borland, developed there own frameworks for building consistent standalone MS-Windows applications; Microsoft Foundation Classes (MFC) and Borland's Object Windows Library (OWL). Over the past five years, commercial frameworks have evolved into two types: high powered client server frameworks for 32-bit operating systems; and high level visual programming tools like MS VisualBasic and Borland's Delphi. The latter are very popular because they allow for Rapid Application Development (RAD) on networked PCs and small database servers.

The market for larger industrial strength object frameworks for 32-bit systems had been small but promising. It is a fact that information system providers do not like to buy into technology that is neither guaranteed to be supported for a long time nor not part of a standard. Therefore, while industry leaders and research institutions moved slowly on UNIX standards like POSIX, X/OPEN, DCE, CORBA, CDE, .etc, information system providers have continued to produce systems without reuse, common frameworks, or maintainability in mind. However, there were some frameworks built for internal use at various research institutions and companies possessing enough foresight.

These internal frameworks turned out to be a good investment. Framework development added to the organization's technical base and the benefits of common frameworks could be realized before the standards were delivered. Some of these home grown frameworks, like the National Institute of Health's Class Library (NIHCL) built by Keith Gorlen, are quite popular now and are the basis for many commercial systems.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Hughes Class Library

I was introduced to the technology of object frameworks at Lockheed Martin (GE Aerospace) where I worked with a object-oriented/C++ framework built by Hughes Aircraft called Hughes Class Library (HCL). I investigated using the design patterns found in HCL for various portions of a satellite ground system. The engineers at Hughes Information Technology Center (HITC) developed a COTS product based on HCL called Delphi for Satellite Mission Scheduling and Planning. As the prime contractor, Hughes is using Delphi or a evolution of its technology today to build the ground station and dissemination systems for the Earth Observation Satellite (EOS).

HCL was the first framework I encountered that allowed for the development of complex distributed systems using a pure object oriented paradigm. During my investigation of HCL I discovered Keith Gorlen's NIHCL, a common object framework that the engineers at HITC partly based their work on. This reuse will be seen in Section 2.6 when NIHCL is shown in detail.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Thesis Topic

I investigated object technology as a topic for my graduate independent study and thesis. I wanted to try and build distributed solutions for some of the real-world problems that I found in the aerospace and business industry and prove their viability. In order to build the prototypes I needed a development environment that would support such distributed object frameworks. It is impossible for one person to make a complete distributed object framework - one must re-use the work of others.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

The Search for the Better Framework

Throughout the second half of 1994 I searched the Internet for object-oriented/C++ frameworks in fulfillment of my graduate research requirements. What I found was both good and bad: good because object-oriented/C++ frameworks were proliferating at universities and research organizations; bad because there was a lot of duplication of effort. Almost all of the research I found was not robust enough for my requirements.

I needed a framework that was designed for the purposes of distributing objects; passing objects between processes on the same host and on remote hosts. Most of the frameworks I found were C++ wrappers around UNIX IPC mechanisms. These frameworks allowed for faster, type safe way of UNIX system programming but not distributed objects.

While reviewing technical journals that I found some viable candidates for my framework. It was in the C++ Report that I found articles on Doug Schmidt's Adaptive Communication Environment and the Common Object Request Broker Architecture (COBRA).[6]

"The Adaptive Communication Environment (ACE) is a collection of reusable C++ class libraries and object-oriented framework components that enhance the development of distributed applications ..."

Doug Schmidt, now the editor of the C++ Report, promotes the ACE framework for use in building the infrastructures of large distributed systems. However, the ACE framework does not address problem of distributed objects - although, ACE does not try to satisfy this requirement. ACE is a low level, systems programming framework.

I also was introduced to an organization, The Object Management Group (OMG), which was developing standards for distributed object systems since 1989. The result of the ideas and technology of the OMG's members was the Object Request Broker (ORB). An ORB is a system that provides services for object location, 9 At the end of 1994 I found a number of companies offering products based on the OMG's revision 1.1 of the CORBA specification.

- [IBM] Distributed System Object Model (DSOM)
- [Expersoft] XShell
- [Iona] Orbix
- [PostModern] ORBeline
- [Hewlett Packard] Orb Plus
- [Digital] Object Broker

- [SunSoft] Distributed Objects Environment (DOE)

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

ORBeline

One company, PostModern Computing (now Visigenic), had a framework for the development of distributed systems called NetClasses. In 1994, with the release of CORBA 1.1, PostModern released ORBeline. ORBeline was built on the proven NetClasses technology. At the beginning of 1995, PostModern offered ORBeline to universities for research purposes. I had two (2) viable development tools to choose from: HCL, and ORBeline. I choose ORBeline for obvious reasons.

- ORBeline is built to the CORBA specification that is the industry standard.
- ORBeline is a commercially supported product.
- ORBeline is available for many platforms and compilers including GCC
- HCL is a proprietary framework.

I requested research support from other commercial ORB vendors including Iona Technologies Inc., Candle Inc., HP, IBM, and DEC. At the time, these companies either declined support, or offered support in form of single node software licenses (runs on one host). What good is an ORB that runs on one (1) machine only? The marketing supervisor Suresh Challa and other PostModern staff members have been very helpful and were very generous to allow me to use ORBeline without sizing restrictions for my research project. Late in 1995, I received some licenses to run Iona's Orbix on Windows 3.1. The ORBeline product supports both the UNIX and NT side of my research and the PC workstation requirements will use the Orbix 3.1 (never completed) - Villanova has hundreds of PCs and many UNIX workstations. The majority of my investigation into distributed programming using ORBs deals with the ORBeline system because I have had access to it for a year.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Research Schedule

This research has been submitted to the computer engineering department in two (2) phases. The first submission was for an independent study course. It included reviews of distributed object frameworks, CORBA, and selected topics in intelligent algorithms. Also, as a proof of concept, a prototyping plan was presented. A supplement to the first report was submitted to complete the thesis requirements. Included in this last submission was the software prototype which was implemented on a distributed, heterogeneous multi-computer. This multicomputer consisted of both Sun Sparc workstations running Solaris and SunOS and PC-compatibles running MS-Windows NT (i.e. SVR4, BSD, NT-WIN32). The prototype implements a parallel version of a dynamic programming algorithm which calculates the optimal solution to both the matrix multiplication order problem and the polygon triangularization problem (See Section 1.4.1) in parallel.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Object Frameworks

High level, domain specific software is usually built on top of low level operating system, database, communication, and user interface services. The programming interface to these services is complicated and software written at this level is error prone and difficult to maintain. It has been common practice to assemble common service requirements and construct high level APIs or service layers. This proven technique has evolved in the form of object frameworks.

Object frameworks provide a consistent and efficient foundation for producing object oriented systems. Object frameworks consists of general, reusable, objects hierarchies in the form of code, binary libraries, compilers/parsers and design patterns. They are more than application programming interfaces (APIs). Frameworks not only provide developers with general objects like storage classes and file I/O but they also encapsulate complicated operating system services and proven design patterns.

These common object frameworks allow for very efficient use of lower level system services through consistent object patterns. An object framework may allow an application level object to be pulled from a database, distributed to remote and local processes, replicated to redundant systems, automatically made persistent, and displayed to an operator. In this way, complicated distributed systems can be designed and built using application level, abstract service layer objects.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Acceptance of Object Frameworks

for many reasons. The reason that object frameworks are not in widespread use today in large projects is lack of adequate training. Many engineers considered qualified in object-oriented analysis and design are actually missing important concepts. Object oriented methodology fundamentals are taught well but implementation strategies of object oriented designs are presented in a limiting way.

Students of object oriented are correctly shown that all information system problems including data-processing, simulation, command and control, and real-time are candidates for object oriented designs. However, they are mislead when example implementations are presented to them. These design and coding examples show object techniques used at the application domain level only. Operating system, database, and networking services are excluded from the design. Therefore, when real-world solutions are attempted in industry, the engineers only apply the object techniques to the domain information, not the whole architecture. Students are rarely shown that features of operating systems, databases and networking protocols can be encapsulated in SmallTalk or C++ objects just like bank accounts and personnel data.[8]

It is complicated and costly to develop software on high performance platforms and operating systems like UNIX. Students have been taught that object oriented techniques facilitate higher quality software in these environments. The information system industry is struggling to deploy distributed systems. The complication and detail of new operating systems, network programming APIs, and parallel architectures is making this deployment slow. Object frameworks apply object technology to simplify distributed programming projects.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

The Divided Framework Camp

There have been two strategies used when designing general object frameworks in C++. (1) Design the framework so that almost all classes are part of the same inheritance tree. In other words, almost all classes are descendants of a super, abstract object. This is similar to the approach seen in SmallTalk. This is called the 'single tree' approach. (2) Design your framework so that it doesn't. This is called the 'forest' approach because there are many trees. For now I am partial to the single tree approach but I have seen some interesting designs using templates and the new Standard Template Library (STL). The STL is not presented in this paper.1

The strategy you should use for an object framework depends on the requirements of the system. One requirement or capability called run-time type information is presented next as an example of a requirement that can only be satisfied with the single-tree object framework. There is presently an effort under way to produce an ANSI/ISO standard for C++ which includes come of the capabilities here including run-time type information.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Run Time Type Information (RTTI)

When writing high quality object systems it is necessary to have collections of objects. As an example, one might need a linked list of bank accounts for a finance program. In a classic C program a linked list is implemented using pointers. A node in the list has a pointer to the next node and a pointer to the data element in the list. This data pointer could be typed to be a pointer to a `bank_account` structure. If there are many types of bank accounts then the `bank_account` structure could be expanded to handle the general case and a type field could be placed in the header of the `bank_account` structure. In this way, a `bank_account` typed linked list can be populated with heterogeneous set of bank account information. See below.

```
struct bank_account {

short account_type;

...

};

struct node {

struct node *previous;

struct bank_account *data;

};

struct linklist {

struct node *head;

struct node *tail;

};
```

Code Block 2 - Example Bank Account Linked List in C

Through object oriented techniques, we can improve on this design by making the linked list typed to a abstract account object. Every type of bank account could be implemented using this design through inheritance. A `checking_account` class could inherit from the abstract `bank_account` class. In fact, all accounts could be descendants of the abstract `bank_account`. All account classes can be stored in the linked list because all accounts are of type `bank_account`. The common operations that can be performed on the accounts are limited to the set of virtual functions in the base class `bank_account`. The object oriented design provides a powerful storage mechanism but the retrieval is still complicated. When a retrieval is made on this link list the result is a pointer to a `bank_account` object. However, the pointer actually represents a specialized, concrete account type.

In order to safely type cast a pointer to a `bank_account` object to its actual defined class, type information can to be stored in the object. Explicit type information, like the type field found in the C implementation `bank_account` structure above, is considered undesirable in object oriented designs. Some argue that a design that requires type information in an object is "ugly". The argument is that the need for type information can be satisfied instead with a better inheritance scheme or application of polymorphism.

However, type information is required for truly heterogeneous collections. Actually, more than just type information is needed to use dynamic heterogeneous collections. RTTI provides the basis for two capabilities: cloning and narrowing.

- Cloning is the ability to duplicate (memory copy) an object without specifying the type explicitly. Normally, to make a copy of an object the size of an object must be known and the type of an object must be known to determine its size.
- Narrowing is the ability to automatically and safely type cast an object to a more specific definition of it. For instance, a pointer to a `back_account` object might need to be safely type casted to a more specific account type without explicit type information in the public interface of the `bank_account` object.

RTTI is an newly adopted requirement of the ANSI C++ standard. However, without standard RTTI and the complicated template classes in the STL, framework designers for years have implemented RTTI with a proven technique of single tree libraries. Briefly stated, RTTI is achieved by including the mechanism for type information in the interface of the root object of the tree library. In this way, the RTTI mechanism is a primitive capability of the framework. Most single tree implementation include additional capabilities in the root object; many of these primitives depend on the RTTI.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Serializing the Object

Another capability that the single tree object framework provides is serializing objects. Objects can be sent across an I/O stream when it is serialized. For instance, an object can be passed between local and remote processes, saved to files, and shared between heterogeneous networks. To this day, I have not found a solution to this distributed object requirement other than the single tree design.

When information is stored to disk or passed over the network it can be read back into a process or received at the other end of the network connection because the reader has knowledge of the format of the information. In addition to the format or size of the information, the reader must know the type currently being read. In a synchronous algorithm, the flow of the algorithm determines the type of the information being read. In the asynchronous case, each segment of information being read must have a common header which contains type information. Using type information, specific formats can be selected from the set of all known formats. As an example, the software might require that all records stored to disk be preceded by a type header and all records/messages sent over a serial stream shall be preceded by the common header also. The common header requirement of the software forms a contract or shared interface to information.

The single tree object framework implements a protocol for distributing objects in a more elegant fashion. The contract or agreement of header format is in the abstract class - the root of the tree. All objects in the framework agree to the contract (method of serializing objects) by inheriting from the abstract root object who declares the format. The single tree framework satisfies the requirement of serializing objects. In general, common object requirements are most easily satisfied by this way as shown in the following section.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Framework and Object Requirements

Consider the question about the goals and capabilities of a framework. If the requirement of a framework is that all objects have a common, primitive set of capabilities and the capabilities are not provided through a language primitive then the single tree library is a viable solution. Some argue this statement by presenting object frameworks that are not singly rooted. These arguments prove to be weak when it is pointed out that even powerful libraries like the Adaptive Communication Environment (ACE) lack many higher level framework requirements like serializing objects. ACE does not use RTTI because there isn't any requirement on ACE to use RTTI. Object frameworks should be compared first according to the requirements of the framework. The goals of NIHCL are different from those of ACE. Therefore, they should be compared carefully if at all.[7]

Other reasons presented in opposition to the single tree object frameworks deal with performance and integration restrictions. A deep class inheritance tree tend to over complicate the virtual function tables, especially when multiple inheritance is used. Dispatching paths through a complicated class tree can create performance problems. The careful analysis of frameworks like NIHCL, HCL, NetClasses, etc. show that most interfaces are designed using inline functions and minimized levels of indirection to provide acceptable performance.

The argument that single rooted libraries do not integrate efficiently with other libraries is misleading when the integration concerns libraries with duplicate capabilities. If reasonable consideration is taken during the design of a single tree library, then integration with other frameworks can be done. Integration of frameworks with overlapping capabilities is always difficult. A good example of this is the concurrent use of the C++ I/O stream library with the C standard I/O library routines which can lead to buffering problems.

I have only found two type of object frameworks - those that do and those that do not use the single tree. Excluding the newly born technology of the ANSI/ISO C++ standard template library (STL), I characterize these two design strategies further: *there are frameworks that do not use the single tree paradigm and there are frameworks that work - there are engineers who debate the single tree paradigm and engineers who get the work done.* In this paper, I present frameworks from both camps. The ACE framework is a forest implementation but ACE doesn't satisfy requirements such as object serializing or heterogeneous collections.

An informative debate should be focused on the goals and requirements of a desirable framework before investigating the use of a single tree or forest. Once these capabilities are determined, the design strategy will be obvious. It can be shown that the single tree strategy satisfies certain library requirements where

forest strategy does not - and vice versa.

The following sections present four object frameworks which were encountered during this research. The NIH Class Library is presented first. The Hughes Class Library and NetClasses Library follow. These two proprietary libraries demonstrate the reuse of many framework techniques from the NIHCL which is a federally funded public domain software. Finally the Adaptive Communication Environment (ACE) is presented not only because it is a high quality framework but also because it is implemented as a forest of objects as opposed to the first three frameworks which are single rooted.

It would take volumes of documentation to fully cover all capabilities and design techniques of the following frameworks. Therefore, only crucial library design details from each framework are discussed here. Special interest is taken in the implementation of the root objects of the single tree libraries and the primitive classes of the ACE library. The interest here is to determine the techniques used by framework designers to satisfy distributed object requirements.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

National Institute of Health Class Library (NIHCL)

The NIH Class Library (NIHCL) [8] is composed of around sixty (60) general purpose classes that all belong to a single inheritance tree with class `Object` as the root. The class `Object` declares data members and virtual functions that allow for copying, printing, storing, reading, and comparing objects for all classes in the library that inherit from `Object`. Almost all classes in the NIHCL and any new class added by users of NIHCL are ultimately descendants of the abstract class `Object`. All these specialized classes inherit the public interface of the class `Object` which allows a common interface for all objects beyond the primitive capabilities of the language. This technique, introduced in the previous section is called a single root tree or common object library.

It is important to note that it is the interfaces to these common functions that are inherited, not the implementations of these functions. The virtual functions of the root `Object` implement the polymorphic characteristic of the common functions in all descendent objects. The inheritance tree for most of the NIH classes is shown below.

```
NIHCL - library static member variables and functions

Object - root of the NIH Class Library inheritance tree

|-Bitset - set of small integers

|-Class - class descriptor

|-Collection - abstract class for collections

| Arraychar - byte array

| ArrayOb - array of Object pointers

| Bag - unordered collection of Objects

| SeqCltn - ordered, indexed collections

| Heap - min/max heap of Object pointers
```

```
| LinkedList - singly linked list

| OrderedCltn - ordered Object pointers

| SortedCltn - sorted collection

| KeySortCltn - keyed sorted collection

| Stack - stack of Object pointers

| Set - unordered collection of non-duplicate Objects

| Dictionary - set of associations

| IdentDict - keyed by Object address

| IdentSet - set keyed by Object address

|-Date - Gregorian Calendar date

|-FDSet - set of file descriptors

|-Float - floating point number

|-Fraction - rational arithmetic

|-Integer - integer number object

|-Iterator - collection iterator

|-Link - abstract class for LinkedList links

| LinkOb - link containing Object pointer

| Process - co-routine process object

| HeapProc - process with stack in free store

| StackProc - process with stack on main() stack

|-LookupKey - abstract class for Dictionary associations

| Assoc - association of Object pointers
```

```
| AssocInt - association of Object pointer with Integer

|-Nil - the Nil object

|-Point - X/Y coordinate pair

|-Random - random number generator

|-Range - range of integers

|-Rectangle - rectangle object

|-Semaphore - Process synchronization

|-SharedQueue - shared queue of Objects

|-String - character string

| Regex - regular expression

|-Time - time of day

|-Vector - abstract class for vectors

| BitVec - bit vector

| ByteVec - byte vector

| ShortVec - short integer vector

| IntVec - integer vector

| LongVec - long integer

| FloatVec - floating point vector

| DoubleVec - double precision floating point vector

|-ReadFromTbl - tables used by Object I/O readfrom()

| OIOifd - file descriptor Object I/O

| OIOin - abstract class for Object I/O

| OIOistream - abstract class for stream Object I/O

| OIOnihin - stream Object I/O
```

|-Scheduler - co-routine Process scheduler

|-StoreOnTbl - tables used by Object I/O storeOn()

OIOofd - file descriptor Object I/O

OIOout - abstract class for Object I/O

OIOostream - abstract class for stream Object I/O

OIOnihout - stream Object I/O

Figure 2 - Inheritance Tree of NIH Class Library

The following discussion of the NIHCL Object and Class classes will refer to selective code blocks in Listing X and Y respectively. Full code listings for pertinent NIHCL classes are in the appendices.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

RTTI

The RTTI capability is implemented in the NIHCL with `Class` object. The `Class` object is not a user object like `Point` or `Date`. The `Class` object encapsulates the "type" information of all NIHCL objects (all objects that inherit from `Object`). Every NIH class has a static member variable which is an instance of `Class`.[9] Class `Object` declares a virtual function

```
virtual const Class* isA() const;
```

which returns a pointer to the `Class` object which is a static member of the referenced object. As an example, if you had a pointer to an `Object` which is really of type `Point`,

```
Object* obj_ptr = new Point(0,0);
```

then a pointer to the `Point::classDesc` static member variable can be obtained using[10]

```
const Class* class_ptr = obj_ptr->isA()
```

Even though `obj_ptr` is of type `Object*,` the virtual specification of `Object::isA()` will map to the `Point::isA()` function (if the object that `obj_ptr` points to is really of type `Point`). This is how all the virtual functions declared in `Object` work; create new primitive capabilities for the classes of the tree.

The `classdesc` member variable of NIHCL classes is designated as "private" so that is can not be modified easily. Explicit access to the static value of `X::classdesc` is through the static member function

```
const Class* X::desc() { return &classDesc; }
```

Again, a pointer to the `Class` instance for any NIHCL class can be accessed without an instance of the class.

```
const Class * class_ptr = Point::desc();
```

The RTTI capability is accessed through two functions declared by all NIHCL classes

```
bool isMemberOf(const Class&) const

bool isKindOf(const Class&) const
```

The `isMemberOf` member function returns `YES` if the object it is applied to shares the argument `Class`. The `isKindOf` member function return `YES` if the object it is applied to shares the argument `Class` or is a descendent of a class which shares the argument `Class`. As seen in the example below, a `String` object is a kind of `Object` but not a member of the `Object` class.

```
String s("Villanova");

if(s.isMemberOf(*String::desc())) // YES

if(s.isKindOf(*String::desc())) // YES

if(s.isMemberOf(*Object::desc())) // NO

if(s.isKindOf(*Object::desc())) // YES

if(s.isMemberOf(*Point::desc())) // NO

if(s.isKindOf(*Point::desc())) // NO
```

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Object Comparison

Another primitive capability of the NIHCL is object comparison. A robust comparison capability both simplifies higher level application programming and also enables the use of complex collection classes such as bags, dictionaries, sets, etc. There are many ways to compare objects: data member comparison, address comparison, and abstract comparisons. Object identification, versions, equality, and scope are all part of the comparison capability. In the NIHCL, comparison is based on an abstract notion of an object's "species". Objects that belong to the same species can be compared. It is a run-time error to compare objects that are not of the same species. The function

```
bool isSpecies(const Class&) const;
```

is similar to `isMemberOf` except that `YES` is returned only if the object that `isSpecies()` is applied to is part of the species of the argument `Class`. Therefore, the equality of objects is determine by the function

```
bool isEqual(const Object&) const;
```

which first checks the species of the argument objects and then checks their explicit equality with the `operator==()` member function. As an example,

```
bool Point::isEqual(const Object& p) const

{

return p.isSpecies(classDesc) &&

*this == (const Point&)p;

}

bool Point::operator==(const Point& p) const

{

return (xc==p.xc && yc==p.yc);
```

```
}
```

Another point of view of equality is whether two argument objects are not just `isEqual()` but are the same object in memory.

```
bool isSame(const Object&) const
```

The difference between `isSame()` and `isEqual()` can be seen by example:

```
String a("Villanova"), b("MRJ"), c("Villanova");

if (a.isEqual(b)) // NO

if (a.isSame(b)) // NO

if (a.isEqual(a)) // YES

if (a.isSame(a)) // YES

if (a.isEqual(c)) // YES

if (a.isSame(c)) // NO
```

The comparison capability is implemented using the functions mentioned above. The compare function

```
int compare(const Object&) const
```

returns a negative value if the reference object is "less" than the argument object; positive if greater than and zero if equal. Again, it is a run-time error to compare two objects that are not of the same species.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Collections and Iterators

It was mentioned in the previous section that it is sometimes necessary to have heterogeneous container classes. The NIHCL allows for this using the robust set of container classes which all inherit from the abstract `Collection` class. The class `Collection` is the primitive interface of all container classes. The class `Collection` models a container of `Object`(s). Because `Collection`(s) can contain instances of `Object`(s), `Collection`(s) can contain another `Collection` which allows for powerful data structures.

In addition to the extensible `Collection` classes, the NIHCL implements a new design pattern called the iterator. The class `Iterator` encapsulates the functionality of moving through a `Collection`. It is called an iterator because it can travel through a `Collection` in many ways: forward, backward, etc. The `Iterator` is related to the `Collection` class so any `Iterator` will work with any class inheriting from `Collection`. This Collection-Iterator model provides a level of indirection that allows for multiple, simultaneous consumers without the performance penalty of the collection managing consumer reference points.

The NIHCL implements the `Nil` object/class to enhance the capabilities of the `Collection` classes. The `Nil` object inherits from the `Object` class so that when a programmer needs to use a reference to nothing (nil), the primitive operations allowed for `Objects` can be applied without causing fatal errors. The `Nil` class prevents the unsafe use of `NULL` or `0` in pointers.

```
class Object {

private:

static Object* reader(OIOin& strm);

static Object* reader(OIOifd& fd);

protected:

void Object(OIOifd&);

void Object(OIOin&);

virtual void storer(OIOofd&) const;
```

```
virtual void storer(OIOout&) const;

void Object(void) {}

public:

static Object* const nil;

static Object& castdown(Object& p) { return p; }

static const Object& castdown(const Object& p) { return p; }

static Object* castdown(Object* p) { return p; }

static const Object* castdown(const Object* p) { return p; }

static const Class* desc();

virtual const Class* isA()const = 0;

virtual Object* shallowCopy() const = 0;

static Object* readFrom(OIOifd& fd);

static Object* readFrom(OIOin& strm);

const char* className() const;

Object* deepCopy() const;

bool isKindOf(const Class&) const;

bool isMemberOf(const Class& clid) const

{ return isA()==&clid; }

bool isSame(const Object& ob) const

{ return this==&ob; }
```

```
bool isSpecies(const Class& clid) const

{ return species()==&clid; }

void storeMemberOn(OIOofd&) const;

void storeMemberOn(OIOout&) const;

void storeOn(OIOofd&) const;

void storeOn(OIOout&) const;

void* _safe_castdown(const Class&) const;

virtual int compare(const Object&) const = 0;

virtual Object* copy() const;

virtual void deepenShallowCopy() = 0;

virtual void dumpOn(ostream& strm =cerr) const;

virtual unsigned hash() const = 0;

virtual bool isEqual(const Object&) const = 0;

virtual const Class* species() const;

virtual void* _castdown(const Class&) const;

};
```

Code Block 3 - NIHCL Object class declaration

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Object Narrowing (safe downcast)

In the previous section the concept of an object downcast operation was introduced. The downcast, or "narrowing" operation, allows an instance or pointer to a general object to be cast to the type of a more specific declaration. In the NIHCL, it is often necessary to downcast a pointer of type `Object` to a pointer of type `Collection`; a pointer of type `Collection` to a pointer of type `Dictionary`. In many cases, information can be managed as pointers to `Object`(s). The generalized virtual functions like copy(), shallowCopy(), storeOn(), readFrom(), etc. can be used until the object needs behave in a specialized manner. Downcasting to a more specific form is required to call specialized member functions. The NIHCL implements downcasting or object narrowing with the static castdown functions

```
static X&

X::castdown(Object& p)

static const X&

X::castdown(const Object& p)

static X*

X::castdown(Object* p)

static const X*

X::castdown(const Object& p)
```

These static function can be called without an instance of a object. Every combination of non-const, const, pointer, and reference argument types is provided. The castdown functions are automatically produced by the NIHCL macros. Also, the complicated situation of multiple inheritance is handled properly in the `castdown()` functions.

The castdown functions work because of the virtual `_castdown()` function. In the example

```
Object* obj_ptr = new Airplane;

Vehicle* veh_ptr = Vehicle::castdown(obj_ptr);
```

The `castdown()` function calls the `_castdown()` function with the pointer to the static class descriptor object of the type to be cast down to as an argument

```
obj_ptr->_castdown(*Vehicle::desc());
```

The virtual nature of the `_castdown()` function causes the "real" `_castdown()` function to be called for the actual type of `obj_ptr` which is `Airplane::_castdown()`

All NIHCL classes implement `X::_castdown()` so that the argument class descriptor object is compared with its own class descriptor. If they are the same - if `obj_ptr` is of type `Vehicle` - then `this` [11]is returned as `*void`. If they are not equal (as in our example where `Vehicle::desc()` is not equal to `Airplane::desc()`) then the `_castdown` function is called for the base class. In this way, the entire inheritance tree is search for a valid cast. The search will continue to the top of the tree at `Object::_castdown()` which will return `NULL` to end the search and the cast will fail. In our example `Airplane::_castdown()` will fail and `Vehicle::_castdown()` is called and `this` is returned.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Copying Objects (Shallow and Deep)

In order to perform object I/O (the ability to send and receive Objects over I/O media like files and communication streams), the ability to accurately copy Objects is needed. In the previous section it was shown that there are many ways to compare Objects. It is the point of view of the Objects that matters when comparing - comparison at the physical (memory) or data level (information)?

Copying Objects deals with similar issues. There two ways that Objects can be copied: shallow and deep. A shallow or deep copy refers to how the resulting Object has its data members copied. The strategy for copying Objects begins the same for shallow and deep copies. In fact, a shallow copy is always done first during a deep copy.

During a copy operation, a new block of memory is allocated from the free store for the new object. Data member values are copied from the source object to the target object memory depending on the type of data member. Data members can be any of the following:

- instance of a NIHCL class
- pointer to an instance of a NIHCL class
- primitive type like int or float
- array (pointer) of primitive types
- non-NIHCL object, or structure

When the data members are instances of data structures or primitives, the shallow and deep copies are the same. In this case, the data member is truly duplicated in memory. However, when the data member is a pointer to an instance of a data structure or primitive, the data member can be shallow copied or deep copied. The shallow copy will duplicate the value of the pointer (address). The deep copy will duplicate in memory the instance of the data structure or primitive type.

The implementation of shallow copy in NIHCL is a virtual member function of `Object`

```
Object* X::shallowCopy() { return new X(*this); }
```

The `X::shallowCopy()` calls the copy constructor of the `X` object. Therefore, the real work of the shallow copy is done by the copy constructor.[12] The `shallowCopy()` function is available to all NIHCL classes and user defined classes that inherit from the NIHCL tree.

In order to handle the deep copy of member variables which are pointers, the NIHCL classes implement the non-virtual `deepCopy()` and virtual `deepenShallowCopy()` functions. The `deepCopy()`

function first creates a shallow copy of the source Object using `::shallowCopy()`. Then, `deepCopy()` calls `deepenShallowCopy()` to change the shallow copied target to a deep version. The `deepenShallowCopy()` function starts by calling the `deepenShallowCopy()` of its parent class. In this way, all ancestor data is properly copied. An Object's `deepenShallowCopy()` function will call `deepenShallowCopy()` on all member class instances and deepCopy on all member pointers to classes.

The goal of the deep copy is to have a new Object which shares no data with the source Object. To do this, the `deepCopy()` function keeps a record of all new instances of member Objects it copies using the `IdentDict` Object. This technique allows for the situation where a source Object has two or more data members all pointing to the same instance of a class. This multi-referenced Object is only deep copied once in the target Object.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Object I/O

The deep copy technique must be implemented before object I/O can be done. The issues involved with object I/O also include the complication of data members with referenced to the same objects. If a target object has many data members that all point to the same object, this multi-referenced object must transferred only once. The values of the data members must be transferred as object references.

Output

The `Object::storeOn()` function implements the I/O capability and is similar to the `deepCopy()` function in that the `storeOn()` function keeps a record of all objects that it has stored. The `storeOn()` function checks this table of stored objects and stores an object reference instead. The `storeOn()` function calls the virtual function `storer()` to actually do the work.

The `storer()` function first calls the `storer()` function of its base class and then stores its own member variables. Member variables are stored by explicitly calling the storage mechanism of its type. The object name is also stored so that later the object's reader function can load the serialized object.

Input

Object input is implemented by the static `readFrom()` function which is the counterpart to the `storeOn()` function. The `readFrom()` function interprets the input stream by reading the objects stored by `storeOn()` and building a table. When the `readFrom()` function encounters an Object reference, the reference is converted into a pointer by looking up the object reference in the table. The `readFrom()` function loads an instance of an object by calling that object's specific I/O constructor. Every NIHCL class has constructors with I/O objects as arguments to the constructor.

The `readFrom()` constructors perform much like the `storer()` functions. A `readFrom()` constructor first calls its base class's `readFrom()` constructor to load the ancestor data members. Then the constructor initializes its own data members by calling the type specific read mechanisms.

The Object I/O is performed in two stages: class independent I/O and class dependent I/O. The `storeOn()` and `readFrom()` functions implement the class independent portion by converting Object pointers to and from Object references using the temporary tables generated during reading and writing. The various Object `storer()` functions and `readFrom()` constructor implement the class dependent I/O.

An integral part of the Object I/O (OIO) capability are the `OIOin` and `OIOout` classes and all their derived Object I/O classes. These classes encapsulate the primitive mechanism that store and retrieve data types in a machine independent, and program independent format.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Writing NIHCL Classes

The NIHCL comes with template files for developing your own classes compatible with the NIHCL tree. Also, the common functions needed for RTTI, Collections, comparison, Object I/O, etc. are easily included in your own classes using an extensive set of preprocessor macros in your class definitions and declarations.

The following code blocks show the detailed pre-processor macros that allow new NIHCL classes to be created with ease. I have found macros very similar to these in other object frameworks. Because of the complicated "contract" that new classes sign up to by inheriting from the NIHCL tree, these macros are necessary to insure correct virtual function declaration and conformance to other requirements of the tree. The common functions need not be typed again because they are produced properly by the macros. The class Point is declared as follows:

```
class Point: public Object {

DECLARE_MEMBERS(Point);

protected:

short xc,yc;

virtual void storer(OIOofd&) const;

virtual void storer(OIOout&) const;

public:

Point () {xc = yc = 0;}

Point (short newx, short newy) {xc=newx; yc=newy;}

short x () const {return xc; }

short x (short newx) {return xc = newx;}

short y () const {return yc;}
```

```
short y (short newy) {return yc = newy;}

Point operator+ (const Point& p) const

{return Point(xc+p.xc, yc+p.yc);}

Point operator- () const {return Point(-xc,-yc);}

Point operator- (const Point& p) const

{return Point(xc-p.xc, yc-p.yc);}

bool operator== (const Point& p) const

{return (xc==p.xc && yc==p.yc);}

double dist (const Point& p) const

{return hypot(xc-p.xc, yc-p.yc);}

virtual int compare (const Object&) const;

virtual void deepenShallowCopy ();

virtual unsigned hash () const;

virtual bool isEqual (const Object&) const;

virtual void printOn (ostream& strm =cout) const;

virtual const Class* species () const;

};
```

Code Block 4 - Example NIHCL Point Class Declaration

Besides the framework functions explicitly placed in the class declaration (shown in bold) the DECLARE_MEMBERS macro supplies the rest of the common functions as follows:

```
#define DECLARE_CASTDOWN(classname) \
```

```
static classname& castdown (Object& p) \

{return (classname&)p;} \

static const classname& castdown (const Object& p) \

{return (const classname&)p;} \

static classname* castdown (Object* p) \

{return (classname*)p;} \

static const classname* castdown (const Object* p) \

{return (const classname*)p;}

#define DECLARE_MEMBERS(classname) \

private: \

static Class classDesc; \

public: \

DECLARE_CASTDOWN(classname) \

static const Class* desc () {return &classDesc;} \

static classname* readFrom (OIOin& strm) \

{return castdown(desc()->readFrom(strm));} \

static classname* readFrom (OIOifd& fd) \

{return castdown(desc()->readFrom(fd));} \

classname (OIOin&); \

classname (OIOifd&); \

virtual const Class* isA () const; \
```

```
virtual Object* shallowCopy () const; \

virtual void* _castdown (const Class&) const; \

private: \

static Object* reader (OIOin& strm); \

static Object* reader (OIOifd& fd);
```

Code Block 5 - NIHCL Framework Declaration Macros

The definition of class `Point` includes the following - notice that the `DEFINE_CLASS` macros takes care of most of the function and member definitions:

```
DEFINE_CLASS(Point,1,"Point.c,v 3.13 92/12/19 15:53:22",NULL)

bool Point::isEqual (const Object& ob) const {

return ob.isSpecies(classDesc) && *this==castdown(ob); }

const Class* Point::species () const { return &classDesc; }

unsigned Point::hash () const { return xc^yc; }

int Point::compare (const Object& ob) const {

assertArgSpecies(ob,classDesc,"compare");

const Point& p = castdown(ob);

int t = yc - p.yc;;

if (t != 0) return t;

else return xc - p.xc; }

void Point::deepenShallowCopy () {}

void Point::printOn (ostream& strm) const {
```

```
strm << '(' << xc << ',' << yc << ')'; }

Point::Point (OIOin& strm) : BASE(strm) {

strm >> xc >> yc; }

void Point::storer (OIOout& strm) const {

BASE::storer(strm); strm << xc << yc; }

Point::Point (OIOifd& fd) : BASE(fd) {

fd >> xc >> yc; }

void Point::storer (OIOofd& fd) const {

BASE::storer(fd); fd << xc << yc; }
```

Code Block 6 - Example NIHCL Point Class Definition

where the DEFINE_CLASS macros is

```
#define _DEFINE_CLASS(classname) \

Object* classname::reader (OIOin& strm) \

{return (Object*)new classname(strm);} \

Object* classname::reader (OIOifd& fd) \

{return (Object*)new classname(fd);} \

Object* classname::shallowCopy () const \

{return (Object*)new classname(*this);}

#define _DEFINE_CLASS_ALWAYS(classname,version,identification,initor) \

Class classname::classDesc (STRINGIZE(classname),\

ClassList(0,BASE_CLASSES,0), \
```

```
ClassList(0,MEMBER_CLASSES-0,0), \

ClassList(0,VIRTUAL_BASE_CLASSES-0,0), \

version, identification, sizeof(classname), \

classname::reader, classname::reader, \

initor ); \

const Class* classname::isA () const \

{return &classDesc;}

#define _DEFINE_CASTDOWN(classname) \

void* classname::_castdown (const Class& target) const \

{ \

if (&target == desc()) return (void*)this; \

return BASE::_castdown(target); \

}

#define DEFINE_CLASS(classname,version,identification,initor) \

_DEFINE_CLASS(classname) \

_DEFINE_CLASS_ALWAYS(classname,version,identification,initor) \

_DEFINE_CASTDOWN(classname)
```

Code Block 7 - NIHCL Framework Class Definition Macros

The introduction to NIHCL presented in this research paper only touches the surface. Many aspects of the NIHCL were skipped including its support for light weight processing (LWP). The goal was to show the NIHCL's support for distributed objects. It will be shown that many designers have reused the coding techniques and design patterns of the NIHCL in their own object frameworks. The next two sections introduce object frameworks which implement industrial strength libraries for distributed object

computing based on NIHCL.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Hughes Class Library (HCL)

The Hughes Class Library (HCL) is a proprietary object framework built and used by Hughes Aircraft for the development of complex, distributed object libraries and applications. I used HCL for over six (6) months at Martin Marietta where Hughes Information Technology (HITC) group was a subcontractor. Because HCL is proprietary material, only an introduction to the HCL design patterns and technology is allowed in this paper; the details of implementation are excluded.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Introduction

HCL is similar and in many ways based on the technology found in NIHCL and it is for both this reason and also HCLís innovative management of distributed objects that I discuss HCL in this paper. I mentioned in Section X that some companies had invested in distributed object frameworks; Hughes Aircraft is one such company. HCL is the basis of many distributed object solutions for the customers of Hughes Aircraft. NASA is using the HCL technology on the Earth Observation Satellite Information System (EOSDIS). Also, Hughes Aircraft has a commercially available product for the mission planning and scheduling of satellites called Delphi in which distributed resource models and activity schedulers were built using HCL.[13]

HCL provides a general framework object applications. However, it is obvious that HCL was developed by an aerospace company because of the robust set of class found in HCL for time management. There are 20 classes associated with date/time management, epoch time information, and temporal-interval objects. In addition to the general framework which HCL provides there is a library extension for the management of distributed objects, object communication, and operating system services called HIPC. The Hughes Interprocess Communication (HIPC) library consists of the following framework extensions:

Nameserver

IPC

Timer

Notifier

Figure 3 has an inheritance diagram of the classes in HCL and the framework extensions. The classes from the framework extensions are labeled respectively. Although they are not included in Figure 3, Hughes also built a HCL extension library for X-Windows programming in OpenLook and Motif. HCL and the library extensions are an excellent example of a industrial strength implementation of distributed objects using a C++ and the single rooted tree design pattern.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

HObject ìthe Hughes version of NIHCL Objectî

Just like in NIHCL, the engineers at Hughes determined that every object in the framework needed to have certain common characteristics and capabilities:

virtual destructor

run-time type information (RTTI)

run-time dynamic object creation

cloning

support for model/view pattern

object I/O streaming and serialization

generic comparison

debugging and logging support

Not only does HCL have the same requirements for objects as NIHCL but also satisfies these requirements by providing a common interface through a single rooted class called HObject. In fact, most of the interface to HObject is the same as NIHCLís Object class. In many cases, the only difference between HObject and Object is in the details like member function names, class names, and other coding styles. However, there are also many differences between HCL and NIHCL. For example, HCL avoids the notion of ìSpeciesî (see section X) in HObject (NIHCL dictates that only Objects of the same species can be compared). HCL has support for debugging or monitoring of object life cycles. HCLís support for multiple inheritance for HObject is not nearly as robust as NIHCL support for multiple inheritance. I think that HCL was developed with some specific software and mission requirements in mind which is why HCL is not as general or abstract as NIHCL. For instance, the elegant concept of class ìspeciesî is missing as are other design patterns because of the strict performance requirements. HCL was probably designed to insure a root class (HObject) that was as lightweight as possible. Some notable features of HCL are shown below.

```
HObject - base class of framework

|-Configuration - provides protocol for configurable entities
```

```
| ConfigurationFile - stores configuration information
|-H2DPoint - unsigned two-dimensional coordinate
|-H2DPointS - signed two-dimensional coordinate
|-Haddress IPC - describes agent by location information
| HAddressCd IPC - describes agent by location information
|-HClassDesc - class meta information for HObject derived classes
| HAbsClassDesc - abstract class meta information
|-HCmdLine - argc and argv command line holding and parsing
|-HCollection - abstract collection
| HObjCollection - base class for HObject collections
| HObjList - object collection using list container
| HIntervalList - collection time intervals
| HObjOrdList - base class for ordered list collections
| HEIntvlListAbs - abstract base class for HEpochInterval lists
| HEpochIntervalList - collection of epoch time intervals
| HObjUniqueOrdList - ordered list collection with unique members
| HIterList - object list of iterators for noids
| HMsgIterList NAMESERVER - noid list iterator
| HObjArray - array collection of HObjects
| HObjSet - object collection using set container
```

| HPoolAbs - abstract object collection using pool container

| HPoolList - object list pool collection

| HDiagMsgPool - storehouse of allHDiagMsgs

| HPoolSet - object set pool collection

| HDiagMsgSet - global storehouse of all HDiagMsgs

| HVCollection - collection of void pointers

| HVList - list collection of void pointers

|-HConnAbs IPC - abstract ipc connections

| HMsgConn IPC - connection between message agents

| HMsgConnPerm IPC - permanent message agent connection

| HMsgConnTemp IPC - temporary message agent connection

| HMsgConnUnk IPC - convertable message agent connection temp/perm

| HMsgService IPC - listens on well known service ports

|-HDAGNode - directed acyclic graph node (member of HDAG)

|-HDate - holds the notion of date YY/MM/DD

|-HDateTime - holds the notion of a time-date HH:MM:SS YY/MM/DD

|-HDay - holds the notion of elapsed days

|-HDayYear - holds the notion of a year and day within the year

|-HDiagMsg - diagnostic message object

| HRankDiagMsg - priority diagnotic message

|-HEpochInterval - closed time interval in epoch time

|-HEpochTime - time since a given absolute time called th epoch

|-HHour - elapsed hours

|-HInterval - time interval

|-HJulianDay - julian day representation

|-HMatrix - 3x3 matrix

|-HMatrix2 - 2x2 matrix

|-HMatrix3 - 3x3 matrix

|-HMatrix4 - 4x4 matrix

|-HMatrixN - NxN matrix

|-HMessage IPC - abstract message

| HMsgData IPC - message representing data

| HMsgAdr IPC - message representing address information

| HMsgIdentify IPC - message identifying address

| HMsgInterest IPC - message representing interest

| HMsgInterestRsp IPC - message interest response

| HMsgReply IPC - message reply

| HMsgRoute IPC - message to be routed

| HMsgCol IPC - message containing collection object

| HMsgTransfer IPC - transffered message

| HMsgAgentShutdown IPC - message representing agent shutdown notification

| HMsgConnType IPC - message containing type of connection

| HMsgDisconnect IPC - disconnect message

| HMsgShutdown IPC - shutdown message

| HMsgStat IPC - status message

| HMsgSysShutdown IPC - total system shutdown message

| HMsgText IPC - text message

| HMsgTextStat IPC - test status message

|-HMinute - elapsed minutes

|-HMonth - elapsed months

|-HMsgAgent IPC - abstract message agent manages connection and address

| HMsgAgentNs NAMESERVER - name server message agent

| HNoidAgent NAMESERVER - noid name server message agent

|-HMsgFinder - finds diag messages

| HRankMsgFndr - finds priority diagnostic messages

|-HNameServAbs NAMESERVER - abstract nameserver

|-HNoid NAMESERVER - lightweight message handler

| HStdNoid NAMESERVER - standard noid

| HStdNoidNs NAMESERVER - noid which uses nameserver for connection

|-HNoidConnAbs NAMESERVER - abstract noid connection class

| HNoidConn NAMESERVER - noid agent connection class

| HExpConn NAMESERVER - explicit noid connection

| HNsConn NAMESERVER - nameserver connection class

| HNsCleaner NAMESERVER - garbage collection of unused addresses

|-HNsAdrCnt NAMESERVER - address and count pair

|-HNotifier NOTIFY - base class of notification managers

| HSelNotifier NOTIFY - select notification manager

| HUnixNotifier NOTIFY - native Unix notification management

| HVmsNotifier NOTIFY - VMS native notification management

| HVmsMNotifier NOTIFY - motif notification for VMS management

| HMotifNotifier NOTIFY - motif notification manager

| HXvNotifier NOTIFY - xview notification manager

|-HNotifyCli NOTIFY - notification client nodes

| HTimNotCli NOTIFY - timer notification client

| HUnixTimNotCli NOTIFY - native Unix notification client

| HMotifTimNotCli NOTIFY - motif timer notification client

| HXvTimNotCli NOTIFY - xview timer notification client

| HVmsTimNotCli NOTIFY - VMS timer notification client

| HIONotCli NOTIFY - I/O notification client nodes

| HSelIONotCli NOTIFY - I/O notification client using select

| HMotifIONotCli NOTIFY - motif I/O notification client

| HXvIONotCli NOTIFY - xview I/O notification clients

```
|-HObjExtAbs - abstract object extractor

|-HObjIter - baseclass for iteration across HObject collections

| HObjArrayIter - iterator across an HObjArray

| HObjListIter - iterator across an HObjList

| HObjListForIter - forward/incrementing list iterator

| HObjListRevIter - reverse/decrementing list iterator

| HObjSetIter - iterator through an HObjSet

|-HObjIterator - iteration across collections of HObjects

|-HObjLink - linked list node

|-HRectangle - rectangle geometry

|-HRectangleS - signed rectangle geometry

|-HSecond - elapsed seconds

|-HSigCatcher - process signal catcher

| HSigMgr - reacts to signals from catchers and dispatches to handlers

|-HSigHand - process signal handler

| HNsSigHand NAMESERVER - name server signal handler

|-HString - null terminated character array

|-HTime - standard time encapsulation

|-HTimeCvtrAbs - overridable timer conversion algorithms

| HTimeCvtr - overridable timer conversion algorithms

|-HTimeProbe TIMER - keeps track of named instant in time (probe)
```

|-HTimeProbeList TIMER - list of time probe objects

|-HTimerAbs TIMER - abstract representation of callback timer object

| HTimerEvent TIMER - event firing timer

| HTimerInterval TIMER - interval firing timer

| HNoidConnTimr NAMESERVER - timer for noid connections

| HTimeProbeTimer TIMER - interupt timer for time probe instance

|-HUserOut - presents diag messages to the user

|-HVIter - iteration across void pointer collections

| HVListIter - iteration across void pointer list collection

| HVListForIter - forward iteration across void pointer list collection

| HVListRevIter - reverse iteration across void pointer list collection

|-HVIterator - iteration across void pointer collections

pointer lists

|-HVect2 - two dimensional vector

|-HVect3 - three dimensional vector

|-HVect4 - four dimensional vector

|-HVectN - N dimensional vector

|-HVector - three dimensional vector

|-HVwDepsAbs - abstract model/view dependencies

| HVwDeps - model/view dependencies

```
|-HVwDepsNd - node in model/view dependency list

|-HWeekDay - elapsed week days

+-HYear - elapsed years

HDAG - directed acyclic graph (collection of HDAGNodes)

HMemDbg - monitors memory leaks

ElapTimeTimer - stop-watch like elapsed timer

HClassDict - dictionary of HClassDesc information

HObjSetNode - node of object set collection
```

Figure 3 - Hughes Class Library and Interprocess Communication Extensions

---

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

RTTI

HCL provides RTTI and object I/O streaming through the HClassDesc object in the same way that NIHCL uses the Class object. The relationship between an HObject (any class which is part of the HObject tree) and its static member HClassDesc is the same as the relationship between NIHCLís Object and Class classes. Every HObject has a static member of type HClassDesc. The HClassDesc contains the meta information of the respective class and has member functions to create a new object instance of the respective class. In NIHCL, the Class objects contained the run-time meta information representing inheritance relationships with pointers/arrays internally in each Class object. Instead, the HClassDesc objects in HCL have a static member pointer to a global class dictionary object (HClassDict). This class dictionary object encapsulates a data structure containing an image of the run-time utilization of all HObjects.

HObjects are identified by a string representation of their names using the HString class. The HClassDesc object has a member of type HString to store the Class name. For instance, when HObjects are serialized on a disk file or sent across a network over a byte stream communication mechanism the objectís type string representation is sent first. In this way, the consumer of the disk file or receiver on the end of the network connection will read the objectís string name first and ask the class dictionary to create an instance of the object from the string name. In this way, objects are instanciated with out ìtypeî knowledge; of course the ìtypeî information is abstracted into the HClassDict container object. Nevertheless, when using the HCL framework, the programmerís can rely on dynamic object creation using ìstringifiedî HObject names.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Object I/O

HObject has a much better foundation for object I/O. In addition to member functions which provide object I/O using ìistreamî and ìostreamî objects, HObject has an interface to a special class called XDR which encapsulates the external data representation (xdr) system for dispatching data on heterogeneous networks. The XDR class and HObject class allow object I/O between hosts which have different implementations of binary data on their hardware. For instance, two machines which store binary data in Big Endian and Little Endian respectively can still transport object between them with XDR and HCL.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Containers

HObject does not have a built in interface for ìdeep copyingî (see section 2.6.5) objects like NIHCL does. If desired, deep copying is left up to the cloning and object I/O implementations of individual HObjects.

HCL employs a very robust hierarchy of container classes and iterators. HCL abstracts the interface of container classes and iterators in HCollection abstract class. Both HObject typed collection classes (HObjCollection) and void typed collection classes (HVCollection) are supported as specialized HCollection classes. This allows not only HObject classes but also standard ëCí structures in HCollection classes.

```
/*************************************************************************

** Copyright (c) 1991 Hughes Aircraft Company.

** UNPUBLISHED WORK

*************************************************************************

class HObject {

protected:

// Constructor is protected since HObject is an abstract class.

HObject();

// _castDown is the function that actually check class

// descriptions. This is usually provided in the single

// inheritance case by the DECLARE/DEFINE macros.

virtual void* _castDown(const HClassDesc&) const;

public:
```

```
virtual ~HObject();

// All classes have a HClassDesc

// HAbsClassDesc.

static HAbsClassDesc myClass;

// Run time class information.

static HClassDesc& classDesc();

static HObject& castDown(HObject&);

static HObject* castDown(HObject*);

virtual void* objCastDown(const HClassDesc& objClassDesc() const;

virtual HObject* hclone();

virtual int isInheritedFrom(const HClassDesc&) const;

// Support for models and views.

virtual int changed(HObject *model, const char *token);

// Support for dynamic creation of HObjects. When storing an object

// on a stream, first the className is put and then the instance data.

// Create a new HObject from the stream.

static HObject* newObj(istream&);

static HObject* newObj(XDR*);

// Store instance data on an ostream

virtual int put(ostream&) const;

// Get instance data from an istream
```

```
virtual int get(istream&);

// ENCODE/DECODE instance data to/from an xdr stream.

virtual int xdr(XDR*);

// Store class name and then call put

virtual int store(XDR*) const;

// Read class name and then call get

virtual int read(istream&);

// Read class name and then call xdr

virtual int read(XDR*);

// Put to cerr - mostly for debugging

void dump() const;

// Comparison: returns 0/1 and checks HObject identity

virtual int isSame(const HObject&) const

// Comparison: returns 0/1 and checks HObject data

virtual int isEqual(const HObject&) const;

// Comparison: returns -1/0/1 and checks HObject dta

virtual int compare(const HObject&) const;

// Comparison: hashes on HObject data - two objects with

// the same data should have the same hash

virtual unsigned hash() const;
```

```
};
```

Code Block 8 - Hughes Class Library (HCL) Root Class HObject

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Using HCL

HCL provides many useful macros to help the programmer adhere to the common framework. The style of these macros is the same as those found in NIHCL. These macros are used in the declaration, definition, and data definition of new HObjects. Many different categories of new HObject -like classes can be formed with the macros. New classes can be abstract or concrete, single or multiple inheritance, distributed or local. The following code block is taken from the ìHClassDefines.h" header file.

```
/*************************************************************************

** Copyright (c) 1992 Hughes Aircraft Company.

** UNPUBLISHED WORK

*************************************************************************/

// To add a simple class d with base b

// add DELARE_CLASS(d) as first line of class definition

// add DEFINE_CLASS(d,b) in one source file

...

#define _DECLARE_CLASS(classname) \

private: \

static HClassDesc myClass; \

static HObject* makeClass();

...

#define _DECLARE_CLONE \

public: \
```

```
virtual HObject* hclone();

...

#define _DECLARE_CASTDOWN(classname) \

protected: \

virtual void* _castDown(const HClassDesc&) const; \

public: \

static HClassDesc& classDesc(); \

virtual HClassDesc& objClassDesc() const; \

static classname& castDown(HObject&p) \

{return *(classname*)(&p ? p.objCastDown(classname::classDesc()) :0);} \

static classname* castDown(HObject *p) \

{return (classname*)(p ? p->objCastDown(classname::classDesc()) :0);}

...

#define DECLARE_CLASS(classname) \

_DECLARE_CLONE \

_DECLARE_CLASS(classname) \

_DECLARE_CASTDOWN(classname)
```

Code Block 9 - Hughes Class Library (HCL) Framework Declaration Macros

```
#define _DEFINE_DEFAULT_CLONE(classname) \

HObject* classname::hclone() {return new classname(*this); }

...
```

```
#define _DEFINE_DEFAULT_CREATION(clasname) \

HObject* classname::makeClass() { \

classname *s = new classname; \

return HObject::castDown((HObject*) s); }

...

#define _DEFINE_CLASS_DATA(classname) \

HClassDesc classname::myClass(#classname,classname::makeClass);

...

#define _DEFINE_CASTDOWN(classname) \

HClassDesc& classname::classDesc() { return myClas; } \

HClassDesc& classname::objClassDesc() const (return myClass; }

...

#define _DEFINE_CAST_FCN(classname, baseclass) \

void* classname::_castDown(const HClassDesc &cl) const { \

if (cl == myClass) return (void*) this; \

else return baseclass::_castDown(cl); }

...

#define DEFINE_CLASS(classname, baseclas) \

_DEFINE_CLONE(classname) \

_DEFINE_DEFAULT_CREATION(classname) \
```

```
_DEFINE_CLASS_DATA(classname) \

_DEFINE_CASTDOWN(classname) \

_DEFINE_CAST_FCN(classname,baseclass)
```

Code Block 10 - Hughes Class Library (HCL) Framework Class Definition Macros

---

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Example

Consider the class HPoint below. Notice that HPoint is very close to the definition of the Point class shown in Code Block 4.

```
/*****************************************************************

** Copyright (c) 1992 Hughes Aircraft Company.

** UNPUBLISHED WORK

*****************************************************************/

class HPoint : public HObject {

DELARE_CLASS(HPoint);

private:

unsigned long myX;

unsigned long myY;

public:

HPoint();

HPoint(unsigned long, unsigned long);

HPoint(const HPoint&);

~HPoint();

unsigned long x() const;

unsigned long y() const;
```

```
void x(unsigned long);

void y(unsigned long);

int put(ostream&) const;

int get(istream&);

int xdr(XDR *);

};
```

Code Block 11 - Hughes Class Library (HCL) Example HPoint Class Declaration

```
/*****************************************************************

** Copyright (c) 1992 Hughes Aircraft Company.

** UNPUBLISHED WORK

*****************************************************************/

#include ìHPoint.hî

DEFINE_CLASS(HPoint, HObject);

HPoint::HPoint() : myX(0),myY(0)

{ DBG_MEM_CONSTRUCTOR(HPoint.1); }

HPoint::HPoint(unsigned long aX, unsigned long aY) : myX(aX),myY(aY)

{ DBG_MEM_CONSTRUCTOR(HPoint.2); }

HPoint::~HPoint()

{ DBG_MEM_DESTRUCTOR(HPoint); }

int HPoint::put(ostream &str) const

{ str << ì[ì << myX << ì, ì << myY << ì]î << endl;
```

```
if (!str) return FALSE;

return TRUE; }

int HPoint::get(istream &str)

{ char c; str >> c; if (!str) return FALSE;

str >> myX; if (!str) return FALSE;

str >> c; if (!str) return FALSE;

str >> myY; if (!str) return FALSE;

str >> c; if (!str) return FALSE;

return TRUE; }

int HPoint::xdr(XDR* xdrs)

{ return xdr_long(xdrs, &myX) && xdr_long(xdrs, &myY); }
```

Code Block 12 - Hughes Class Library (HCL) Example HPoint Class Definition

---

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Distributed Programming with HCL Interprocess Communication

In addition to the object I/O capabilities of HObject, the HIPC libraries form an environment which allows peer-to-peer communications between concurrent processes on local or remote hosts in a heterogeneous network. Using a message passing architecture HIPC provides transient and persistent connections between processes, asynchronous and synchronous messages, and remote-procedure-call (RPC) like designs.

Message, Agents, Address, and Connections

The are four (4) specialized HObject classes responsible for communicating objects/messages: HMsgConn, HAddress, HMessage, and HMsgAgent. HMsgConn class abstracts the concept of a connection between two (2) end-points. Processes can communicate if there is a valid HMsgConn tied to both ends. The end-points are abstracted in the HAddress class. This class encapsulates the host name, IP address, port number, service name, etc. of the end point. Information is passed between HAddress objects over a HMsgConn object using abstract HMessage class. Even though any HObject could be sent over a network connection, HIPC designers imposed the notion of a specialized message object to encapsulate the details of HIPC. If there was not a HMessage class then the HObject class would be ìheavierî with more detailed HIPC material. At the lowest level the object I/O interfaces of HObject are used to marshal HMessage objects over the XDR stream. The HIPC designers wanted to decouple the interface to HIPC from HObject. The management of HAddress and HMsgConn objects and the dispatching of asynchronously received HMessage objects is performed by the HMsgAgent class.

A process can have many HMsgAgent objects; each with a HAddress associated with it. HMsgAgents manage connections with other HMsgAgents on a local or remote host. All message passing is done asynchronously using notifier systems available from the operating system. HIPC is capable of using the SunSoft Notifier system, the X notifier, or a generic reactor/notifier pattern built with standard poll/select system calls with sockets. Connections between agents can be permanent or temporary; temporary connections are perfect for event notification and permanent connections are well suited for streaming of data between agents.

The four main HIPC classes introduced above have very detailed private implementations using operating system services and other HIPC classes which encapsulate more complicated mechanisms like high resolution timers and sockets. The public interfaces are straight forward and are somewhat deceptive in that the hide such powerful UNIX network programming techniques.

Connection End-point Determination

The reason why HCL/HIPC is industrial strength is how it handles the end-point determination problem. As long as a HMsgAgent ìknowsî the HAddress of another HMsgAgent then communication is simple. The HMsgAgent can ask the HAddress object to create a connection and send a HMessage across in one member function call. However, if the HAddress is not known then the situation becomes complicated. The HIPC designers offer three (3) solutions to this problem and they all work well according to there constraints:

Pre-allocate the end-point/location information before running/compiling the system. This information could be hard coded or configured at run-time through command line arguments or configuration files. In many cases this is an acceptable solution for a small distributed system that does not change its configuration often.

Deploy a specialized HMsgAgent in one or more processes in a network which all HMsgAgents know how to contact (through a well-known service port maybe). In this way a HMsgAgent could register itself with this well-known ìNameServerî and supply logical information about the services/objects which it manages. In this way, a interested HMsgAgent could query the NameServer and receive location/address information for desired agents and then connect to them. This level of indirection between communicating agents is very powerful for it allows systems to configured differently at run-time. Also, a more intelligent NameServer can be used to allow HMsgAgents and there associated process/objects to re-locate during run-time for purposes of load-balancing and fail-over. In fact, redundant/duplicate HMsgAgents/objects could be deployed using this architecture.

Similar to the previous option, the HMsgAgents could register in a nameservice provided by a standard product like the OSI X.500 directory or DCE.

On inspection of the inheritance diagram in , one sees that there are already many specialized HMessage classes supplied by HIPC. Actually, these HMessages classes are used by the internals of the HIPC library to manage the connections, address, and agents. For instance, there is a specialized HMessage class called HMsgShutdown which is sent by a HMsgAgent to another to notify that agent to shutdown. There is a generic text message class called HMsgText for sending status messages. There is a abstract HMsgData class which has specialized classes for sending various forms of information. The HMsgCol class is very powerful. This class encapsulated the capability to transport any object of derived from HCollection. In this way, all HCL container classes can be transferred to another process.

When developing a distributed application with HCL/HIPC you have to create specialized objects from HObject and also enhance the agent object with specialized message(s) objects associated with you new HObject class. As we will see in section 3, on CORBA, classes are specified in Interface Definition Language (IDL) and a pre-compiler creates the actual implementation class and the message/dispatching class(es) for you.

The details of the message object I/O use the object I/O interface of HObject. When a message is marshaled on to a XDR stream, three (3) element are streamed; the message header containing the

ìstringifiedî name of the HMessage object in xdr string format, the message body according to the virtual xdr() method of the HMessage class, and a xdr end-of-record marker. The receiving process is notified by the operating system through an asynchronous I/O system and the respective HMsgAgent handles the receipt of the HMessage. The agent reads the ìstringifiedî name of the HMessage and uses the HObject/HClassDesc/HClassDict system to create a new instance of the specified HMessage object. The agent then hands off control to the new HMessage object and instructs it to marshal its own data off the xdr stream by calling the HMessage::xdr() method. Then, according to the type of HMessage, the agent dispatches the information to the appropriate internal object in the process.

Example

The following is an example taken from Hughes Aircraft internal white papers [14],:

```
/*****************************************************************

** Copyright (c) 1992 Hughes Aircraft Company.

** UNPUBLISHED WORK

*****************************************************************/

#include ìHMessage.hî

class HMsgDoSomething : public HMessage {

DELARE_CLASS(HMsgDoSomething);

public:

HMsgDoSomething();

~HMsgDoSomething();

// over ride from virtual HObject::xdr(XDR*)

int xdr(XDR* stream);

private:

int myIntValue;
```

```
HString myStringValue;

}

/*****************************************************************

** Copyright (c) 1992 Hughes Aircraft Company.

** UNPUBLISHED WORK

*****************************************************************/

#include ìHMsgDoSomething.hî

DEFINE_CLASS(HMsgDoSomething, HMessage);

...

int HMsgDoSomething::xdr(XDR* stream)

{ if(!HMessage::xdr(stream) || // call base class behavior

!xdr_int(stream,&myIntValue) || // xdr filter on C++ data type

!myStringValue.xdr(stream)) // ask object to xdr itself

return FALSE;

return TRUE;

}
```

Code Block 13 - Hughes Class Library (HCL) Example HIPC Class

At a minimum, a HAddress and a HMessage is needed to send a information. A HMessage has an interface to send itself given a HAddress or HMsgConn as an argument. A HMsgConn or HAddress each have interfaces to send a supplied HMessage. As long as the HMsgAgent is in a notification/listening mode on the other end, the message will be sent.

HCL is an evolution of NIHCL and extends the object I/O solution to real world operating systems like Solaris, VMS, AIX, and IRIX. HCL is an excellent example of the single root tree design pattern and HCL provided a solution which allow for development of all aspects of distributed applications including

the user interface, interprocess message system, and internal implementations. HCL proves that a common object framework can be used to build such systems without having to integrate separate libraries for application development, network communication, and user interface.

The HCL framework can be used for general C++ application development using the robust container classes and the objects specific to temporal processing. Also, many advanced features can be employed by the HCL user because of the capabilities of the single rooted tree design based on `HObject`. However, it is the message agent services of the HIPC extensions which make HCL well suited for distributed object implementations. The following section presents NetClasses, the distributed object system from PostModern which is the foundation for their CORBA compliant commercially available product ORBeline. NetClasses and ORBeline provide distributed object services; components for general application development are not integrated in the framework. In fact, I have not found many systems which provide components for all aspects of distributed object application development. I usually find general frameworks or distributed object frameworks, but never both. In some cases, general framework vendors and ORB vendors integrate their libraries into a more efficient package which offers cross-platform GUI development, general application components, and distributed object services. HCL provides such a solution.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

PostModern's NetClasses

NetClasses is a framework for the development of distributed object system applications. ORBeline, PostModern Computing (now Visigenic)'s CORBA compliant product, is built on the foundation of the NetClasses framework. There is no documentation that explicitly asserts this but I have found evidence to support the latter statement. In my review of the C++ header files that come with the ORBeline product I found references to the NetClasses material. Some of the lower level classes in the ORBeline framework are prefixed with "NC" and "DS" which represent components of "NetClasses" and "Distributed Services" respectively. Distributed Services [15] is the named subsystem of NetClasses which is responsible for the management of object dissemination and peer to peer connections - much like the Message Agent components of HCL.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Reuse

Another reason that PostModern Computing (now Visigenic) probably reused NetClasses for the ORB product is that it makes sense. In fact, reuse is practiced at PostModern Computing (now Visigenic) in earnest. The technical staff at PostModern Computing (now Visigenic) are up front about reuse practices. Both NIH Class Library design patterns and the dispatching code from the object oriented X-Windows toolkit, Interviews, have been reused in PostModern Computing (now Visigenic)'s products. The following sections will reflect this reuse, especially the reuse of the single rooted tree technique.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Features

Before looking at the details of NetClasses, the high level design goals of the NetClasses product should be reviewed. NetClasses provides more than just Object Transport (PostModern Computing (now Visigenic)'s term for object I/O). NetClasses provides for, as stated above, management of object services and connections through the Distributed Services subsystem. Also, NetClasses features "Remote Method Invocation (RMI)" and "Security". RMI and security features are based on Distributed Services.

Distributed Services

The Distributed Services feature of NetClasses provide the user an abstraction to the service ports paradigm for socket and remote procedure call programming. A service agent design pattern very much like the message agent of HCL provides this abstraction. Objects on a network provide services by advertising to agents processes who monitor the service providers and service consumers. The agent is also the location for connection management. PostModern Computing (now Visigenic)'s technology overview document of NetClasses presents the capabilities of Distributed Services and is an excellent statement in general for requirements of any distributed object system:

Service Registration. What network services are available, and which machines on the network are providing them?

Connection Management. How are service providers connected to service consumers? Which consumers are connected to which servers?

Fault Tolerance Strategy. When a server or client fails, what service reconnection and fault tolerance strategies are in effect?

Load Balancing. How are servers loaded? Do we need to reconnect some client and servers dynamically to even out machines loads?

Service Location/Location Transparency. How does a client with no knowledge of current servers status locate the services it requires?

Remote Method Invocation

Remote Method Invocation (RMI) is implemented on top of Distributed Services. Through RMI, a client with an object reference to an external service (in a separate process on a local or remote machine) can invoke an object member function using the object reference. This is the abstraction of the classic remote

procedure call (RPC) and socket programming paradigms which have complicated interfaces and are difficult to maintain and scale. The RMI feature makes the fault tolerance and connection management of Distributed Services transparent to the programmer.

Security

The Security feature of NetClasses provides programmers with the means to implement encryption/decryption for object transport, and method invocation. Also, authentication of consumer accesses to remote service objects is provided and aided by this feature.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Design

The following presentation of the design of the C++ objects used in NetClasses, which are also the foundation for PostModern Computing (now Visigenic)'s CORBA compliant product ORBeline, is limited by two factors. First, the only information available are the C++ header files for these classes provided in the publicly available ORBeline distribution. The C++ header files represent the interface to the classes and provide only an estimate of the implementation hidden in the binary library. Obviously, PostModern Computing (now Visigenic) only supplies the header files to the user because the C++ source files reveal confidential, proprietary, and trade secret material. The header files are necessary for the programmer to access the binary libraries. However, the header files also reveal a lot of confidential material. A copyright and disclosure statement is a part of all header files in the ORBeline distribution.[16] In the same way as I presented HCL, I will use common sense when showing the following material in such a way as to promote the advanced design patterns and techniques found in PostModern Computing (now Visigenic)'s product without violating the copyright statement. The following material is my interpretation of the C++ header files which anyone can download from the Internet anyway.

Single Rooted Tree

The inheritance diagram of all the C++ objects for the ORBeline product can be found in Appendix A (page 2). Many of the classes in the diagram are named with the prefix "CORBA::". This prefix denotes classes which are part of the "CORBA" namespace using the scope delimiter operator of C++. [17] Notice that all the classes which are part of the CORBA namespace inherit from a single class called DSResource. The DSResource class denotes the Distributed Service Resource class of the NetClasses framework. The DSResource class represents the root class of the single rooted tree pattern. DSResource is the equivalent to the Object class in NIHCL and the HObject class in HCL. The DSResource class provides a common interface for all class instances which are to act a "distributable" objects. The DSReource class provides the primitive features for such objects. (See Appendix A, page 4)

DSResource features and interfaces are composed of its own members and those inherited from its multiple parent classes NCResource and NCObject. The NCResource and NCObject classes are low level NetClasses components. The NCResource class supplies a reference counting capability. The NCObject class supplies the RTTI, object I/O, and other capabilities similar to Object and HObject.

NCResource

The NCResource class encapsulates the concept of reference counting for objects which are shared.

Reference counting is a form of garbage collection which in most likely a reuse of the reference counting component of the object -oriented X-Windows toolkit, Interviews. Because the NCResource class only does reference counting, the class' interface is very simple. A reference counted object has a private/hidden member which keeps track of the number of references to itself which are held by other entities or objects. For NCResource, this private member is the _ref_count integer value. (See Appendix A, page 4)

When a pointer to an object, which inherits from NCResource, is added to a linked list, set, or any collection, the object which is pointed engages its "reference" behavior. The "reference" behavior simply increments the private reference count of the object. Objects inheriting from NCResource record the number of entities currently consuming or referencing the object. When the consuming entity is finished with the referenced object it calls the "unreference" behavior or the referenced object. The "unreference" behavior decrements the private reference count of the object. Objects which inherit from NCResource are never explicitly deleted by another entity using the standard destructor behavior. The object will delete itself when its reference count reaches zero. The reference count is initialized to one (1) when the object is constructed. The reference counting capability encapsulated by the NCResource class provides referenced objects with the self discipline to self destruct when they are no longer required. The DSResource class inherits from NCResource to add the referencing counting capability. See Code Block 14.

```
/*****************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/*****************************************************************/

class NCResource {

private:

unsigned _ref_count;

public:

NCResource() { _ref_count = 0; }

virtual ~NCResource();
```

```
NCResource(const NCResource& ) { _ref_count = 0; }

void operator=(const NCResource& ) { _ref_count = 0; }

static void ref(NCResource *obj);

static void unref(NCResource *obj);

void ref() { _ref_count++; }

void unref();

unsigned numRefs() const { return _ref_count; }

};
```

Code Block 14 - PostModern Computing (now Visigenic)'s NCResource Class

There are five important interfaces to NCResource. A NCResource object can be referenced by calling the static NCResource::ref() function with a pointer to itself or calling the object's ref() member function with no argument. A similar interface is provided for unreferencing NCResource objects. Also, the private reference count can be obtained by calling the NCResource object's numRefs() member function.

NCObject

The other class which DSResource inherits from is the NCObject class. The NCObject class provides most of the capability for object I/O, RTTI, and heterogeneous collections to the DSResource class. (See Appendix A, page 4) The NCObject declaration demonstrates a different strategy for the single rooted tree pattern. In the cases of NIHCL and HCL, all the common, primitive capabilities of the framework are encapsulated in one class; Object and HObject respectively. The NetClasses framework uses composition and inheritance techniques to complete the common framework provided by the DSResource class. DSResource takes its reference counting capability from NCResource. The rest of the capabilities are inherited from NCObject. NCObject forms its capabilities in three (3) ways: inheriting the object I/O interface from NCTransObject, the RTTI components by composition of NCTypeInfo, and adding its own behaviors for transportable, heterogeneous collections. The following two sections present NCTransObject and NCTypeInfo.

NCTransObject

The NCTransObject is an abstract class which encapsulates object I/O and provides the abstract interfaces for RTTI. (See Appendix A, page 4) Even though the NCTransObject does not have a data

member of type NCTypeInfo, interfaces are provided for accessing class information, classInfo(), and verifying safe cast requests with hasBase() and canCast().

In the case of HCL, the HObject class provided interfaces for reading and writing to I/O stream objects and the special XDR class. NetClasses, however, employs a more detailed object I/O scheme. It is important to reiterate that the purpose of the NetClasses framework is to provide distributed object only. Because it is not a general framework, more detailed is placed on the object I/O implementation.

The NCTransObject has a tightly coupled association with two (2) I/O classes: NCistream and NCostream. These stream classes decouple the NCTransObject from the standard iostream classes istream and ostream. Also, the NCistream and NCostream each have a helper class for conducting object I/O: NCInTbl and NCOutTbl respectively.

There are four (4) member functions for object I/O in NCTranObject which declare the abstract interface for the framework - two functions for input and two for output. First, member functions for reading and writing to standard iostreams are provided using readFromStream() and printOn() respectively. The arguments to these two functions are references to the standard iostream objects. Object output using the NCostream class is accessed with the write() member function. Object input using the NCistream class is accessed with the operator>>() member function which overloads the >> operator. Refer to Code Block 15. This member function's inline implementation shows that the real work for object input is done by the NCistream class. The implementation calls the readTransObj() member function of NCistream to perform the object input and return a pointer to the new NCTransObject. The NCistream and NCostream classes direct the real work of object I/O. For this reason, NCTransObject and the NetClasses I/O classes are tightly coupled. The reason for this design cannot be determined with access to more detailed implementation code.

```
/*************************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/*************************************************************************/

class NCTransObject {

public:

virtual void write(NCostream& strm) const =0;
```

```
virtual const NCTypeInfo* classInfo() const = 0;

virtual void printOn(ostream& strm=cout) const;

virtual void prettyPrint(NCostream& strm) const {

write(strm); }

virtual void readFromStream(istream& ) {}

inline friend NCistream& operator>>(NCistream& strm,NCTransObject*& obj){

obj = strm.readTransObj();

return strm; }

virtual ~NCTransObject() {}

NCTransObject() {}

NCTransObject(NCistream& strm) { strm.addToTbl(*this); }

int hasBase(const NCTypeInfo& p) const {

return ( classInfo()->hasBase(p) ); }

int canCast(const NCTypeInfo& p) const {

return ( classInfo() == &p || hasBase(p) ); }

};

ostream& operator<<(ostream& ostrm, const NCTransObject& obj);

istream& operator>>(istream& istrm, NCTransObject& obj);
```

Code Block 15 - PostModern Computing (now Visigenic)'s NCTransObject Class

NCTypeInfo

The NCTypeInfo class is very similar to the Class and HClassDesc classes of NIHCL and HCL.

NCTypeInfo encapsulates the RTTI for the framework. NCTypeInfo is similar in that it has a private character string for the class name , _name, and a private function which knows how to assemble an object of that type from a byte stream object, _read_func. (See Appendix A, page 4) Multiple inheritance is provided in the RTTI using NCTypeInfo's array of NCTypeInfo for the base class(es), _bases. See Code Block 16.

Even though a user would not normally use the NCTypeInfo interface, a public interface is provided for use in the low level framework classes and macros. The character string name of the class can be accessed using the classname() member function. A pointer to the function which can read the object off a byte stream can be accessed with the readFunc() member function. The safe casting capability is provided with the hasBase() and canCast() member functions. A boolean response is returned from these functions when the NCTypeInfo argument is compared to the private type information. The NCObject class and some detailed preprocessor macros provides the real user interface to the RTTI.

The most powerful methods of NCTypeInfo are the static methods for determining NCTypeInfo, NCTypeInfo::typeInfo() and read function from a character string argument, NCTypeInfo::readFunc(). It is these static or global methods which allow objects to be constructed from a byte stream without type information. When an object is marshaled to a byte stream the object class name is marshaled before the instance information. In this way, the reader of the byte stream can call NCTypeInfo::typeInfo() static function with the character from the stream as an argument. The type information is determined from the character string class name and the proper read function can be engaged to unmarshal the object instance data from the byte stream. Again, this is the same pattern used in NIHCL and HCL.

```
/*************************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/*************************************************************************/

typedef NCTransObject *( *NCTransReadFunc)(NCistream& );

class NCTypeInfo {

private:

const char *_name;
```

```
NCTransReadFunc _read_func;

short _num_bases;

NCTypeInfo **_bases;

NCTypeInfo(const NCTypeInfo& );

NCTypeInfo& operator=(const NCTypeInfo&);

public:

NCTypeInfo(const char *name,NCTransReadFunc readfunc,...);

virtual ~NCTypeInfo();

int operator==(const NCTypeInfo& p) const {

return (_name == p._name); }

int operator!=(const NCTypeInfo& p) const {

return (! (*this == p) ); }

const char *className() const { return _name; }

NCTransReadFunc readFunc() const { return _read_func; }

int hasBase(const NCTypeInfo& p) const;

int canCast(const NCTypeInfo& p) const {

return ( this == &p || hasBase(p) ); }

static const NCTypeInfo *typeInfo(const char *name);

static NCTransReadFunc readFunc(const char *name);

};
```

Code Block 16 - PostModern Computing (now Visigenic)'s NCTypeInfo Class

```
/*************************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/*************************************************************************/

class NCObject : public NCTransObject {

private:

static NCTypeInfo class_info;

protected:

NCObject();

public:

virtual ~NCObject() {}

static const NCTypeInfo *desc();

NCObject(NCistream& strm);

inline friend NCistream& operator>>(NCistream& strm, NCObject*& obj) {

obj = (NCObject *)strm.readTransObj();

return strm; }

static NCObject *readFrom(NCistream& strm) {

return (NCObject *)strm.readTransObj(); }

virtual const NCTypeInfo *classInfo() const;
```

```
virtual unsigned hash() const = 0;

virtual int compare(const NCObject& ) const =0;

virtual NCbool isEqual(const NCObject& p) const {

return ( (classInfo() == p.classInfo()) && compare(p) == 0 ); }

virtual NCbool isSame(const NCObject& p) const {

return ( this == &p ); }

const char *className() const {

return classInfo()->className(); }

NCbool isMemberOf(const NCTypeInfo& info) const {

return (classInfo() == &info); }

NCbool isMemberOf(const NCObject& p) const {

return (classInfo() == p.classInfo() ); }

NCbool isKindOf(const NCTypeInfo& p) const {

return ( classInfo()->hasBase(p)); }

NCbool isKindOf(const NCTypeInfo *p) const {

return ( classInfo()->hasBase(*p)); }

virtual NCbool operator==(const NCObject& p) const {

return isEqual(p); }

virtual NCbool operator!=(const NCObject& p) const {

return (!isEqual(p) ); }

};
```

Code Block 17 - PostModern Computing (now Visigenic)'s NCObject Class

The NCObject class builds on the abstract interface of NCTransObject to include behavior that allow classes derived from NCObject to be placed in collection classes. The components of Object from NIHCL and HObject from HCL which provide methods for comparing ject class. The RTTI class NCTypeInfo is a static data member of the NCObject class. A class derived from NCObject will have one (1) copy of a NCTypeInfo Object pre class. The private data member class_info can be accessed publicly by the static class function X::desc() where X is a class derived from NCTypeObject. The NCTypeInfo for a particular class can also be accessed with the member function classInfo() and the character string class name can be accessed with the member function className().

NCObject builds on the lower level interface of the safe casting behavior, canCast() and hasBase(), with the following methods: isMemberOf() and isKindOf(). The isMemberOf() method takes a NCTypeInfo object reference as an argument and checks to see if the subject is the same class as the argument. The isKindOf() works the same way but returns TRUE if the subject is derived from the argument.

A complete set of preprocessor macros are provided with NetClasses to guarantee conformance to the common framework. (See Appendix A, page 6 &7) These macros a very similar to those found in NIHCL and HCL.

```
/*************************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/*************************************************************************/

#define DECLARE_CLASSINFO(classname) \

static NCTypeInfo class_info;

#define DECLARE_NCOBJECT_CLASS(classname) \

private: \

DECLARE_CLASSINFO(classname) \

public: \
```

```
static const NCTypeInfo *desc(); \
virtual const NCTypeInfo *classInfo() const; \
inline friend NCistream& operator>>(NCistream& strm,classname*& obj) { \
obj = (classname *)strm.readTransObj(); \
return strm; } \
virtual void write(NCostream& ) const; \
virtual void prettyPrint(NCostream& ) const; \
virtual unsigned hash() const; \
virtual int compare(const NCObject& ) const; \
static NCTransObject *readFrom(NCistream& strm); \
classname(NCistream& strm)
#define DEFINE_NCOBJECT_CLASS(classname) \
const NCTypeInfo *classname::classInfo() const \
{ return &classname::class_info; } \
const NCTypeInfo *classname::desc() { return &classname::class_info; } \
NCTransObject *classname::readFrom(NCistream& strm) \
{ return new classname(strm); } \
NCTypeInfo classname::class_info(STRINGIZE(classname), \
(NCTransReadFunc)&classname::readFrom, NCObject::desc(), 0)
```

Code Block 18 - PostModern Computing (now Visigenic)'s NetClasses NCObject Macros

DSResource

The ORBeline product uses the classes which all derive from DSResource. As stated before, DSResource inherits its capability from both the NCResource class and the NCObject class. The interface to the resource referencing capability from NCResource class is enhanced in DSResource. An object can be requested to create a new reference to itself - return a pointer to itself. This behavior is encapsulated in the public member function _duplicate() which calls the lower level ref() member function of the NCResource class to increment the reference count and return a pointer to itself. This is similar to the cloning capability found in NIHCL and HCL but no copy is made of the object; a copy is made of the object reference (pointer). There is no standard copy constructor or cloning mechanism for the DSResource class. A subsequent request to dismiss an object reference is done by calling the member function _release() on the object reference (pointer). The _release() member function simple calls the lower level unref() member function of the NCResource class to decrement the reference count and potentially self destruct the object.

In addition to the CORBA::X classes defined in the ORBeline product, there are some other Distributed Services classes which derive from DSResource. The DSCollection and DSAssoc classes inherit from DSResource and are fundamental components for other container classes. The abstract DSCollection class encapsulates the common functions for use by any collections of DSResource objects. There are three concrete collection classes which inherit from DSCollection. The DSAssoc class encapsulates the concept of key value pairs. DSAssoc objects are used in DSSet and DSDictionary classes.

```
/*************************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/*************************************************************************/

class DSResource: public NCObject, public NCResource {

private:

static NCTypeInfo class_info;

public:

static const NCTypeInfo *desc();
```

```
DSResource();

DSResource(NCistream& strm);

virtual ~DSResource();

inline friend NCistream& operator>>(NCistream& strm, DSResource*& obj) {

obj = (DSResource *)strm.readTransObj();

return strm; }

virtual const NCTypeInfo *classInfo() const;

DSResource *_duplicate() { ref(); return this; }

void _release() { unref(); }

};
```

Code Block 19 - PostModern Computing (now Visigenic)'s DSResource Class

---

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Using the NetClasses Framework

As stated above, the NetClasses framework is the foundation of the ORBeline product for reasons presented above. A detailed explanation of the use of the NetClasses and Distributed Services classes will be given in Section 3 on CORBA and ORBeline. For the purposes of being consistent with the previous two frameworks, NIHCL and HCL, an example will be presented by showing how a "Point" object would be formed using NetClasses. Even though this might not be consistent with how PostModern Computing (now Visigenic) intends their product to be used, a trivial example is consistent with this presentation on distributed object frameworks.

A user defined class would inherit publicly from DSResource and the NCObject macros would be used to supply common framework interfaces for RTTI and object I/O. The user specific interface and data members would be in addition to the macros.

```
class DSPoint : public DSResource {

DECLARE_NCOBJECT_CLASS(DSPoint);

private:

unsigned long myX;

unsigned long myY;

public:

DSPoint();

DSPoint(unsigned long,unsigned long);

DSPoint(const DSPoint&);

virtual ~DSPoint();

void copyFrom(NCistream&);

void get(NCistream& strm) { copyFrom(strm); }
```

```
void put(NCOstream& strm) const { write(strm); }

unsigned long x() const;

unsigned long y() const;

void x(unsigned long);

void y(unsigned long);

};
```

Code Block 20 - PostModern Computing (now Visigenic)'s NetClasses Example DSPoint Class Declaration

The above class declaration of DSPoint is almost identical to that of HPoint for HCL example and Point in the NIHCL example. The exact definition of the DSPoint class is not easily determined because implementation details of NetClasses is not supplied with the ORBeline distribution; this information is company proprietary. However, the Interface Definition Language (IDL) compiler which comes with the ORBeline product (as explained in Section 3.3.3.3) translates IDL into C++ header files and source files. These files declare and define the C++ classes used in the ORB. The ORBeline IDL compiler produces C++ code which adheres to the NetClasses framework. The C++ code is a mixture of ORBeline specific code and NetClasses code. The code output from the ORBeline compiler gives a very good indication as to the implementation details of the NetClasses discuss above. The following is a simple example of a possible definition of the DSPoint class. It is just an estimate but is actually is very similar to the definitions of HPoint from HCL and Point from NIHCL.

```
#include "DSPoint.h"

DEFINE_NCOBJECT_CLASS1(DSPoint,DSResource);

DSPoint::DSPoint(NCistream& strm) : DSResource(strm) {

strm >> myX;

strm >> myY;

}

void DSPoint::copyFrom(NCistream& strm) {
```

```
strm >> myX;

strm >> myY;

}

DSPoint::DSPoint(const DSPoint& obj) {

myX = obj.x();

myY = obj.y();

}

void DSPoint::operator=(const DSPoint& obj) {

myX = obj.x();

myY = obj.y();

}

void DSPoint::write(NCostream& strm) const {

strm << myX;

strm << myY;

}

DSPoint::DSPoint() {

myX = 0;

myY = 0;

}

DSPoint::~DSPoint() {

}
```

Code Block 21 - PostModern Computing (now Visigenic)'s NetClasses Example DSPoint Class Definition

---

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Success of the Single Rooted Tree Framework

The power of the above example is the demonstration of the obvious benefit of reuse, especially in the case of distributed object frameworks. Also, NetClasses proves the capability of the single rooted tree paradigm in that NetClasses and ORBeline are both industrial strength products.

It will be seen in the presentation on CORBA in Section 3 and ORBeline specifically, that PostModern Computing (now Visigenic)'s investment in the technology found in NetClasses based in part on NIHCL, created a strong foundation for the development of their CORBA 1.1 compliant product. Using a proven distributed object framework, this company was among the first to market with their ORB. It is important to note that a year after the first release of ORBeline, PostModern Computing (now Visigenic) was again first to market with their CORBA 2.0 compliant version of ORBeline. I think that the strong foundation of the NetClasses distributed object framework helped speed the evolution of the ORBeline product to version 2.0.

In the next section on the Adaptive Communication Environment (ACE) a different framework is presented that does not use the single rooted tree paradigm. The single rooted tree provides a viable solution for the requirement of object transport or object I/O. It will be shown that ACE satisfies a different set of requirements for a distributed framework.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Adaptive Communication Environment (!!w2wpACE!!http://www.cs.wustl.edu/~schmidt!pe!)

The Adaptive Communication Environment (ACE) is a C++ framework for the development of high speed, object-oriented, distributed applications such as concurrent network servers and gateways. The distinction between the goals of the ACE framework and the three presented in the previous sections is best shown by example.

The ACE framework is being used for the network management portion of the Motorola Iridium global mobile communication system. The high speed, distributed network servers built using ACE manage the connection and synchronization of services among world wide satellite ground stations. However, the implementation of a service in a thread of control in a particular server would probably be designed and coded from scratch or more than likely in another framework for general application development. ACE provides the building blocks for the servers and is not intended for the development of the services installed in the servers. The NIHCL or HCL frameworks are not optimized for high speed network server creation because they provide for general application development also. There is some overlap in the two frameworks with respect to interprocess communication schemes. The ultimate framework would combine the fast operating system wrappers and distributed service frameworks of ACE and the elegant distributed object patterns of NIHCL, HCL, and NetClasses.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Object Request Broker (ORB)

One of the driving goals for this research was to identify the best distributed object framework. The conclusion of this research is that the Object Request Broker (ORB) represents the future of distributed object systems. As stated before, ORBs provide the framework for object location, activation, and communication. 9 Standards for ORBs and commercially available products which deliver ORBs have been insufficient in the eyes of information system providers. However, the potential advantages of using ORBs for the development of the new generation of world wide, high performance, distributed systems reflects the wide spread investment in object systems. In other words, the use of ORBs today is not wide spread, but it is accepted that ORBs and ORB-like systems will become common place.

The Object Management Group (OMG) defines, in the Common ORB Architecture (CORBA) specification, that the ORB is a system which provides mechanisms for sending requests and receiving responses to and from objects. In addition, the ORB operates in a heterogeneous distributed environment and interconnects unrelated object systems. [18] The OMG specifies that CORBA is based on a concrete object model derived from the abstract OMG Object Model.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

OMG Concrete Object Model

The ***object model*** defines the terms and concepts used in the CORBA specification. The following is a list of terms and concepts as defined by the OMG.

- An ***object system*** is a collection of objects that isolate the requesters of services (clients) from the providers of services by a well-defined encapsulating interface.
- The ***classical object model*** represents the situation where a client sends a message to an object, the object interprets the message to decide what service to perform, the object determines both the method to call and the object responds to the client.
- A ***client*** of a service is any entity capable of requesting a service from an object in an object system.
- An ***object*** is an identifiable, encapsulated entity that provides one or more services that can be requested by a client.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Request

- A ***request*** for services is an event sent by a client to an object consisting of an operation, a target object, zero or more parameters, and an optional request context.
- A ***value*** is an instance of an Interface Definition Language (IDL) datatype that may be a legitimate parameter in a request.
- An ***object name*** is a value which identifies an object type in an object system.
- An ***object reference*** is an object name that reliably denotes a particular object; an object reference will identify the same object each time the reference is used in a request.
- A ***request context*** is a request parameter which provides state information about the associated request.
- An ***exception*** represents an abnormal condition occurring during the execution of a request.
- A ***request parameter*** may be one of three (3): ***input parameter***, ***output parameter***, or ***input-output parameter***.
- Objects are created or destroyed as an outcome of issuing requests in the object system.
- A ***type*** is signature which restricts all parameters or return values in the object system and requests.
- A ***value*** is an instance of a type.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Values

- There are three (3) value types: ***object references***, ***basic values***, and ***constructed values***.
- The ***basic values*** are more specifically of value type {Short | Long | UShort | ULong | Float | Double | Char | String | Boolean | Octet | Enum | Any }
- The ***Short*** and ***UShort*** values are 16-bit, 2's complement integers.
- The ***Long*** and ***ULong*** values are 32-bit, 2's complement integers.
- The ***Float*** and ***Double*** values are 32-bit and 64-bit IEEE floating point numbers.
- The ***Octet*** value is an 8-bit opaque datatype, guaranteed to not undergo any conversion during transfer between systems.
- The ***String*** value consists of a variable-length array of characters; the length of the string is available at run-time.
- The ***Any*** value can represent any possible basic or constructed type.

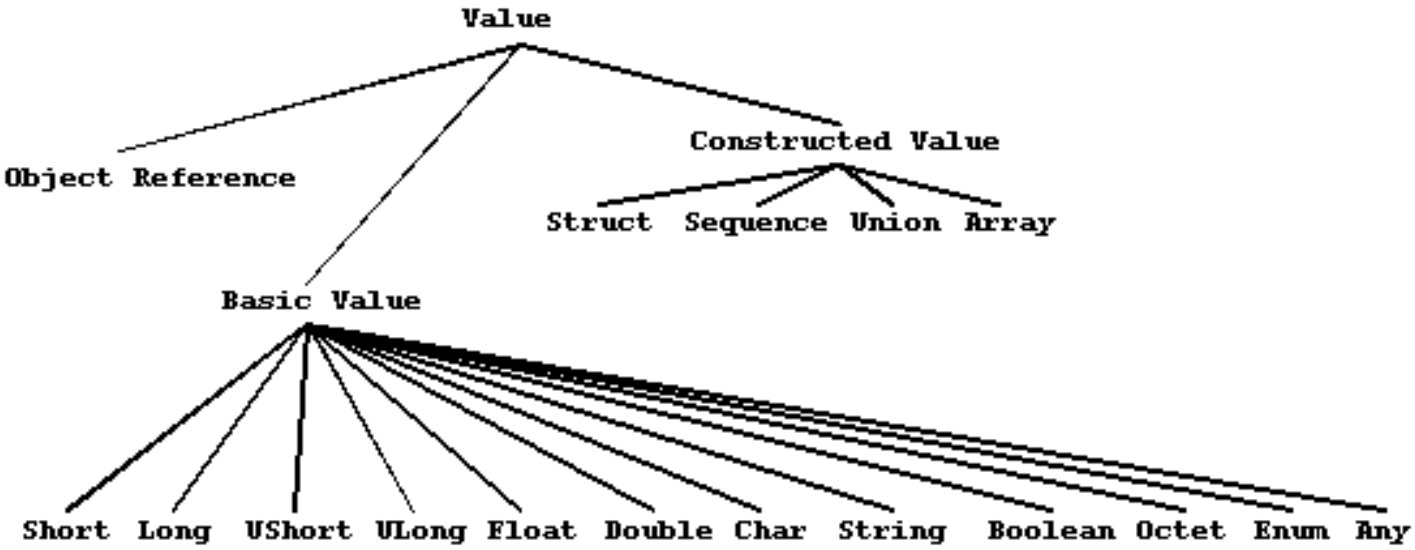


Figure 4 - OMG Specified Type Values

- The ***constructed values*** are more specifically of value type {Struct | Sequence | Union | Array } and represent user defined data structures through IDL.
- The ***Struct*** value is a record type consisting of an ordered set of (name, value) pairs.
- The ***Sequence*** value consists of a variable-length array of a single type; the length of the sequence is available at run-time.
- The ***Union*** value consists of a discriminator followed by an instance of a type appropriate to the discriminator value.
- The ***Array*** value consists of a fixed-length array of a single type.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Object Interface

- An ***interface*** is the description of the set of possible operations that a client may request of an object. The interface hides the implementation details of the object.
- An ***interface type*** is a type that is satisfied by any object that satisfies a particular interface. An ***interface type*** is any value that identifies an object.
- Interfaces are specified in ***Interface Definition Language (IDL)***. ***Interface inheritance*** provides the composition mechanism for permitting an object to support multiple interfaces.
- The ***principal interface*** is the most-specific interface that the object supports.
- An interface may have attributes. An attribute is logically equivalent to declaring a pair of accessor functions.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Object Operations

- An ***operation*** is an identifiable entity that denotes a service that can be requested.
- An operation is identified by an ***operation identifier***.
- An operation has a signature that describes the legitimate values of request parameters and returned results.
- An ***operation signature*** consists of the required parameters, the result, the exceptions that may be raised and the parameters, and contextual or execution semantics.
- The operation parameters are specified by their ***mode*** (IN, OUT, or INOUT) and type.
- The code that is executed to perform a service is called a ***method***.
- The execution of a method is called a ***method activation***.

When a client issues a request, a method of the target object is called. The input parameters passed by the requester are passed to the method and the output parameters and return value are passed back to the requester.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

The ORB Structure

In any discussion of ORBs, the term "client" represents the initiator of a request and the term "object" or "object implementation" represents the service being requested. However, this client/server paradigm only applies to the point of view of the request. It is the existence of a request which labels one entity as the client and the other as the server. In fact, a server object might also be the client of another server object while it is servicing a request.

The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the parameters making up the request. The interface the client sees is completely independent of where the object is located and how the object is implemented.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Static IDL ORB Interface

There are three (3) ways which the client communicates with the ORB. The client can directly interact with the ORB for administrative functions through the common ORB Interface. See Figure 5. The other two forms of client/ORB communication are used for request invocation. A specific, IDL defined interface to an object can be called through the IDL Stub to invoke a request. A request can also be invoked through the Dynamic Invocation interface (DII) for cases when the IDL stub is not present in the client (a request is made to an object interface for which no a priori knowledge of the interface through IDL is known) The object implementation (server) receives a request as an up-call through the IDL generated skeleton.

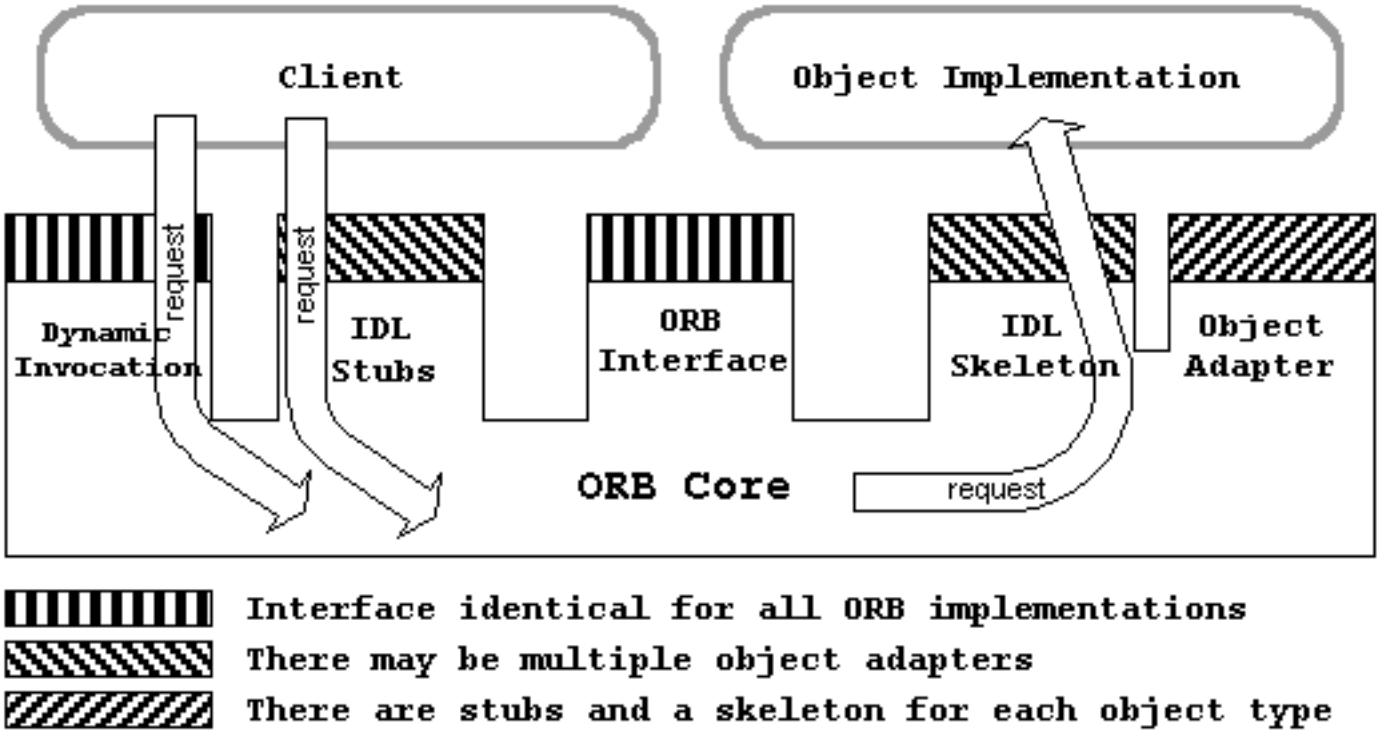


Figure 5 - OMG ORB Structure

The IDL is the language used to describe the interface to object implementations or services which a client object calls. In the ORB architecture, all communication is done through object interfaces (defined by IDL) and the end points of communication are the caller entity (client) and the object implementation. The client can invoke operations of a specific interface because the client has an object reference to the object implementation. An object reference represents the information needed to specifically an object within the ORB. The IDL which defines the object interface is used to construct the IDL stub and IDL

skeleton located in the client and server respectively. The client initiates a request by calling IDL stub routines that are specific to the object interface pointed to be the object reference.

After a request has been accepted by the ORB from the IDL stub or DII, the ORB locates the appropriate object implementation and transmits parameters and control to the object implementation through the IDL skeleton. See Figure 5. The object implementation services the request and the passes return parameters and control back to the client.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Language Mapping

The programmer interfaces to the ORB, DII, and object adapter are specified according to a language mapping. From a programmer's point of view, their programs must interface to the ORB in the same language that the program is written in (C, C++, SmallTalk, Java, etc.) Also, the portable IDL code is compiled into a programming language according to a CORBA specified language mapping. As an example, the same object interface can be used in a Java program and a C++ program because the interface is written in IDL and can then be compiled separately into Java and C++ stubs and skeletons. In fact, a client could call a specific object stub using SmallTalk and the stub converts the request into canonical form. The ORB then transmits the request to a IDL skeleton mapping to a C++ object implementation. The ORB, the ORB interfaces, and the IDL not only allow the object system to work properly in a heterogeneous hardware network but also in a heterogeneous coding environment.

Another important factor in having the ORB work in a heterogeneous environment is the object reference. Object references can be implemented according to any language mapping. Through common ORB functionality, object references can be converted to a string form which can be transmitted to another part of the object system which needs the object reference in an alternate language mapping.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Interface Repository (IR) and the Dynamic Invocation Interface (DII)

The interface repository (IR) is a run-time database of the IDL declarations. The interface repository allows a programmer to query an object interface in the repository to find out what types of operations and parameters are supported by the interface. Dynamic requests are built at run-time using the interface repository. A client which has an object reference for which no IDL stub is available can call the ORB's common `get_interface()` function to retrieve the `InterfaceDef` object from the interface repository. The `InterfaceDef` object is a standard type. Using the `InterfaceDef` object, other associated objects can be retrieved including `OperationDef` objects for the operations of the object interface. Even the exceptions associated with an operation can be retrieved from the `ExceptionDef` object.

The IR and DII provide the user/programmer interface to the low level implementation of the request protocol. All object interfaces in the ORB are unique, identifiable entities. Also, within an object interface, there are uniquely identifiable operations, parameters, and exceptions. When a request is issued, it is these low level unique IDs for the interfaces, operations, parameters, and exceptions which are assembled and sent to and from the ORB. The DII processes requests which are already in the low level form and have no relationship with the client or server code. Requests which originate in the IDL Stubs are formed at compile-time and are therefore known to the programmer and are represented as code in the language mapping.

The low level aspect of dynamic requests make them powerful but also very abstract. As an example, using a dynamic request is like using a protocol like TCP but accessing it at the IP level at one end of a connection stream. In this case, just like the case of DII, the other end of the TCP protocol connection doesn't know the difference. When a IDL skeleton receives a request, the origin of the request is not known. The request can come from a pre-compiled IDL stub in the client program or can be built at run-time using the DII and IR.

I find the IR and DII so abstract that I haven't found a use for it in any application program. When I find examples of applications using dynamic requests, the applications are always interacting with a human to create requests. For example , the IR could be used to query the interface of an object reference returned in a previous request. In this way, the IR could be used to find reusable software components. In another example, inspired during an open discussion of CORBA at Object Expo NY 1995, the IR and DII could be used to monitor and control abstract components in a large computer network. If every node in the network had a built in system management object, even an unrecognized node could be monitored by querying the IR with a reference to the system management object. Here again, human intervention is needed to interpret the IR query results.

For almost all cases, the standard IDL stubs provide the best interface for invoking requests. Even if code could be designed to use the IR and DII without human intervention, the performance would be degraded. In order to create a dynamic request, a static operation for all object references is provided to create a request object. The request object encapsulates the behavior of adding arguments and dispatching the request. As the name implies, the dynamic request is actually "built" during run-time. The IDL stub contains pre-compiled request which are obviously faster.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Implementation Repository

The OMG also specifies another ORB database called the implementation repository which stores the unique details the ORB uses to locate and activate object implementations. The implementation repository usually contains operating system and vendor specifics which are not portable. Installation of implementations and control of policies related to the activation and execution of object implementations is done through operations on the implementation repository. A client can call a common ORB function get_implementation() on an object reference to retrieve the implementation information.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Object Adapter

The Object Implementation communicates with the ORB through the common ORB Interface or the Object Adapter component. As an example, the Object might register its service(s) with the ORB through calls to the Object Adapter. Services provided by the ORB through an Object Adapter also include: generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, and mapping object references to implementations.

- A CORBA compliant ORB shall have at least on type of object adapter called the Basic Object Adapter (BOA). The requirements on the BOA are [19]:
- The BOA shall supply a Implementation Repository that lets the server install and register object implementations.
- The BOA shall provide mechanisms for generating and interpreting object references, activating and deactivating object implementations, and invoking methods and passing request parameters.
- The BOA shall provide an identification mechanism for clients. The client or "principal" identification is provided by the adapter but it is up to the implementation to use it.
- The BOA shall provide for method invocation through stubs.

The activation of object implementations is governed by four specified policies: **shared server**, **unshared server**, **server-per-method**, and **persistent server**.

Multiple objects reside in the same server program (process) in a shared server activation policy. Using the Implementation Repository, the BOA starts the server process following the first request. A server process notifies the BOA that it is ready by calling the common object adapter function `impl_is_ready()`. The server handles one request at a time. When the server is ready to terminate, it calls the common object adapter function `deactivate_impl()`. The unshared server activation policy provides for one object implementation per process. When the object implementation server process is ready it notifies the BOA by calling the common adapter function `obj_is_ready()`. The server object remains active and will receive requests until it calls the common object adapter function `deactivate_obj()`.

The server-per-method activation policy allows for a new server process to be started for every request made on an object implementation. The persistent server activation policy is for server objects which are started outside of the control of the BOA.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

CORBA Standard

The state of the art in CORBA compliant ORBs are those compliant with the new (late 95) CORBA 2.0 specification from the OMG. CORBA 2.0 addresses some of the short comings of the CORBA 1.1 specification such as standard inter-ORB compatibility. I found that most commercial CORBA 1.1 vendors provided extensions to the standard which made there ORB a viable product. In many case, CORBA 2.0 provided a standard for what ORB vendors were already doing but doing differently. For instance, CORBA 2.0 requires that TCP/IP be used for inter-ORB communication but most ORB vendors already used TCP/IP. The following section describes ORBeline, the CORBA 1.1 compliant ORB sold by PostModern Computing (now Visigenic) and Orbix, the CORBA 1.1 compliant ORB produced by Iona Technologies. PostModern Computing (now Visigenic) and Iona have provided their ORBs for distributed object research at Villanova University.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

PostModern Computing (now Visigenic)'s ORBeline

The ORBeline system from PostModern Computing (now Visigenic) is a complete CORBA 1.1 implementation with extensions added for fault-tolerance, distributed directory services, and automated protocol selection. ORBeline works on top of the proven distributed object system, NetClasses (see PostModern's NetClasses Section 2.8). This section describes the overall architecture of ORBeline, the implementation details and relationship to NetClasses, and some example IDL code.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

ORBeline's Architecture

The heart of ORBeline is the replicated dynamic directory service called the Smart Agent which keeps track of all objects in the ORB. The Smart Agent is a process running on a host in a network. Not only the activated objects but also the dormant object implementations registered with the object activation daemons are monitored by the Smart Agent. Upon the ORBeline ORB receiving a request from a client, the Smart Agent will either map the request to an activated object (already running) or a new server process will be started according to the object implementation details found in an associated object activation daemon.

In addition to the ORB objects, the Smart Agent keeps track of all active clients and their respective connections to servers in the ORB. The Smart Agent monitors these connections and the end-point processes using heart beats. A heart beat refers to operating system mechanisms which are used to determine the state of another process. If a connection is broken or an end point process is no longer running, the Smart Agent will facilitate a reconnect or the object server will be restarted. The ORBeline architecture is fault-tolerant because of the Smart Agent. ORBeline allows objects to be replicated so that the Smart Agent can reconnect a client to a duplicate object server when the first one is unavailable. For example, if a client is connected to an object running on host A and host A crashes, the Smart Agent will seamlessly reconnect the client to an object running on host B. The Smart Agent itself can be replicated on many hosts in a network. If the host running the primary Smart Agent crashes, a replicated Smart Agent will takeover on another host.

As mentioned above, the Smart Agent relies on object activation daemons to start object servers. Object implementation can be registered with an object activation daemon running on a particular host. The object activation daemon will be notified upon a client request to start the server object.

ORBeline provides a very elegant distributed object implementation. Object implementations can be anywhere in the ORB with respect to the client. The objects can be collocated in the same process or on the same machine as the client. The objects can also be on another machine in the same network or in a remote network. The Smart Agent will monitor other Smart Agents registered from other network. No matter where the requested object is located, this fact is totally transparent to the client. Again, the notion of a client/server relationship is from the point of view of the request endpoints. ORBeline's CORBA 1.1 compliance and fault tolerant extension operates on the strong foundation of the NetClasses product.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

The

ORBeline Framework

In addition to the executable objects provided with ORBeline such as the Smart Agent, IDL compiler, Object Activation Daemons, and the Interface Repository, PostModern created an object framework which supports the CORBA 1.1 specification. This object framework (now referred to as the ORBeline framework), brings the NetClasses framework up to compliance with CORBA 1.1 and completes a seamless C++ mapping of CORBA IDL. The ORBeline framework is a forest of classes where most of the classes are part of the tree inheriting from DSResource and the rest are supporting classes which complete the ORB interface. (See Appendix A, page 2) As presented in Section 2.8.3.2, the DSResource class is an abstract interface for distributed objects which provides reference counting, object I/O, RTTI, and other primitive behaviors necessary to implement CORBA objects. This section describes the details of the ORBeline framework including the configuration, the implementation of primitive ORB types, the exception handling system, and the ORB Object.

Namespaces and the

CORBA class

The ORBeline framework consists of many classes (including all those in the DSResource tree) which are part of the CORBA namespace. A namespace, in the context of C++, refers to the specification of the scope of a symbol in the code. Namespaces are part of the new ANSI/ISO standard but are not available in all compilers. Namespaces provide a almost perfect guard against symbol collisions. An example of a symbol or name collision is when two (2) object frameworks are being used at the same time and they both declare a type `boolean`. The ideal solution to this problem is to allow designers the freedom to use names which they deem appropriate and provide a name collision prevention mechanism. The ANSI/ISO standard specifies such a solution.

The ORBeline framework provides a protected namespace by declaring names within the bounds of the class `CORBA` (nested classes). For instance, ORBeline declares a class `Object` in a header file called `object.h`. The `object.h` header file is never explicitly included in a user file. Instead, the file `corba.h` declares the class `CORBA` which declares all the ORBeline classes, typedefs, statics, and includes the ORBeline header files inside the `CORBA` class declaration. (See code block below and Appendix A, page 3) The only header needed by a user file is `corba.h`.

```
/**************************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/**************************************************************************/

class CORBA

{

public:

class Object;

class Structure;

class Environment;

class Array;

class Sequence;

class String;

class Union;

class ORB;

class BOA;

class MarshalStream;

class Context;

class Property;
```

```
class PropertyList;

class TypeCode;

class StructTypeCode;

class EnumTypeCode;

class UnionTypeCode;

class StringTypeCode;

class ObjRefTypeCode;

class SequenceTypeCode;

class ArrayTypeCode;

class Any;

class Value;

class ValueShort;

class ValueLong;

class ValueUShort;

class ValueULong;

class ValueChar;

class ValueOctet;

class ValueBoolean;

class ValueString;

class ValueObjRef;

class ValueTypeCode;
```

```
class ValueStruct;

class ValueUnion;

class ValueEnum;

class ValueSequence;

class ValueArray;

typedef short Short;

typedef long Long;

typedef unsigned short UShort;

typedef unsigned long ULong;

typedef char Char;

typedef float Float;

typedef double Double;

typedef unsigned char Boolean;

typedef unsigned char Octet;

static Boolean TRUE;

static Boolean FALSE;

#include "env.h"

#include "cstring.h"

#include "struc.h"

#include "array.h"
```

```
#include "seq.h"

#include "union.h"

#include "typeinfo.h"

#include "princ.h"

#include "idefclnt.hh"

#include "orb.h"

#include "boa.h"

#include "bopt.h"

#include "object.h"

#include "typecode.h"

#include "any.h"

#include "value.h"

#include "mstrm.h"

#include "prop.h"

#include "context.h"

#include "optr.h"

};
```

Code Block 22 - PostModern Computing (now Visigenic)'s CORBA Class

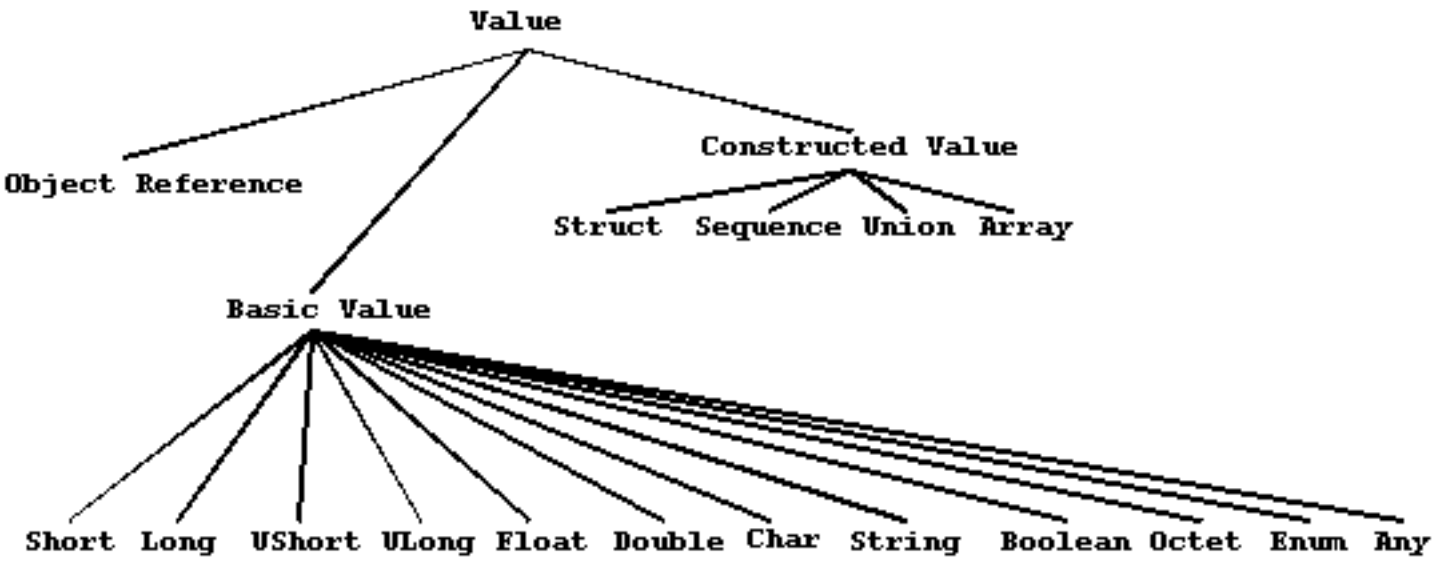
By declaring all the ORBeline names inside the class `CORBA`, they can be referenced by using the scope delimiter operator and the encapsulating class. Continuing the above example, the class `Object` is denoted

```
CORBA::Object
```

When an ORBeline type name is referenced outside the scope of another ORBeline class, the scope delimiter must be used.

ORBeline's Typecodes , Any, and Value Classes

In section 3.1.2, the primitive concept of an CORBA Value and Type was presented. Figure 4 - OMG Specified Type Values, is shown here again.



The CORBA concepts of Values and Types is confusing because they refer to the same thing; only the point of view is different. A CORBA object is an interface, specified with IDL, which hides an object implementation in the ORB. The object interface declares all possible public requests on the object implementation. These requests can have input, output, and return arguments. The "type" of these arguments are referred to as CORBA Type and an instance of a particular CORBA Type is a CORBA Value. For necessary reasons which will be describes shortly, ORBeline adds to the confusion by declaring, in many cases, three (3) C++ classes for every apparent "type". Below are the three classes which model the CORBA "String" type/value. (See Appendix A, page 10,12, & 26)

```
class CORBA::String; // "cstring.h"

class CORBA::ValueString; // "value.h"

class CORBA::StringTypeCode; // "typecode.h"
```

In ORBeline, the CORBA value "String" is implemented using the `CORBA::String` class. The CORBA type "String" is implemented using the C++ type of the `String` class. In other words, the

CORBA value "String" is an instance of the ORBeline C++ class `CORBA::String`. The CORBA type "String" maps directly to the C++ type for the ORBeline class `CORBA::String`.

When you read the ORBeline framework specifications, you find many "String" classes. What are the `CORBA::ValueString` and `CORBA::StringTypeCode` classes for? The answer is that they are used by the implementation of the `CORBA::Any`, `CORBA::Value`, and `CORBA::TypeCode` classes. As presented in Section 3.1.2, the CORBA Value "Any" is an opaque type place holder used for wildcarding "any" IDL type including primitive types and user defined "constructed" types.

ORBeline implements the CORBA "Any" value using the C++ class `CORBA::Any`. The `CORBA::Any` class uses the `CORBA::Value` class and the `CORBA::Value` class uses the `CORBA::TypeCode` class. In order to implement the CORBA "Any" concept, every primitive "type" needs an associated class derived from both `CORBA::Value` and `CORBA::TypeCode`. In addition to using the CORBA type "Any" in an IDL declaration, the programmer can build dynamic requests using the Dynamic Invocation Interface and Interface Repository. ORBeline provides the capability to build dynamic requests using the `CORBA::Value` classes and `CORBA::TypeCode` classes. The rest of this section and the following two (2) sections describe the implementation details of the ORBeline framework with respect to the `CORBA::TypeCode`, `CORBA::Any`, and `CORBA::Value` classes.

The TypeCode Classes

There is a `CORBA::ValueX` class for every "Value" specified in the CORBA Specification (See Figure above). These classes inherit from the `CORBA::Value` class. The "constructed values" are specialized in the `CORBA::ValueConstructed` class and its derived sub-classes. (See Appendix A, page 11) Since the `CORBA::Value` classes are used for referencing within the ORB even the C++ primitives are modeled (e.g. `CORBA::ValueShort`, `CORBA::ValueChar`).

In addition to the implementation classes and typedefs for the primitive CORBA "Values",

```
typedef Object * ObjectRef;

typedef short Short;

typedef long Long;

typedef unsigned short UShort;

typedef unsigned long ULong;
```

```
typedef char Char;

typedef float Float;

typedef double Double;

typedef unsigned char Boolean;

typedef unsigned char Octet;

static Boolean TRUE;

static Boolean FALSE;

class Structure;

class Array;

class Sequence;

class String;

class Union;

class Any;
```

2>Code Block 23 - PostModern Computing (now Visigenic)'s CORBA "Values" Implementation

there are also CORBA "TypeCodes". ORBeline implements CORBA "TypeCodes" as C++ classes. This is why the `CORBA::StringTypeCode` class exists. A CORBA "TypeCode" consists of a "Kind" value and a parameter list for more complicated types. The following table shows this relationship.

TypeCode Const (* C++ class)	Kind	Parameter List
`TC_null`	`tk_null`	none
`TC_void`	`tk_void`	none
`TC_short`	`tk_short`	none
`TC_long`	`tk_long`	none
`TC_ushort`	`tk_ushort`	none

TC_ulong	tk_ulong	none
TC_float	tk_float	none
TC_double	tk_double	none
TC_boolean	tk_boolean	none
TC_char	tk_char	none
TC_octet	tk_octet	none
TC_any *	tk_any	none
TC_TypeCode *	tk_TypeCode	none
TC_Principal *	tk_Principal	none
TC_Object *	tk_objref	none
Structure * (const generated)	tk_struct	struct-name, {member, TypeCode}
Union * (const generated}	tk_union	union-name, switch TypeCode {label-value, member-name, TypeCode}
TC_string *	tk_string	maxlen
Enum *	tk_enum	{enum-name, enum-id}
Sequence * (const generated)	tk_sequence	TypeCode, maxlen
Array * (const generated)	tk_array	TypeCode, length

Table 2 - PostModern Computing (now Visigenic)'s TypeCodes and "Kinds"

The "kind" values are declared in the `CORBA::TypeCode` class. The TypeCodes for the C++ primitive types are also defined as static const instantiations within the `CORBA::TypeCode` class. Notice in the following declaration of the `CORBA::TypeCode` class: (1) the enumeration of `TCKind` and `tk_x` values, (2) the static const members for the primitive types (3) and the `_kind` and `_parameters` data members.

```
/*************************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */
```

```
/*************************************************************************/

class TypeCode: public Object

{

public:

enum TCKind {

tk_null,

tk_void,

tk_short,

tk_long,

tk_ushort,

tk_ulong,

tk_float,

tk_double,

tk_boolean,

tk_char,

tk_octet,

tk_any,

tk_TypeCode,

tk_Principal,

tk_objref,
```

```
tk_struct,

tk_union,

tk_enum,

tk_string,

tk_sequence,

tk_array

};

TypeCode() { _kind = tk_null; }

TypeCode(TCKind kind);

TCKind kind() const { return _kind; }

Long param_count() const { return (Long)_parameters.size(); }

Boolean equal(const TypeCode& tc) const

{ return (_kind == tc._kind &&

_parameters.isEqual(tc._parameters)); }

Any* parameter(Long index, Environment& env ) const;

static const TypeCode TC_null;

static const TypeCode TC_void;

static const TypeCode TC_short;

static const TypeCode TC_long;

static const TypeCode TC_ushort;

static const TypeCode TC_ulong;
```

```
static const TypeCode TC_float;

static const TypeCode TC_double;

static const TypeCode TC_boolean;

static const TypeCode TC_char;

static const TypeCode TC_octet;

static const TypeCode TC_any;

static const TypeCode TC_TypeCode;

static const TypeCode TC_Principal;

virtual ~TypeCode() {};

virtual Boolean is_primitive() const { return 1; }

static const char *convert_to_string(TCKind kind);

static TypeCode *_read(NCistream& strm);

static TypeCode *_create(const TypeCode& code);

protected:

TCKind _kind;

DSOrderedCltn _parameters;

void _read_parameters(NCistream& strm);

void operator=(const TypeCode&);

private:

static const TypeInfo _class_info;
```

```
public:

TypeCode(NCistream& strm);

static const TypeInfo *_desc();

virtual const TypeInfo *_type_info() const;

virtual void *_safe_narrow(const TypeInfo *)const;

static TypeCode *_narrow(const Object *obj);

static Object *_reader(NCistream& strm)

{ return TypeCode::_read(strm); }

void _writer(NCostream& strm) const;

Boolean _is_local() const { return 1; }

int compare(const TypeCode& tc) const;

unsigned hash() const;

virtual const char *_interface_name() const

{ return "CORBA::TypeCode";}

};
```

Code Block 24 - PostModern Computing (now Visigenic)'s CORBA::Typecode Class

The `CORBA::TypeCode` class inherits from `CORBA::Object` which will be presented in the next section. The `CORBA::Object` class encapsulates an object interface within the ORB. By having `CORBA::TypeCode` or any other class inheriting from `CORBA::Object` makes the derived class a specialized object in the ORB. Developers will use the "TypeCode" objects when querying the Interface Repository or using an "Any" argument. In the example mentioned above, the `CORBA::String` class also has the `CORBA::StringTypeCode` and `CORBA::ValueString` classes. The `CORBA::StringTypeCode` class inherits from `CORBA::TypeCode` and is therefore derived from `CORBA::Object` and can be an object in the ORB. The static const value `TC_String` is defined in

the class.

```
/*************************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/*************************************************************************/

class StringTypeCode: public TypeCode

{

private:

static const TypeInfo _class_info;

public:

static const StringTypeCode TC_string;

StringTypeCode(Long maxlen);

StringTypeCode(ValueLong *maxlen);

StringTypeCode(NCistream& strm);

Boolean is_primitive() const { return 0; }

~StringTypeCode() {}

static const TypeInfo *_desc();

virtual const TypeInfo *_type_info() const;

virtual void *_safe_narrow(const TypeInfo *)const;
```

```
static StringTypeCode *_narrow(const Object *obj);

const char *_interface_name() const

{ return "CORBA::StringTypeCode";}

};
```

Code Block 25 - PostModern Computing (now Visigenic)'s CORBA::StringTypeCode Class

The Any Class

When the "Any" type is specified in IDL then the `CORBA::Any` class is used in the ORBeline implementation. Every instance of the `CORBA::Any` class is associated with an instance of the `CORBA::Value` class as seen in the following declaration:

```
/*************************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/*************************************************************************/

class Any: public DSResource

{

DECLARE_NCOBJECT_CLASS(Any);

protected:

Value *_value;

public:

Any();
```

```
Any(const Any& a);

Any(Value *value);

Any(const Value& value);

Any(const TypeCode& code);

Any(TypeCode::TCKind kind);

~Any();

void copyFrom(NCistream& strm);

const TypeCode* type() const;

void type(const TypeCode& code);

TypeCode::TCKind kind() const;

Value * value() { return _value; }

const Value * value() const { return _value; }

void value(const Value& );

void value(Value *);

Boolean is_primitive() const;

Long member_count() const;

short shortValue() const;

long longValue() const;

unsigned short ushortValue()const;

unsigned long ulongValue()const;

float floatValue( ) const;
```

```
double doubleValue() const;

char charValue() const;

const char * stringValue() const;

const TypeCode& typeCodeValue() const;

Object *objrefValue() const;

void shortValue(short);

void longValue(long);

void ushortValue(unsigned short);

void ulongValue(unsigned long);

void floatValue(float);

void doubleValue(double);

void charValue(char);

void stringValue(const char*);

void typeCodeValue(const TypeCode& );

void objrefValue(Object *obj);

const Value& member(const char *member_name) const;

Value& member(const char *member_name);

void get(NCistream& strm);

void put(NCostream& strm) const;

};
```

Code Block 26 - PostModern Computing (now Visigenic)'s CORBA::Any Class

Notice that the CORBA::Any class has many member functions which allow the developer to treat the "Any" argument as what ever primitive type the developer chooses. The CORBA::Any class inherits from DSResource which allows CORBA::Any to be a distributed object in that it can be send over the wire to another process host, or network. The _value data member of CORBA::Any is a pointer to the abstract CORBA::Value object. This object encapsulates the real value represented by the CORBA::Any instance.

The Value Class

The CORBA::Value class is shown here:

```
/*************************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/*************************************************************************/

class Value: public DSResource

{

DECLARE_NCOBJECT_ABSTRACT_CLASS(Value);

protected:

Value() {}

public:

static Value* factory(TypeCode *);

static Value* factory(const TypeCode& );
```

```
virtual ~Value() {}

virtual TypeCode::TCKind kind() const =0;

virtual const TypeCode& typeCode() const=0;

virtual void value(const Value& val) = 0;

virtual void copyFrom(NCistream& strm) = 0;

virtual Boolean is_primitive() const { return 1; }

virtual void add_member(Value *);

virtual Value& member(const char *name);

virtual const Value& member(const char *name) const;

virtual ULong member_count() const { return 0; }

virtual Value& memberAt(Long index);

virtual const Value& memberAt(Long index) const;

virtual short shortValue() const;

virtual long longValue() const;

virtual unsigned short ushortValue()const;

virtual unsigned long ulongValue()const;

virtual float floatValue( ) const;

virtual double doubleValue() const;

virtual char charValue() const;

virtual const char * stringValue() const;

virtual const TypeCode& typeCodeValue() const;
```

```
virtual const Any& anyValue() const;

virtual Any& anyValue();

virtual Object *objrefValue() const;

virtual void shortValue(short);

virtual void longValue(long);

virtual void ushortValue(unsigned short);

virtual void ulongValue(unsigned long);

virtual void floatValue(float);

virtual void doubleValue(double);

virtual void charValue(char);

virtual void stringValue(const char*);

virtual void stringValue(const String&);

virtual void typeCodeValue(const TypeCode&);

virtual void anyValue(const Any&);

virtual void anyValue(Any *value);

virtual void objrefValue(Object *value);

};
```

Code Block 27 - PostModern Computing (now Visigenic)'s CORBA::Value Class

The CORBA::Value class uses a static factory member function [20] called `factory()` which produces the appropriate `CORBA::ValueX` object from a TypeCode or another `CORBA::Value` object. The `CORBA::Value` class also provides accessor functions for use by derived classes. Again,

the CORBA::Value class inherits from DSResource so that it can be a distributed object. Continuing the example of the CORBA::String class, the CORBA::ValueString class is shown here:

```
/*************************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/*************************************************************************/

class ValueString: public Value

{

DECLARE_NCOBJECT_CLASS(ValueString);

private:

String _value;

public:

ValueString() {}

ValueString(const char *val) : _value(val) {}

ValueString(const String& val) : _value(val) {}

~ValueString() {}

TypeCode::TCKind kind() const

{ return TypeCode::tk_string; }

const TypeCode& typeCode() const

{ return StringTypeCode::TC_string; }
```

```
void copyFrom(NCistream& strm)

{ _value.copyFrom(strm); }

void value(const Value& val)

{ stringValue(val.stringValue()); }

const char *stringValue() const

{ return (const char *)_value; }

void stringValue(const char * val)

{ _value = val; }

void stringValue(const String& val )

{ _value = val; }

};
```

Code Block 28 - PostModern Computing (now Visigenic)'s CORBA::StringValue Class

The `CORBA::ValueString` class contains a private data member which is a pointer to the actual data object which the Value class encapsulates in the ORB. The `kind()` member function always returns `CORBA::TypeCode::tk_string. The typeCode()` member function always returns `CORBA::StringTypeCode::TC_string`. Because the `kind()`and `typeCode()` member functions are virtual, the proper TypeCode and Kind will be returned when these functions are called on a `CORBA::Value` object pointer.

The Value and TypeCode classes are primarily used when querying/browsing the Interface Repository (IR) for the purposes of producing Requests on the Dynamic Invocation Interface (DII). As mentioned in Section 3.2.3, there is no application of the DII and IR in this research. Therefore, further details on the use of Value and TypeCode classes will not be shown in this report. In addition to Value and TypeCode, the following ORBeline namespaces are provided for the classes which interface to the DII and IR:

```
CORBA_DII::
```

```
CORBA_IR::
```

The use of the "Any" wildcard type specifier is a consideration in many object interfaces. However, polymorphism can replace the "Any" type in many situations. For instance, if an ORB object interface is declared to perform searching then abstract Goal and Domain classes could also be declared to be used as arguments to the Search() method. Specialized Goal and Domain classes can be declared as the developer needs. An Any class would not be a productive argument for the Search() method. However, an Any might be a good type specifier for a function to populate a given domain because "anything" can be part of a domain.

This presentation of the opaque "Any" type and the complex dynamic request "typecodes" and "values" represent a large volume of the ORBeline implementation. The material presented here shows that a lot of the requirements of the CORBA 1.1 specification might never be used by most ORB users. The rest of the material in this section describe the components of the ORBeline framework which allow the developer to build static IDL interfaces in C++.

Environment Class

One of the most important features of the ORB is the exception handling capability. Exceptions are errors that occur during run-time. For instance, if an object's interface is called which causes the object to try to read some information out of a file and the file can not be opened, an error or exception is raised. If another part of the system has an interest in the presence of that type of exception, a specific error or exception handler will be executed. The ANSI/ISO C++ standard has approved an exception handling feature but it also is not common in all compilers. Presently, CORBA 2.0 compliant products are starting to map CORBA exceptions directly into standard C++ exceptions. The ORBeline framework implements CORBA 1.1 compliant exceptions using the CORBA::Environment and CORBA::Exception classes.

The CORBA::Environment class is the interface to the ORBeline exception system. Whenever a request is made on an ORB Object an instance of a CORBA::Environment object is integrated with the request arguments and returned to the caller. The caller or client can inspect the CORBA::Environment object returned with the results if the request failed to see why. A request can fail for common reasons like a network connection going down or a more specific reason which is user defined. Below is the declaration of the CORBA::Environment class. Notice that it is part of the DSResouce tree so that an CORBA::Environment can be a distributed object. The important interfaces to this class are the member functions which set an exception, check an exception, and clear an exception. There is a private data member which is a pointer to an instance of a CORBA::Exception class.

```
/*************************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */
```

```
/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/*****************************************************************/

class Environment: public DSResource

{

DECLARE_NCOBJECT_CLASS(Environment);

private:

Exception *_exception;

public:

Environment()

{ _exception = (Exception *)NULL; }

virtual ~Environment()

{ DSResource::unref(_exception); }

virtual void copyFrom(NCistream& strm);

void get(NCistream& strm)

{ copyFrom(strm); }

void put(NCostream& strm) const

{ write(strm); }

Boolean check_exception() const

{ return _exception != (Exception *)NULL; }

const Exception *exception_value() const
```

```
{ return _exception; }

Exception *exception_value()

{ return _exception; }

void exception_value(Exception *exp)

{ DSResource::unref(_exception);

_exception = exp;

DSResource::ref(_exception); }

void clear_exception()

{ DSResource::unref(_exception);

_exception = (Exception *)NULL; }

};
```

Code Block 29 - PostModern Computing (now Visigenic)'s t Class

Exception Class

As described above, ORBeline implements exceptions as distributed C++ objects. The CORBA::Exception class inherits from DSResource so that it can be distributed object. The CORBA::Exception class is abstract because its constructor is not public. Exceptions are further classified per the CORBA 1.1 specification as system exceptions and user exceptions. There are a finite number of system exceptions implemented as specialized CORBA::SystemException classes. Through an IDL declaration, a programmer create a user exception which derives from the CORBA::UserException exceptions are:

System Exception Class Names	Description
CORBA::BAD_PARAM	an invalid parameter was passed
CORBA::NO_MEMORY	dynamic memory allocation failure
CORBA::IMP_LIMIT	violated the implementation limit

CORBA::COMM_FAILURE	communication failure
CORBA::INV_OBJREF	invalid object reference passed
CORBA::NO_PERMISSION	no permission for attempted op
CORBA::INTERNAL	ORB internal error
CORBA::MARSHAL	error marshalling param/result
CORBA::INITIALIZE	ORB initialization failure
CORBA::NO_IMPLEMENT	operation implementation unavailible
CORBA::BAD_TYPECODE	bad typecode
CORBA::BAD_OPERATION	invalid operation
CORBA::NO_RESOURCES	insufficient resources for request
CORBA::NO_RESPONSE	response to request not yet availible
CORBA::PERSIST_STORE	persistent storage failure
CORBA::BAD_INV_ORDER	routine invocation out of order
CORBA::TRANSIENT	transient failure
CORBA::FREE_MEM	cannot free memory
CORBA::INV_IDENT	invalid identifier syntax
CORBA::INV_FLAG	invalid falg was specified
CORBA::INTF_REPOS	error accessing interface repository
CORBA::CONTEXT	error processing context object
CORBA::OBJ_ADAPTER	failure detected by object adapter
CORBA::DATA_CONVERSION	data conversion error
CORBA::UNKNOWN	the unknown exception

Table 3 - PostModern Computing (now Visigenic)'s Standard System Exceptions Descriptions

The abstract CORBA::Exception class is very simple as seen below. When a client receives a response and the environment returns an exception, two (2) actions can be taken. First, the string name of the exception can be retrieved using the _id() member function. If more information is needed by the client, the CORBA::Exception pointer needs to be cast to a more specific CORBA::SystemException or CORBA::UserException.

```
/*************************************************************************/
```

```
/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/*********************************************************************/

class Exception: public DSResource

{

DECLARE_NCOBJECT_CLASS(Exception);

protected:

Exception() {}

public:

virtual ~Exception() {}

virtual const char *_id() const;

};
```

Code Block 30 - PostModern Computing (now Visigenic)'s CORBA::Exception Class

A client can cast a returned CORBA::Exception pointer to a CORBA::SystemException pointer by calling the static _cast() member function of the CORBA::SystemException class. The resulting pointer will be NULL if the cast was invalid. The standard system exception classes contain a "minor" code which further describes the exception which occurred. Since exceptions occur during requests, standard system exceptions also contain a "completion" status. The completion status indicates whether the request was completed regardless of the exception. The CORBA::SystemException class is below:

```
/*********************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */
```

```
/* All rights reserved. */

/*************************************************************************/

class SystemException: public Exception

{

DECLARE_NCOBJECT_CLASS(SystemException);

public:

enum completion_status {

YES =0,

NO=1,

MAYBE=2

};

private:

ULong _minor;

completion_status _status;

protected:

SystemException() { _minor = 0; _status = NO; }

public:

SystemException(ULong minor, completion_status status) {

_minor = minor;

_status = status; }

virtual ~SystemException() {}

ULong minorFlag() const { return _minor;}
```

```
ULong& minorFlag() { return _minor; }

void minorFlag(ULong val) { _minor = val; }

completion_status completed() const { return _status; }

completion_status& completed() { return _status; }

void completed(completion_status status) { _status = status; }

static SystemException *_cast(Exception *exc);

static const SystemException *_cast(const Exception *exc);

};
```

Code Block 31 - PostModern Computing (now Visigenic)'s CORBA::SystemException Class

The CORBA::UserException class is very similar to the CORBA::SystemException class however, it does not have a minor code or completion status. Again, the user exception classes are declared in IDL and defined by the programmer.

IDL Constructed Types (array,sequence,union,structure)

As describes in section 3.1.2, the programmer can declare user defined, "constructed" types using IDL. These user types are declared in IDL from the primitive CORBA types like short, ushort, float, etc. Also, constructed types can contain previously declared types. The four types of constructed value implementations will be described below. For all the types, the IDL compiler will map the declarations into C++ code.

The array and sequence types are mapped to C++ using macros and typedefs. There are simple, abstract classes defined which all user defined arrays and sequences classes inherit from. IDL structure and union declarations are mapped into C++ classes which inherit from CORBA::Structure and CORBA::Union respectively. Consider the IDL declaration of a Point structure. Notice that the IDL code is very similar to C++.

```
struct Point

{

float x;
```

```
float y;

};
```

Code Block 32 - Example of IDL for Point Structure

The IDL compiler creates a simple implementation class derived from CORBA::Structure

```
class Point : public CORBA::Structure

{

DECLARE_NCOBJECT_CLASS(Point);

private:

CORBA::Float _x;

CORBA::Float _y;

public:

Point();

~Point();

void copyFrom(NCistream& strm);

void operator=(const Point& obj);

Point(const Point& obj);

CORBA::Float& x() { return _x; }

CORBA::Float x() const {return _x; }

void x(CORBA::Float val) { _x = val; }

CORBA::Float& y() { return _y; }
```

```
CORBA::Float y() const {return _y; }

void y(CORBA::Float val) { _y = val; }

};
```

Code Block 33 - PostModern Computing (now Visigenic)'s ORBeline IDL Compiler Output (Point Structure Class)

The IDL compiler will also create member function definitions for the functions declared by the NetClasses macros. (see section 2.8.4)

```
DEFINE_NCOBJECT_CLASS1(Point, CORBA::Structure);

Point::Point(NCistream& strm) : CORBA::Structure(strm)

{

strm >> _x;

strm >> _y;

}

void Point::copyFrom(NCistream& strm)

{

strm >> _x;

strm >> _y;

}

Point::Point(const Point& obj)

{

_x = obj.x();

_y = obj.y();
```

```
}

void Point::operator=(const Point& obj)

{

_x = obj.x();

_y = obj.y();

}

void Point::write(NCostream& strm) const

{

strm << _x;

strm << _y;

}

void Point::prettyPrint(NCostream& strm) const

{

strm << "Point" << endl;

strm << "\tx: " << _x << endl;

strm << "\ty: " << _y << endl;

}

int Point::compare(const NCObject& p) const

{

const Point& nc_temp = NCCONST_REF_CAST(Point, p);
```

```
int diff = 0;

diff = ((int) _x - (int) nc_temp._x);

if (diff)

return diff;

diff = ((int) _y - (int) nc_temp._y);

if (diff)

return diff;

return diff;

}

unsigned Point::hash() const

{

unsigned h = 0;

h ^= (unsigned) _x;

h ^= (unsigned) _y;

return h;

}

Point::Point()

{

_x = 0;

_y = 0;

}
```

```
Point::~Point()

{

}
```

Code Block 34 - PostModern Computing (now Visigenic)'s ORBeline IDL Compiler Output (Point Structure Definition)

IDL union types are built in the same fashion where user declared CORBA Unions are classes derived from CORBA::Union.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

ORBeline's ORB Objects

In addition to interactions with the ORB core, all interactions and dialog between clients and servers are done with respect to ORB Objects. From the point of view of the server, the ORB Object is an Object Implementation which is registered with the ORB. In the client, the same Object is hidden by a Object Reference which the client uses to access the interface of the Object in the server.

Object Reference

If the ORB Object is located in another process or node then the Object Reference refers to an ORB Object proxy class which hides the remote aspect of the actual ORB Object. The standard convention for specifying an Object in the ORB is through the Object Reference. ndard CORBA type/value. The implementation details of the Object Reference is hidden from the user.

In OBReline, Object References are mapped directly into C++ pointers to classes derived from `CORBA::Object`. In order for a client to access an ORB Object the client must have the Object Reference. An Object Reference can be obtained directly from the ORB, returned as an argument from a previous request, or acquired outside the ORB in a string form. An Object Reference can be "stringified" (converted to a string form) and converted back to a real Object Reference again using ORB functions. This implementation independent string form allows Object References to be stored in non-ORB systems, files like World Wide Web (WWW) hypertext markup language (HTML), or database tables.

Interface Name and Object Name

In the case where the ORB generates and returns an Object Reference ("binding"), a hierarchy of information can be used to specify the desired Object. All ORB Objects are at least described by their Interface Name. The Interface Name is the name of the interface specified in the IDL. For example, the IDL for a `search_engine` interface is specified in IDL as:

```
interface search_engine {

...

};
```

The Interface Name "`search_engine`" specifies an interface of type `search_engine` which will be implemented in ORBeline as a C++ class called `search_engine` which is derived from

CORBA::Object. In ORBeline, an ORB Object can be referenced more specifically by its Object Name. The Object Name is a string which denotes a particular instance of an Interface Name. For example, a client could asked the ORB to locate an Object not only by its IDL matching Interface Name but also the particular Object Name. Even though the ORB can find an Object when the requester is on a different node in the network, a hostname can also be used to specify the node location of an ORB Object. In other words, a client can ask for a `search_engine` Object on host A.

Using IDL, a designer can have Object Interfaces inherit from other interfaces. For instance, the `distributed_search_engine` Object Interface can inherit from the general `search_engine` Object Interface. In this case, if a client asks the ORB to return an Object Reference to a search_engine Interface, the resulting Object Reference could reference a `distributed_search_engine` instead. The feature of Object Interface inheritance works just like C++ class inheritance (See section 1.1.1.3). Both Object Interface functions and properties are inherited.

Object Class

One of the most powerful features of ORBs is the ability to "bind" to a specified Object in the ORB. When a client asks the ORB to generate and return an Object Reference to specified Object, it is called "binding" to an ORB Object. The details of the CORBA::Object class must be reviewed to understand the implementation of the "bind" capability. (See Appendix A, page 22)

The abstract CORBA::Object class encapsulates the details of sustaining an Object within the ORB. As mentioned above, every Object Interface declared in IDL is mapped to a C++ object which derives from CORBA::Object. Therefore, in addition to the interface members specified in the IDL, the new object class inherits all the data and function members of the CORBA::Object class. For instance, the interface `Grid` is declared in IDL as:

```
// struct Point already declared

interface Grid

{

void query(in Point the_point, out float the_value);

};
```

Code Block 35 - PostModern Computing (now Visigenic)'s ORBeline IDL Compiler Output (Grid Interface)

The IDL compiler will create the C++ code for the Grid class as follows:

```
class Grid: public virtual CORBA::Object

{

private:

static const CORBA::TypeInfo _class_info;

public:

static const CORBA::TypeInfo *_desc();

virtual const CORBA::TypeInfo *_type_info() const;

virtual void *_safe_narrow(const CORBA::TypeInfo *) const;

static CORBA::Object *_reader(NCistream& strm)

{return new Grid(strm);}

protected:

Grid(const char *obj_name = NULL) :CORBA::Object(obj_name) {}

Grid(NCistream& strm) :CORBA::Object(strm) {}

virtual ~Grid() {}

public:

enum _Grid_Methods {

_Grid_M_query = 0

};

static Grid *_narrow(const CORBA::Object *obj);

static Grid *_bind(CORBA::Environment &_env,
```

```
const char *object_name = NULL,

const char *host_name = NULL,

const CORBA::BindOptions* opt = NULL);

static Grid *_bind(const char *object_name = NULL,

const char *host_name = NULL,

const CORBA::BindOptions* opt = NULL)

{ CORBA::Environment env;

return _bind(env, object_name, host_name, opt); }

virtual const char *_interface_name() const { return "Grid"; }

void query(const Point& the_point, CORBA::Float& the_value,

CORBA::Environment& _env);

virtual void query(const Point& the_point,

CORBA::Float& the_value)

{ query(the_point, the_value, _environment()); }

};

typedef Grid* GridRef;
```

Code Block 36 - PostModern Computing (now Visigenic)'s ORBeline IDL Compiler Output (Grid Interface Class)

Notice that most of the `Grid` class is generic except for the query functions. There is a query function for the case when the global, static `CORBA::Environment` object (see Section 3.3.2.3) should be used and a second query function for the case where the requester calls the query function with a local `CORBA::Environment` object reference. The remaining class functions come with every derived `CORBA::Object` class. The `CORBA::Object` class, which `Grid` and all other interfaces inherit

from, is declared as follows.

```
/*************************************************************************/

/* ORBeline (c) by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */

/* All rights reserved. */

/*************************************************************************/

class Object {

public:

ULong _ref_count() const;

void _ref();

void _release();

Object *_duplicate() const;

Object *_clone() const;

Boolean _is_nil() const;

virtual Boolean _is_local() const;

Boolean _is_remote() const;

Boolean _is_bound() const;

Boolean _is_persistent() const;

const char *_object_name() const;

virtual const char*_interface_name() const;
```

```
const Principal *_principal() const;

void _principal(Principal *principal);

const BindOptions *_bind_options() const;

static const BindOptions *_default_bind_options();

static void _default_bind_options(const BindOptions& );

_ObjectImpl *_object_impl() { return _impl; }

Environment& _environment();

static const TypeInfo *_desc() { return &_class_info; }

virtual const TypeInfo *_type_info() const {return _desc();}

virtual void _writer(NCostream& ) const;

static Object *_reader(NCistream& strm);

static Object *_read(NCistream& strm,

const TypeInfo *expectedType);

static void write(const Object *obj, NCostream& strm,

const TypeInfo *expectedType);

virtual void *_safe_narrow(const TypeInfo *desc) const;

private:

static const TypeInfo _class_info;

_ObjectImpl *_impl;

protected:

Object(const Object& ) {}
```

```
void operator=(const Object& ) {}

virtual ~Object();

Object(const char *object_name = NULL,

ORB::_ORBType type =

ORB::_ORB_POSTMODERN);

Object(NCistream& strm);

void _object_name(const char *name);

void _bind(const char *interface_name,

Environment& env,

const char *object_name =

(const char *)NULL,

const char *host_name =

(const char *)NULL,

const BindOptions *options =

(const BindOptions *)NULL);

void _rebind(Environment& env);

void _unbind(Environment& env);

MarshalStream *_create_oneway_request(

const char *interface_name,

ULong method_id,

Environment& env);
```

```
MarshalStream *_create_invoke_request(
const char *interface_name,
ULong method_id,
Environment& env);
void _send_oneway(Environment& env);
MarshalStream *_invoke(Environment& env);
void _orb_type(ORB::_ORBType
type=ORB::_ORB_POSTMODERN);
void _register_implementation(
const char *interface_name,
ULong num_methods,
_PMCSkelFunc *func_list,
void *user_data,
Environment& env,
BOA::RegistrationScope scope,
BOA::IMPLEventHandler *handler=
(BOA::IMPLEventHandler *)NULL);
void _register_implementation(
const char *interface_name,
ULong num_methods,
```

```
_PMCSkelFunc *func_list,

void *user_data,

Environment& env,

BOA::IMPLEventHandler *handler=

(BOA::IMPLEventHandler *)NULL)

{register_implementation(interface_name,num_methods, func_list, user_data,env, BOA::scope(),
handler);}

void _unregister_implementation(

const char *interface_name,

void *user_data,

Environment& env);

static Object *_implementation(const char *interface_name,

const char *object_name =

(const char *)NULL);

public:

virtual void _receive_reply(MarshalStream&,Environment& ) {}

virtual void _exception(Environment& env);

};

typedef Object *ObjectRef;
```

Code Block 37 - PostModern Computing (now Visigenic)'s CORBA::Object Class

Even though the `CORBA::Object` class does not inherit from `DSResource`, there are many member functions which resemble the behaviors of `DSRsource` for reference counting, cloning, RTTI, object

I/O, and safe run-time casting. In addition, the CORBA::Object class provides behavior for determining ORB Object state, identity, client or "principal" state, bind options, operating system implementation details, and exception environment. Other private member functions provide the behaviors for binding Object instances to clients, generating requests, and registering implementations.

The reference counting feature is very important during distributed processing. The size of the reference count of an object determines how many clients are using the ORB Object. (See Section 2.8.3.2) The reference count can be accessed with the `_ref_count()` member function. The Object's can be referenced (increase the reference count) by calling `_ref()` and dereferenced or released calling the `_release()` function.

The Object can be queried using `_is_local()`, `_is_remote()`, `_is_bound()`, and `_is_persistent()` to determine if the Object is in the same address space, on another network node or process, currently bound to a client, or persistent respectively. The Object Name can be accessed using the `_object_name()` function and the Interface Name by the `_interface_name()` function. Notice that the `_interface_name()` function is virtual so that proper Interface Name is returned when the caller has a pointer to a general interface which is actually a more specific class. For instance, in the following code example

```
distributed_search_engine the_DSE;

search_engine *the_SE_ptr = &the_DSE;

char *the_interface_name = the_SE_ptr->_interface_name();
```

the result of the call to _interface_name() would call the _interface_name() function of distributed_search_engine class instead because of the virtual specifier.

The CORBA::Object class is associated with CORBA::Principal, CORBA::BindOptions, and CORBA::ObjectImpl classes and there are accessor functions for those as well. These other ORBeline classes will be described later.

As mentioned above, there are member functions in the CORBA::Object class which resemble the behavior of the DSResource class. Instead of using the NCTypeInfo (see section 2.8.3.3.2) class to provide RTTI, a special class called CORBA::TypeInfo is used which is specific to CORBA::Object. As in the case with NCObject, the CORBA::Object class has a static member of type CORBA::TypeInfo. A run-time safe casting operation called `_safe_narrow()` is also provided.

ORB Object Binding

In order the understand the details of the CORBA::Object class with respect to binding and Object Interface functions, the implementation code of the Grid class is presented below. Refer to the class declaration generated by the ORBeline IDL compiler (Code Block 36).

```
const CORBA::TypeInfo Grid::_class_info("Grid",

&Grid::_reader,

CORBA::Object::_desc(),

0);

const CORBA::TypeInfo *Grid::_desc()

{

return &_class_info;

}

const CORBA::TypeInfo *Grid::_type_info() const

{

return &_class_info;

}

void *Grid::_safe_narrow(const CORBA::TypeInfo *info) const

{

if (&_class_info == info)

return (void *) this;

void *ret = NULL;
```

```
return ret;

}

Grid *Grid::_narrow(const CORBA::Object *obj)

{

void *ptr = obj->_safe_narrow(&_class_info);

return (Grid *) ptr;

}

Grid *Grid::_bind(CORBA::Environment &_env, const char *_object_name,

const char *_host_name, const CORBA::BindOptions *opt)

{

_env.clear_exception();

Grid *_impl;

CORBA::Object *_obj = _implementation("Grid", _object_name);

if (!_obj) {

_impl = new Grid(_object_name);

_impl->CORBA::Object::_bind("Grid", _env, _object_name, _host_name, opt);

if (_env.check_exception()) {

delete _impl;

return NULL;

}
```

```
}

else

_impl = Grid::_narrow(_obj);

return _impl;

}

void Grid::query(const Point& the_point,

CORBA::Float& the_value, CORBA::Environment& _env)

{

_env.clear_exception();

if (_is_local()) {

query(the_point, the_value);

return;

}

CORBA::MarshalStream *_strm = _create_invoke_request(

"Grid", _Grid_M_query,_env);

if (_env.check_exception())

return;

_strm->putStructure(the_point, CORBA::MarshalStream::ARG_IN);

_invoke(_env);

if (_env.check_exception()) {
```

```
if (CORBA::StExcep::TRANSIENT::_cast(_env.exception_value()) != NULL)

query(the_point, the_value, _env);

return;

}

_strm->getFloat(the_value, CORBA::MarshalStream::ARG_OUT);

_strm->flush(_env);

return;

}
```

Code Block 38 - PostModern Computing (now Visigenic)'s ORBeline IDL Compiler Output (Grid Interface Definition)

In the declaration of the `Grid` class, a private data member, `_class_info`, is declared as a static `CORBA::TypeInfo` object. All instances of the `Grid` class are associated to a single `CORBA::TypeInfo` object. This data member,

```
static const CORBA::TypeInfo _class_info;
```

is defined in the source code for the `Grid` class. Again, the definition of `_class_info` is almost identical to the `NCTypeInfo` object. The class name, `"Grid"`, the address of the `Grid::_reader()` member function, and the `_class_info` of the parent class are arguments to the constructor of the `CORBA::TypeInfo` object. The `Grid::desc()` member function accesses the static `_class_info` member. The `Grid::_safe_narrow()` member function, as stated above, provides run-time safe casting according to the type information found in the `_class_info` data member. The `CORBA::Object::_safe_narrow()` function is declared `virtual` so that the correct narrowing function is called in the polymorphic case. The `Grid::_narrow()` member function is a static access to the `Grid` class' narrowing behavior.

The ORBeline IDL compiler declares a new type to implement the specific Object Reference for the `Grid` class. In C++ CORBA mappings, an Object Reference mapped to a typed pointer to the Object class. However, in other language mappings, the Object Reference might be more complicated so the details are kept hidden.

```
typedef Grid* GridRef;
```

The constructors defined for the `Grid` class simply call parent class constructors. The `Grid` class can be initialized with just an Object Name or a new instance can be read off a NetClasses I/O stream class.

Every Interface Class, as seen in the `Grid` class declaration, declares an enumeration of methods specific to the class. In the `Grid` class, the enumeration,

```
enum _Grid_Methods { _Grid_M_query = 0 };
```

provides the low level protocol for ORB requests. When a remote client makes a request on an ORB Interface Object, in addition to the Object Reference and request arguments, the client must tell the ORB the proper ID of the Object Interface's method the request is for. The enumeration of class methods defines the method IDs. These method Ids are the key to performing dynamic requests through the DII. In the case of the static IDL stubs, the method IDs are hard coded into the stub. However, the method IDs are returned from the Interface Repository(IR) during dynamic request creation. The `_Grid_M_query` value is used in the implementation of the `Grid::Query()` member function.

The `Grid::_bind()` member functions are factory methods 24 [21] for generating Object References. These static functions return pointers to `Grid` objects. The actual type of the object depends on the location of the "real" ORB Object. The ORBeline IDL compiler generates a second class which derives from the first class. In this example, the `Grid_impl` class is generated by the ORBeline IDL compiler.

```
class Grid_impl: public virtual Grid

{

protected:

Grid_impl(const char *object_name = NULL);

virtual ~Grid_impl();

public:

virtual const CORBA::TypeInfo *_type_info() const

{ return Grid::_type_info(); }

virtual void *_safe_narrow(const CORBA::TypeInfo *inf) const
```

```
{ return Grid::_safe_narrow(inf); }

virtual const char *_interface_name() const

{ return Grid::_interface_name(); }

virtual CORBA::Boolean _is_local() const

{ return 1; }

virtual void query(const Point& the_point,

CORBA::Float& the_value) = 0;

static void _query(void *obj, CORBA::MarshalStream &strm,

CORBA::Environment& _env,

CORBA::Principal *principal);

};
```

Code Block 39 - PostModern Computing (now Visigenic)'s ORBeline IDL Compiler Output (Grid_impl class)

The `Grid_impl` class is abstract (can not be instanciated) because the constructor function has `protected` access. The `Grid` class represents the "stub" or proxy class for the Grid Object on the client side. The `Grid_impl` class represents the "skeleton" or agent for the Grid Object on the server side. The intention here is to have the programmer derive a third class from `Grid_impl` which actually implements the Grid Object's behavior. Therefore, the `Grid::_bind()` function returns a pointer to an instance of either the `Grid` class, if the Grid Object is remote, or the user class derived from `Grid_impl` if the Grid Object is local. This is the most powerful and elegant aspect of PostModern Computing (now Visigenic)'s ORBeline system. If the "real" ORB Object is in the same address space then the ORB is bypassed entirely and a pointer to the actual Object is returned in a `::_bind()`. The local locator service is provided through the `CORBA::Object` class. The slower (IPC) ORB Core call to the ORBeline Smart Agent for network locator services is only done when the Object is not local. There are two (2) Grid::_bind() functions: one for the case of a supplied CORBA::Environment object and a second for the global CORBA::Environment object. The bind operation takes optional arguments which specify the Object Name, host name, and/or bind options.

Continuing the `Grid` class example, the detailed operations of the `Grid::_bind()` function is as follows (refer to Code Block 38):

1. The `CORBA::Environment` argument, `_env`, is cleared of old exceptions.
2. A local pointer to a `Grid` object is defined named `_impl`.
3. A local pointer to a `CORBA::Object` is defined named `_obj`.
4. `_obj` is assigned the return value of the call to the static `CORBA::Object::_implementation()` member function.
5. The `CORBA::Object::_implementation()` function is called with the hard coded Interface Name `"Grid"` and Object Name (possibly `NULL`).
6. The return value will be a pointer to a valid `CORBA::Object` which satisfies the Interface Name and Object Name if the one is available locally.
7. If the pointer assigned to `_obj` is not `NULL`, then `_obj` is narrowed by calling the static `Grid::_narrow()` method and the result is returned.
8. If the pointer assigned to `_obj` is `NULL`, then the remote bind (connect) is attempted.
9. `_impl` is assigned the address of a newly instanciated `Grid` object (the client side stub class).
10. The `CORBA::Object::_bind()` member function of `_impl` is called with the target arguments. This function will contact the ORB Core to create the connection (according to the `CORBA::BindOptions` argument).
11. After the function returns, the environment, `_env`, is checked for exceptions. If none have occurred then the new `Grid` object is returned. Otherwise, the new `Grid` object is deleted and `NULL` is returned.

ORB Object Method Invocation (Client-Side)

After a `::_bind()` factory function returns with a valid Object Reference, both the accessors to the Object Interface's properties and the methods of the Interface can be called. In the `Grid` class example, the `Grid::query()` method is called with a reference to a `Point` object and a reference to a `CORBA::Float` type. In this case the `Point` object is an input argument and the `CORBA::Float` reference is a return argument. The query behavior takes an x, y coordinate (`Point`) and returns the float value at that location.

If the `Grid::_bind()` factory returned a user derived, local object

```
Grid *the_grid = Grid::_bind(); // returned user derived, local object
```

them a call to the virtual function

```
the_grid->query(the_point,the_float);
```

will engage the correct `query()` function for actual type of the object pointed to by `the_grid`. Therefore, if the object is local then the ORB is bypassed and the actual query behavior is executed as normal in C++. However, if the object is the Interface stub class, `Grid`, then the `Grid::query()` function is executed. The details of the `Grid::query()` function is as follows:

1. The `CORBA::Environment` argument, `_env`, is cleared of old exceptions.
2. A local pointer to a `CORBA::MarshalStream` object is defined named `_strm`. The `CORBA::MarshalStream` class multiply inherits from `NCistream` and `NCostream` to provide a utility class for directing object I/O within the ORB. (See Section 2.8.3.3)
3. `_strm` is assigned the `CORBA::MarshalStream` pointer returned from the `CORBA::Object::_create_invoke_request()` member function. The hard coded Interface Name `"Grid"`, the `Grid` class method ID `_Grid_M_query`, and the `CORBA::Environment` reference `_env` is passed as arguments to the `_create_invoke_request()`.
4. If the pointer assigned to `_strm` is valid, then `the_point` argument is placed on the `CORBA::MarshalStream` variable `_strm` using the `CORBA::MarshalStream::putStructure()` function.
5. The request is then started by calling the `CORBA::Object::_invoke()` member function.
6. If a `CORBA::stExcep::TRANSIENT` exception occurs, the request will be recursively tried again.
7. If the request was successful, the output arguments are pulled off the `CORBA::MarshalStream`, `_strm`, by calling the `CORBA::MarshalStream::getFloat()` function and then returns.

ORB Object Method Invocation (Server-Side)

The scenario shown above has been from the point of view of the client or calling side. Again, in the case where the ORB Object is remote, the calls to the `::_bind()` and Object Interface methods engage the behavior of the client side stub class. On the server side, the abstract skeleton interface class, provides the interface to the ORB. In Code Block 39, the Grid Interface skeleton class `Grid_impl` derives from the client stub class `Grid`. In the ORBeline system, as stated above, the programmer will derive a third class from the abstract `Grid_impl` class which will actually implement the Grid Interface methods. Below is the definitions for the `Grid_impl` class produced by the ORBeline compiler.

```
static CORBA::_PMCSkelFunc _Grid_func_array[] =
```

```
{ &Grid_impl::_query, 0 };

Grid_impl::Grid_impl(const char *object_name) : Grid(object_name)

{

_object_name(object_name);

CORBA::Environment _env;

_register_implementation("Grid", 1, _Grid_func_array,

(void *) this, _env);

}

Grid_impl::~Grid_impl()

{

CORBA::Environment _env;

_unregister_implementation("Grid", (void *) this, _env);

}

void Grid_impl::_query(void *obj, CORBA::MarshalStream &strm,

CORBA::Environment& _env,

CORBA::Principal *principal)

{

Grid_impl *_impl = (Grid_impl *) obj;

_env.clear_exception();

Point the_point;
```

```
strm.getStructure(the_point, CORBA::MarshalStream::ARG_IN);

CORBA::Float the_value;

the_value = (CORBA::Float) 0;

strm.flush(_env);

if (_env.check_exception())

return;

_impl->_principal(principal);

_impl->query(the_point, the_value);

strm.putEnvironment(_env);

strm.putFloat(the_value, CORBA::MarshalStream::ARG_OUT);

_impl->_principal((CORBA::Principal *) NULL);

}
```

Code Block 40 - PostModern Computing (now Visigenic)'s ORBeline IDL Compiler Output (Grid_impl class definitions)

The first component of all skeleton implementations is a static array (scope within the source file only) of type `CORBA::_PMCSkelFunc`. The declaration of the `CORBA::_PMCSkelFunc` type is

```
typedef void (*_PMCSkelFunc)(void *,

MarshalStream& strm,

Environment& env,

Principal *principal);
```

a pointer to a function which takes four (4) arguments: a pointer to `void`, a `CORBA::MarshalStream` object reference, a `CORBA::Environment` object reference, and a pointer to a `CORBA::Principal` object. The function type matches the prototypes of skeleton class

static member functions which bootstrap the Object Interface method invocations. For the Grid Interface example, the `Grid_impl` class has a static member function, `Grid_impl::_query()`. The Grid Interface skeleton's static array of CORBA`::_PMCSkelFunc` types is called `_Grid_func_array`. The `_Grid_func_array` is initialized with the addresses of the skeleton's methods and is then `NULL` terminated. This static function pointer array is used by the ORB Core to find the appropriate method to call according to the ORB Object method ID passed by the client in a request. A server does not know the origin of the request - the request can be from a ORBeline generated stub or a dynamically built request. In the current example, the `_Grid_M_query` method ID is the index into the `_Grid_func_array`. In this way, the proper method can be called using the method ID and bootstraping function array.

The constructor and destructor methods for the skeleton class call the CORBA`::Object::_register_implementation()` and CORBA`::Object::_unregister_implementation()` member functions respectively. The CORBA`::Object::_register_implementation()` function takes the Interface Name, the number of Interface methods (size of the static skeleton function array), the address of the skeleton function array, user data, environment, and optional bind options. In the `Grid_impl` class case

```
_register_implementation("Grid",1,_Grid_func_array,(void *) this, _env);
```

When the ORB Core receives a request from a client the combination of the Interface Name and Method ID in the request is a key into the implementation information. The Interface Name/Method ID key allows the correct static method address to be found. The Object Reference information provided with the request determines the skeleton object pointer in the server. This skeleton object pointer is passed as an argument to the static method along with an environment object reference, CORBA::MarshalStream object reference, and a pointer to principal object.

```
void Grid_impl::_query(void *obj, CORBA::MarshalStream &strm,

CORBA::Environment& _env,

CORBA::Principal *principal)
```

The details of the Grid_impl::_query() are

1. A local Grid_impl object pointer is defined named _impl and is assigned the address of the obj void pointer. _impl actually points to the user object derived from Grid_impl found using the Object Reference in the server.
2. The CORBA`::Environment` argument, `_env`, is cleared of old exceptions.
3. A local Point object is defined named the_point.

4. the_point is initialized by pulling data off the marshal stream by calling the CORBA::MarshalStream::getStructure() function.
5. A local CORBA::Float type is defined named the_value and is set to zero.
6. If there is an exception the request is aborted, otherwise, the Grid_impl object's principal is set. The CORBA::Principal encapsulates information about the caller.
7. Then, the real, user defined method is called on the Grid_impl pointer with the input and output arguments. This is a virtual function call so that the correct method is called in the polymorphic case.
8. The resulting environment, _env, and the resulting float value, the_value, are placed on the marshal stream and the object's principal is cleared.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Advanced Features

The ORBeline 1.1 system is used in the software prototype portion of this research. In the prototype, more detailed aspects of the ORBeline product are demonstrated including ORB event handling, request filtering, object replication and migration, and inter-ORB communication. See following section.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Distributed Object Application Prototype

This detailed description of the distributed object application prototype is presented in the enhanced version of this report which completes the thesis requirements.

For the development of the software prototype, advanced techniques involving the decomposition of parallel problems were used. These techniques are presented below with special attention placed on the utilization of distributed objects.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Hardware Platforms

The intent of this research is to show the accessibility of parallel programming to those who have access to simple networks. The conventional PCs and workstations available today conform to the von Neumann architecture.[22] In this architecture, the computer is composed of a central processing unit, a control unit, a memory unit, and I/O units. The common PC and workstation networks are basically collections of autonomous von Neumann processors, each with their own memory, interconnected using a variety of network architectures. Both the object-oriented implementation of parallel programming techniques and the inherently portable architecture of CORBA allow this prototype to conform to other hardware configurations including shared memory processors and multidimensional interconnection networks.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Multicomputer

The distributed memory, interconnected processor environment used in the prototype is referred to as the multicomputer. The interconnected processors are referred to as nodes in the multicomputer. The multicomputer used in this prototype was a heterogeneous multicomputer in that various workstations, operating systems, and interconnects were present. The software prototype was fully implemented on this heterogeneous multicomputer. However, because of inadequacies in the load-balancing, task scheduling, and data partitioning, the best results were limited to executing the software prototype in a homogeneous scenario. The details of the this result is explained in detail in Section 4.5.6.

The multicomputer consisted of PCs and workstations at three locations: Villanova University's ECE labs, MRJ Inc., and development facilities at Infonautics Inc. Available platforms included PCs running MS Windows NT 3.51, and SMS SparcStations running SunOS 4.x and Solaris 2.x. These nodes varied in CPU speeds and memory. The three network locations are connected with a "highly-contended" 64KB Internet services. Again, poor results were achieved when executing the prototype across locations for the same reasons mentioned before. The extent of the multicomputer was limited by supported for ORBeline 1.1 development environment. MS Visual C++ 2.2 was used with MS Windows NT on the PCs and GNU GCC 2.6.x for UNIX. As the software prototype first entered development, ORBeline 2.0, a newer CORBA 2.0 compliant ORB was available. However, required compiler upgrades were not available so the older version was used.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Parallel Programming with Objects

Most research involving parallel programming and objects concentrates on the concept of the 'server'.[23] In this context, server refers to an object which encapsulates a running task. Servers are objects in execution.

"servers are relatively self-contained programs that implement one or more tasks of a parallel program. In the server model, a parallel program is a collection of miniature programs running independently, but occasionally interacting with one another by message passing."

The real benefit with using the CORBA architecture for parallel processing is that the above server model is provided by the ORB. The ORB does most of the work for you. In every parallel system design, the programmer must perform the same decomposition of data and functionality into concurrent tasks. This design step is easier with object-oriented analysis for all the usual reasons. However, once the object model is complete for the specific problem, the ORB allows the concurrent solution to be distributed with little effort. The results of this prototype show that the most difficult step in the process was formulating the concurrent tasks for the problem at hand. Once concurrency was achieved in the object analysis, implementing the solution in parallel was trivial. This, of course, was the goal of the research; show that parallel processing can be harnessed with commonplace hardware and software without incurring the expertise of complex network programming. The ORB abstracts distributed processing to the more intuitive server model of concurrent tasks.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Data Partitioning and Domain/Functional Decomposition

As stated above, the most difficult portion of the prototype design was the formulation of the concurrent tasks for the target problem. The goal here is simple; the problem has to be broken into smaller pieces which can be executed concurrently.[24] There are two ways to break up a problem:

ctional Decomposition: realize what functionality from the normal serial execution can be placed in parallel.

Domain Decomposition: realize which data in the system is independent and can therefore be processed in parallel and later combined into a complete solution.

The application of objects here is that the decomposed components described above can be encapsulated in objects; whether the components are function or data makes no difference. Once the components are determined, the granularity of the components is adjusted to improve salability and performance. This step turned out to be the pitfall during the prototype design. Also, the mapping of these component tasks to available CPUs in a efficient manners was just as difficult. The next two sections discuss these problems in detail. For now, it is enough to understand that using the ORB to distribute the parallel components or objects is not the end of the road - intelligent scheduling of these objects on processors with varying performance is required. As stated before, the prototype did not perform properly on the entire multicomputer for this reason. It will be shown that this deficiency was overcome be directing execution to processor sets with equivalent resources.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Manager-Worker Paradigm and Load Balancing

Some problems have regular pattern in the data domain which allow these problems to be decomposed into regularly fitted components. This is the case for large sparse and non-sparse matrices used in linear algebra, computational grids used in flow dynamics and genetic sequences used in biology.[28] There is another class of problems where the exact set of parallel components are not known a priori but rather, the components are calculated during run-time. Combinatorial search algorithms are usually part of this class.[3] The Traveling Salesman and Shortest Path are good examples of these search problems where the search space is dynamically changed during execution.

One solution to the class of problem which have this dynamic behavior is the "replicated worker" or "manager-worker" paradigm. In this case, identical worker processes are assigned to run on each physical processor, and computing tasks are dynamically assigned to the workers as the program runs. The workers are all associated with an abstract managing structure called the "work pool" or "manager". As workers become available and tasks become candidates, tasks are assigned to workers and when workers complete tasks, the work pool or manager is updated. This paradigm maps well into an object model and was used in the prototype. A detailed presentation of the prototype's object model follows the discussion of the combinatorial search problem which is solved in the prototype.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Classic Complex Problems

In Section , a real world combinatorial search problem was presented. The problem there was to decompose complex polygons into triangles for use in a digital mapping database. It turns out that this problem has been documented before as the optimal polygon triangularization problem.[25] Also, the optimal polygon triangularization problem is a more specific version of the generalized matrix chain multiplication problem. The matrix chain multiplication problem is presented in detail below and is also the problem solved by the prototype. The optimal polygon triangularization scenario is shown in Figure 6 where a complex polygon is first dissected into a convex polygon and then triangularized.

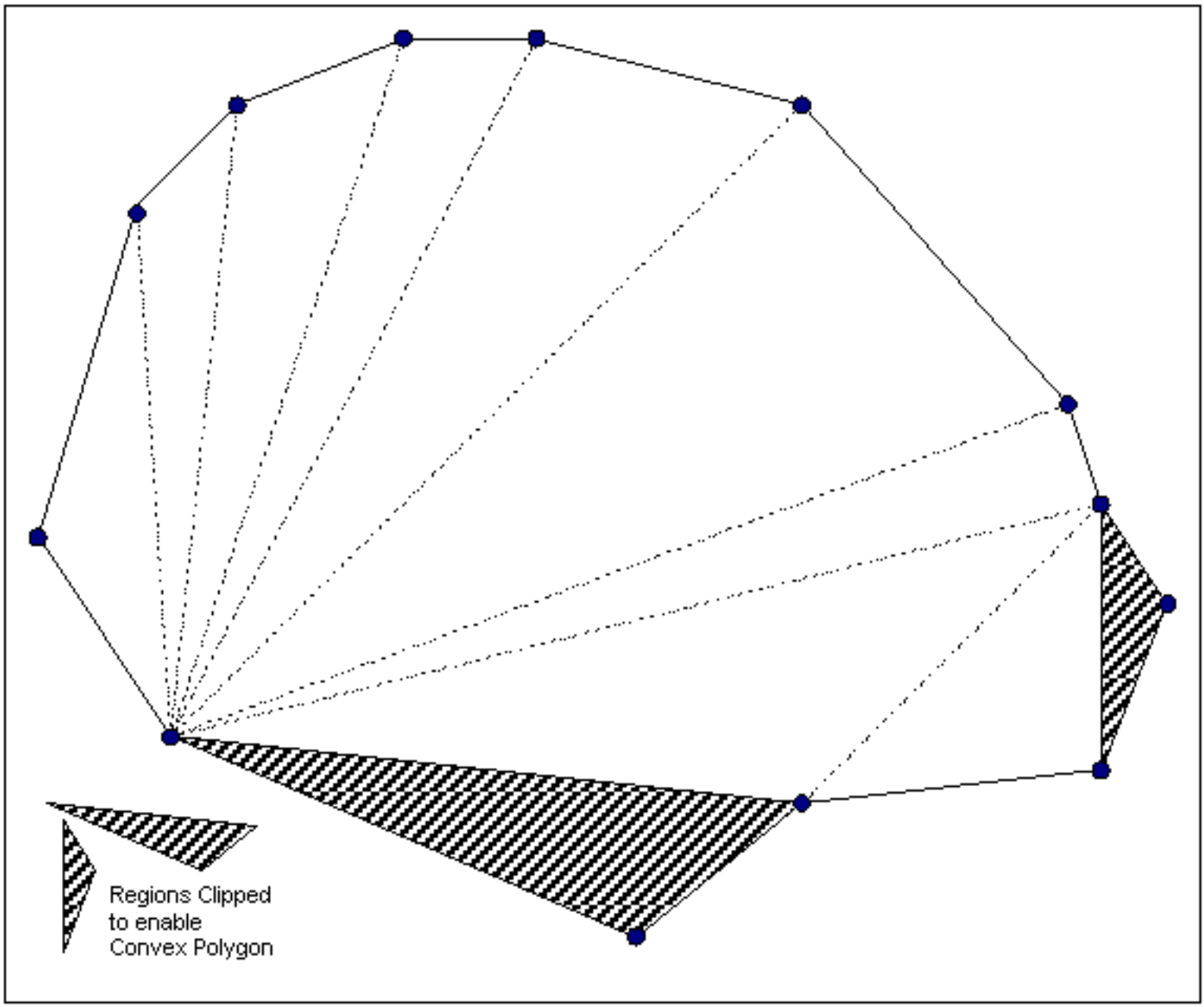


Figure 6 - Example of decomposing convex polygon into component triangles.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Matrix Chain Multiplication

This problem is characterized by a sequence (chain) of matrices $\{ A_1, A_2, ..., A_n \}$ to be multiplied together to form a resulting matrix or product.

$A_1A_2...A_n$

In order for a chain of matrices to be multiplied together, it must be fully parenthesized, that is, it must be either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. Because matrix multiplication is associative, all possible parenthesizations of a matrix chain yield the same product matrix.

Where is the search problem? Consider the simple multiplication of two matrices, A and B. The inner dimensions must be the same. In other words, the number of columns of A is equal to the number of rows of B. If A is a p by q matrix and B is a q by r matrix, the resulting product C is a p by r matrix. The time required to compute C is dominated by the scalar multiplication's of the form (p)(q)(r). For the matrix chain situation, the way the chain is parenthesized makes a considerable difference in how long it takes to compute the product.

Table 4 has dimensions for six matrices. These dimensions are used in the continuing examples. The graph in Figure 7 shows all five (5) parenthesizations of the first four (4) matrices. The graph in Figure 8 shows all fourteen (14) parenthesizations of the first five (5) matrices. In both examples, there is an optimal parenthesization which is less than half the worst case. By picking random values for the matrix dimensions, one can find variations in many orders of magnitude.

matrix	dimension
A_1	30 35
A_2	35 15
A_3	15 5
A_4	5 10
A_5	10 20
A_6	20 25

Table 4 - Example of six (6) matrices

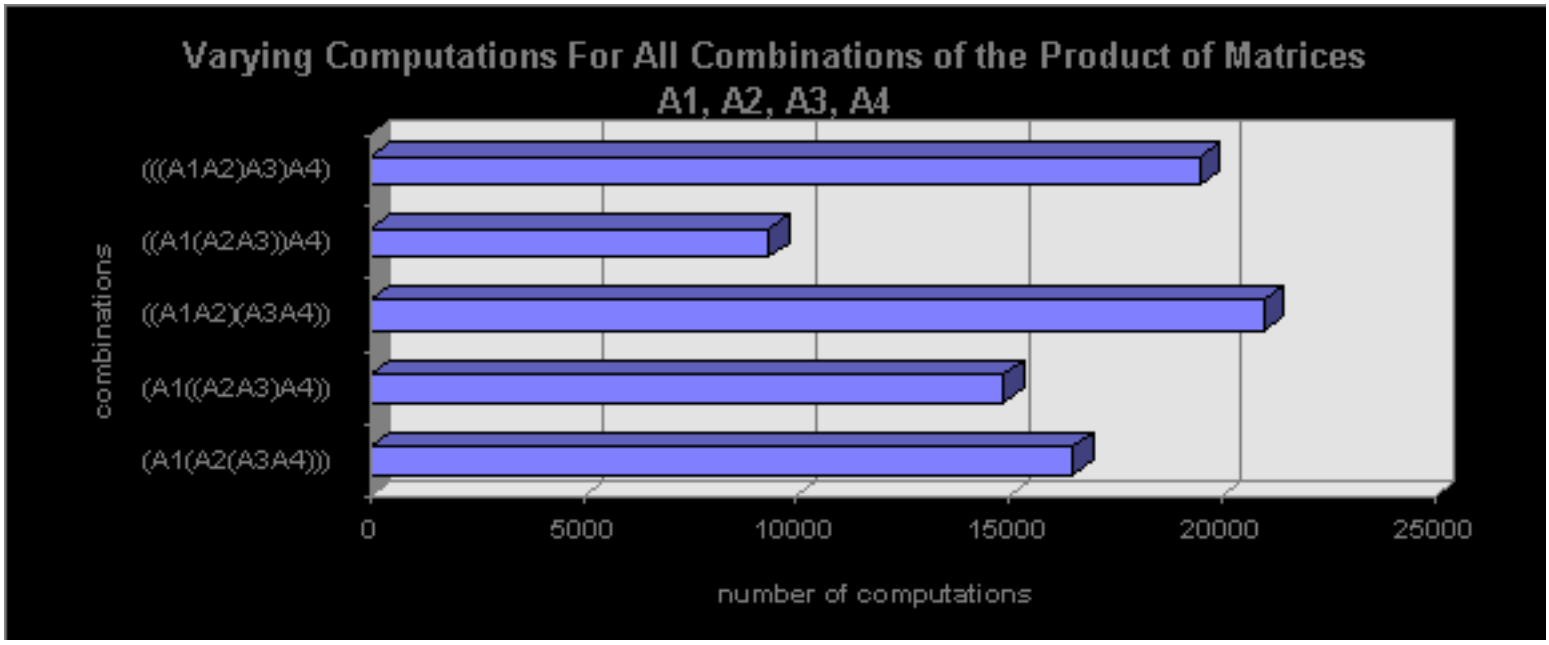


Figure 7 - Bar graph comparing all five (5) combinations of multiplying four (4) matrices.

```
(A1(A2(A3A4)))

(A1((A2A3)A4))

((A1A2)(A3A4))

((A1(A2A3))A4)

(((A1A2)A3)A4)
```

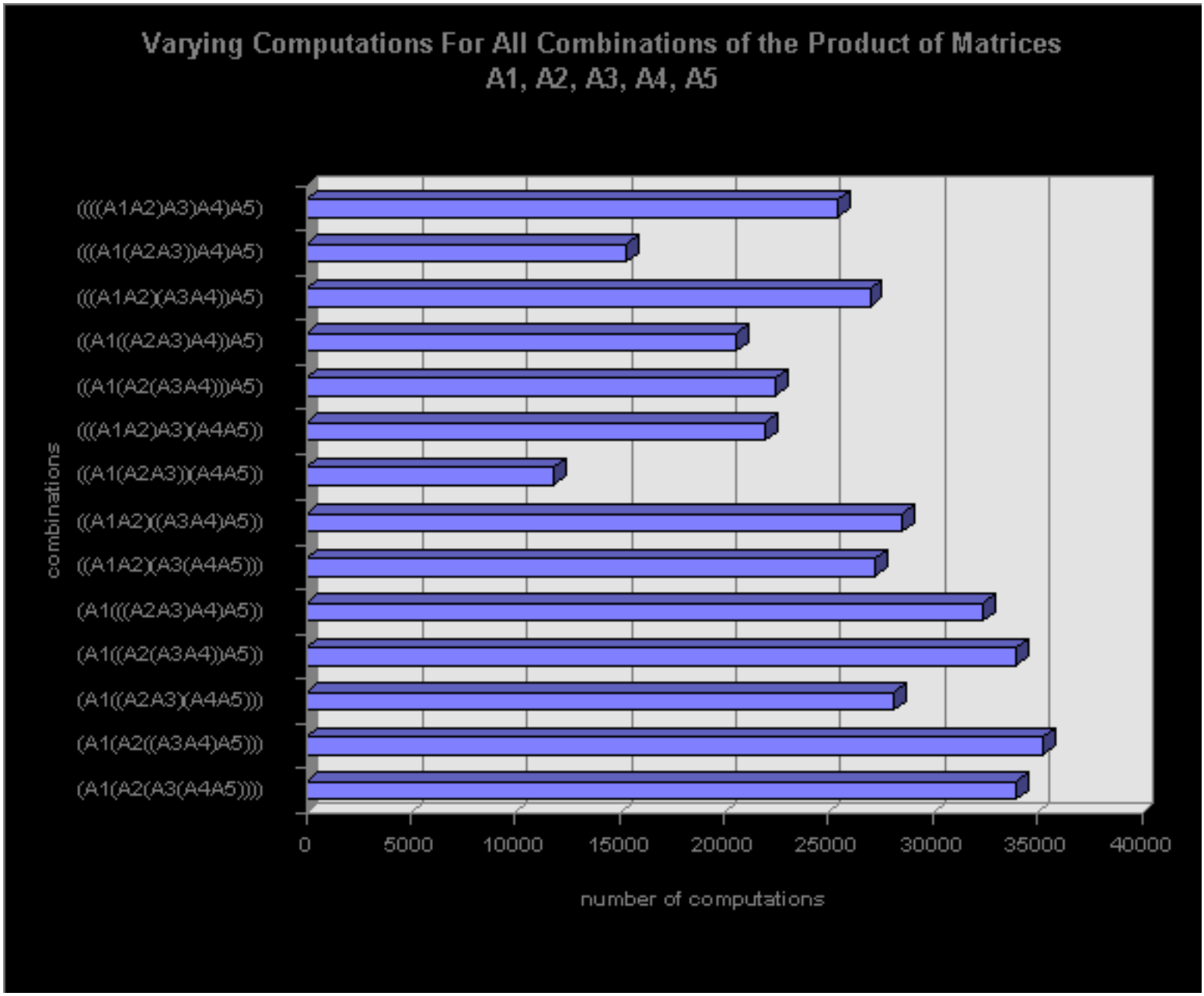


Figure 8 - Bar graph comparing all fourteen (14) combinations of multiplying five (5) matrices

```
(A1(A2(A3(A4A5))))

(A1(A2((A3A4)A5)))

(A1((A2A3)(A4A5)))

(A1((A2(A3A4))A5))

(A1(((A2A3)A4)A5))

((A1A2)(A3(A4A5)))
```

```
((A1A2)((A3A4)A5))

((A1(A2A3))(A4A5))

(((A1A2)A3)(A4A5))

((A1(A2(A3A4)))A5)

((A1((A2A3)A4))A5)

(((A1A2)(A3A4))A5)

(((A1(A2A3))A4)A5)

((((A1A2)A3)A4)A5)
```

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Catalan Numbers

The example of all six matrices is too difficult to show because the number of possibilities is so large (42). In fact, this number grows very fast with the size of matrix chain - just like the case of decomposing polygons. Therefore, an exhaustive search of all possibilities is impossible. The domain of the search space sizes can be characterized as a sequence of Catalan Numbers.

The following graph shows Catalan Numbers up through n=16. If the number of possible parenthesizations of a chain of n matrices is P(n), then P(n) = C(n-1), where C(n) is the Catalan Number for n. Notice that the Catalan Number C(16) is over 9 million.

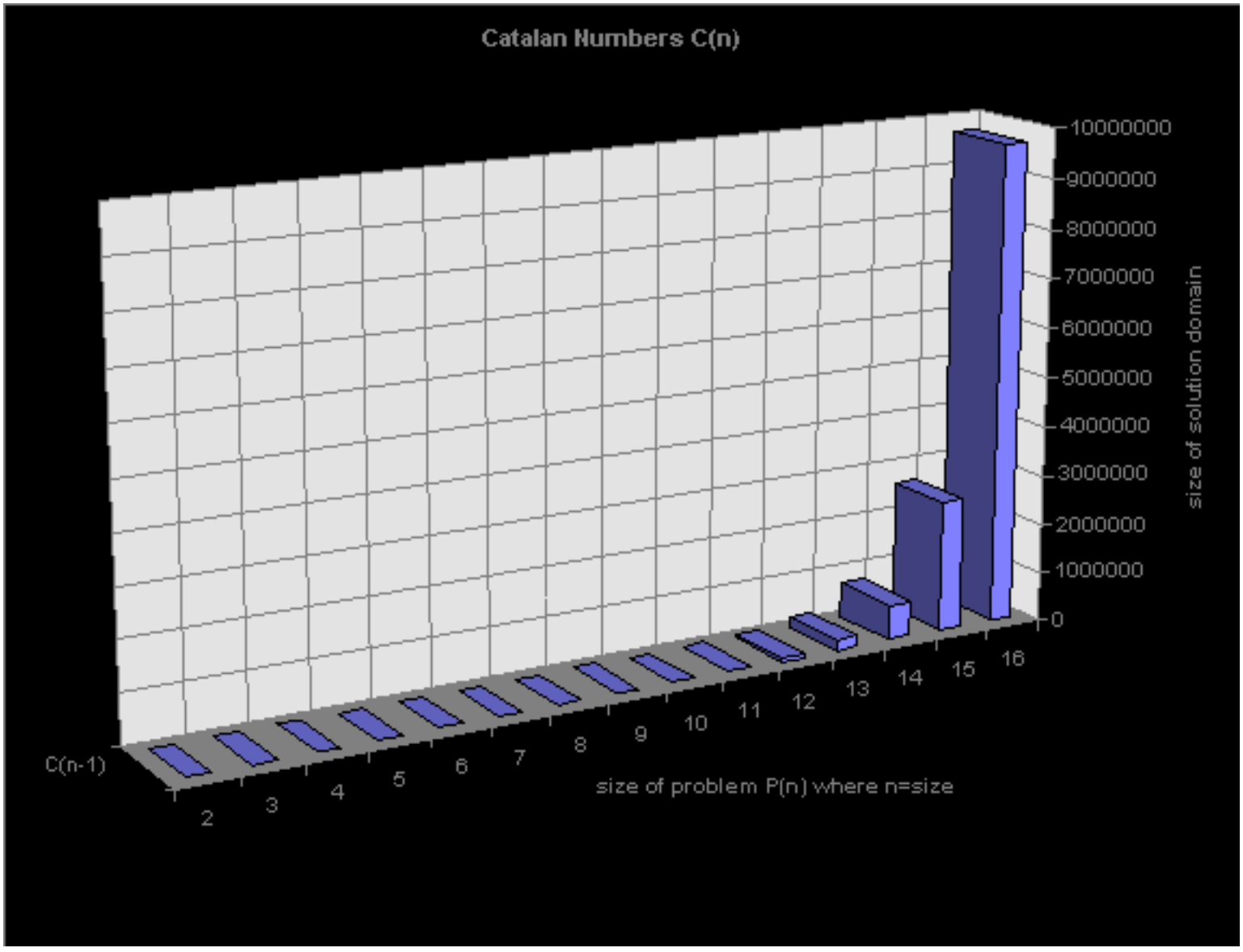


Figure 9 - Graph showing relationship between Catalan numbers and the matrix multiplication search space.

The combinatorial explosion found in the sequence of Catalan Numbers is common for many search problems. There are many techniques and algorithms that produce an optimal result and at the same time, avoid the search space explosion. The technique of dynamic programming is presented by example below and is used in the prototype to determine optimal matrix chain multiplication's for chains on the order or n = {512,1024,2048,4096}

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Dynamic Programming

An optimization problem can be solved by dynamic programming techniques if the structure of the optimal solution reflects optimal substructure and overlapping subproblems. In other words, the optimal solution must exhibit the recursive property of being directly related to solutions to optimal subproblems in which these subproblems encompass subsets of the problem domain.

In addition to the optimal substructure, the domain of subproblems must be small or at least the subproblems should repeat more than once. The matrix chain multiplication problem demonstrates these characteristics. The example which follows uses the data from . In order to find the optimal solution for all six (6) matrices, *m[i,j]* is defined as the minimum number of scalar multiplication's needed to compute the matrix $A_{i..j}$

where $i=1$ and $j=6$

If $i=j$ then the matrix chain consist of just one (1) matrix so there are no scalar multiplication's are required so the value $m[i,j] = 0$ for $i=j$.

If $i<j$ then *m[i,j]* equals the cost of the two (2) subproblems *m[i,k]* and *m[k+1,j]* plus the cost of multiplying the two resulting matrices together. Since computing the matrix product $A_{i..k}A_{k+1..j}$ takes

$p_{i-1}p_kp_j$ scalar multiplication's,

m[i,j] can be generalized *as* $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$.

For each problem *m[i,j]*, the value of *k* is unknown and must be selected from the *(j - i)* possible values in the domain

$k = i, i+1, ..., j-1$

The *m[i,j]* values give the costs of optimal solutions to subproblems. To track of how to construct an optimal solution, *s[i,j]* is defined to be a value of *k* at which we can split the product $A_iA_{i+1}...A_j$ to obtain an optimal parenthesization. That is,

s[i,j] equals a value *k* such that $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$.

The figure below shows a easy way to visualize this data. This table shows the structure of the problem

m[1,6] and all the subproblems, m[1,2], m[1,3], etc. The problem m[1,6] is at the top of the table with the cost of 15125. As an example, the subproblem m[2,5] is highlighted in the darkly shaded square. Notice that this subproblem is made up of the further subproblems to the left and down. These subproblems are defined by the domain of k for m[2,5]. The minimal cost of m[2,5] is found by exhaustively searching all values of k.

m[2,5] = min { m[2,2] + m[3,5] + p1p2p5 = 0 + 2500 + (35)(15)(20) = 13000 ,

m[2,3] + m[4,5] + p1p3p5 = 2625 + 1000 + (35)(5)(20) = 7125 ,

m[2,4] + m[5,5] + p1p4p5 = 4375 + 0 + (35)(10)(20) = 11375 }

= 7125

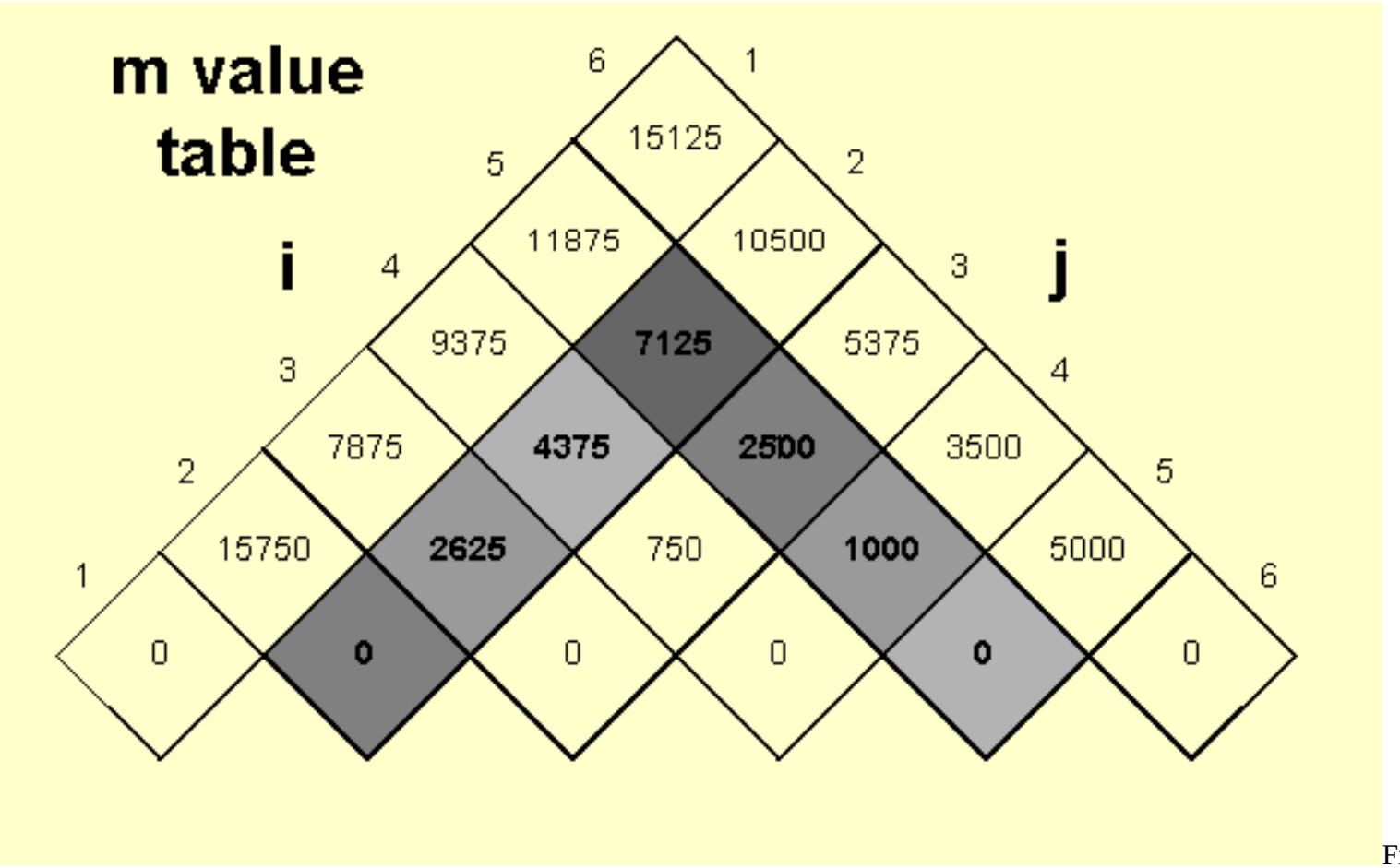


Figure 10 - Diagram of data space of m(i,j) values for example matrix multiplication order problem using dynamic programming.

Figure 11 shows the S table which is structured just like the M table in the above figure. Each entry in the table is indexed by the [i,j] pair. The S table contains the calculated k value for each entry in the M table. The S table can be used to formulate the optimal paraenthesization of the matrix chain. This completes the structure of the problem. The input to the problem is the vector containing the dimensions of the matrix chain. The output is the S table.

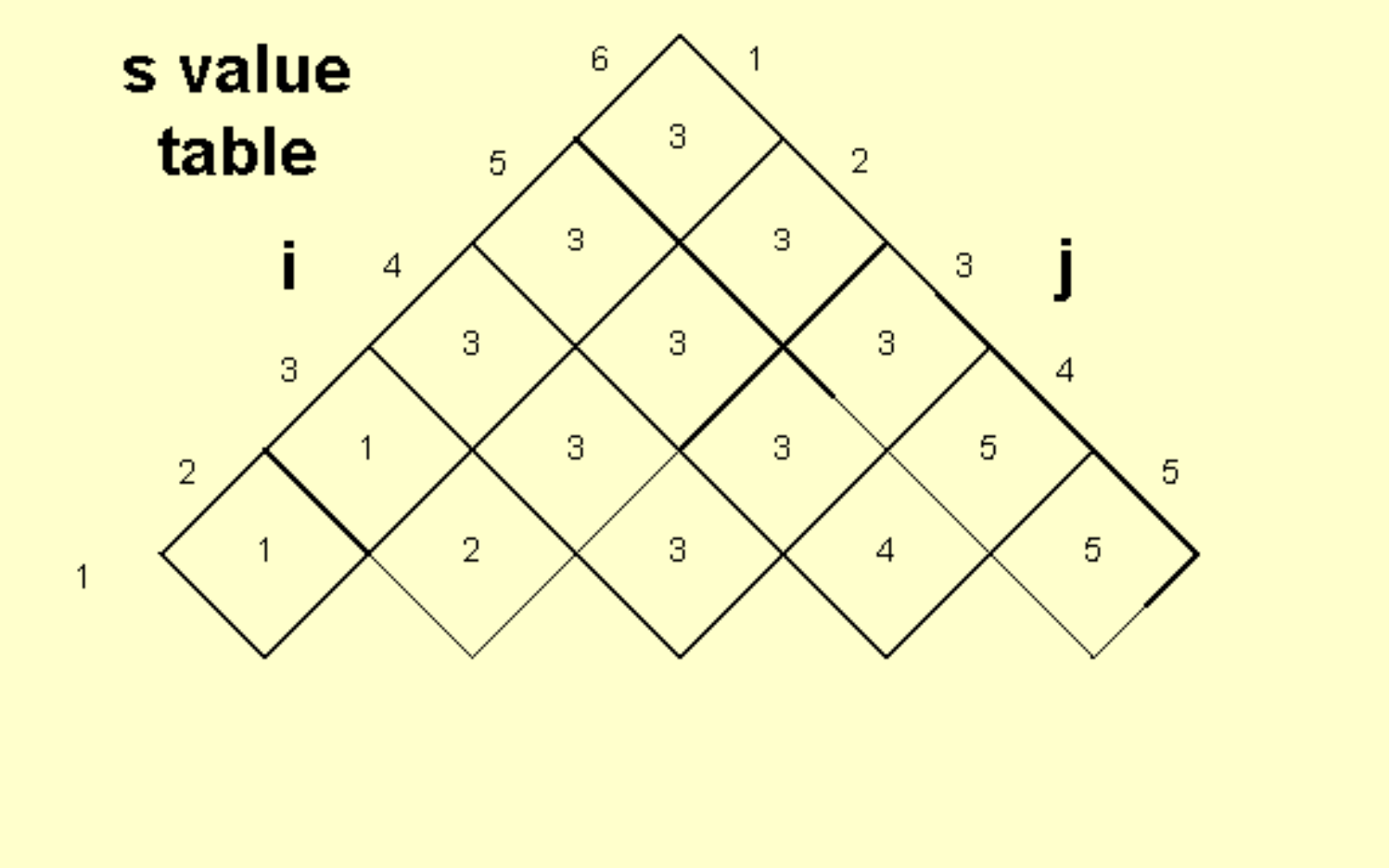


Figure 11 - Diagram of data space of s(i,j) values for example matrix multiplication order problem using dynamic programming

matrix	dimension
A_1	30 35
A_2	35 15
A_3	15 5
A_4	5 10
A_5	10 20
A_6	20 25

It is the recursive nature of the matrix chain problem which allows this problem to be decomposed into concurrent components for use in a parallel implementation. The next section will show the details of the data decomposition used in the prototype.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Prototype

For the purposes of demonstrating the power of distributed objects and the Object Request Broker (ORB), an example of a distributed search algorithm was built using a commercial ORB. The ORBeline System from PostModern Computing (now Visigenic) was used for the prototype.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Data Decomposition

The first step made when building the prototype was to define the software problem by decomposing the structure of the matrix chain problem in a way that is can be implemented using concurrent tasks or objects. A form of data decomposition was performed on the problem structure which allowed for concurrent tasks to be created. Figure 12 shows this decomposition for an arbitrary matrix chain of length n=20. In this case, the ultimate goal of finding the optimal parentheszation of matrices A1..A20 is done by completing the S table for m[1,20] and all its subproblems. Any intersection on the M table m[i,j] cannot be calculated without knowing the answers to the subproblems. This is true for all intersections except for those on the center diagonal. The center diagonal represent all intersections where m[i,j] | i = j are zero and have no subproblems. The structure of the problem revolves around the diagonals of the table. The diagonals represent the size of the matrix chain. For every intersection of the table, the value (j - i) is the size of the matrix chain. Along every diagonal, the matrix chain length is the same. According the form,

$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$.

each entry in the table is dependent upon all the intersections to the left and down. This fact forms the basis of how this problem is decomposed for parallelism.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Corners

In addition to the M table on Figure 12, the data decomposition is overlaid with "corners" labeled 0-15. In this prototype, "corners" refer to the components which are separately calculated. Corners which form a diagonal are totally independent. For instance, corner 0 can be calculated independently from corner 1,2,3, or 4. Once this first corner-diagonal is completed (0,1,2,3,4), then corners 5,6,7, and 8 can be calculated, in parallel, to complete the second corner-diagonal. This decomposition design has a few problem explained as follows.

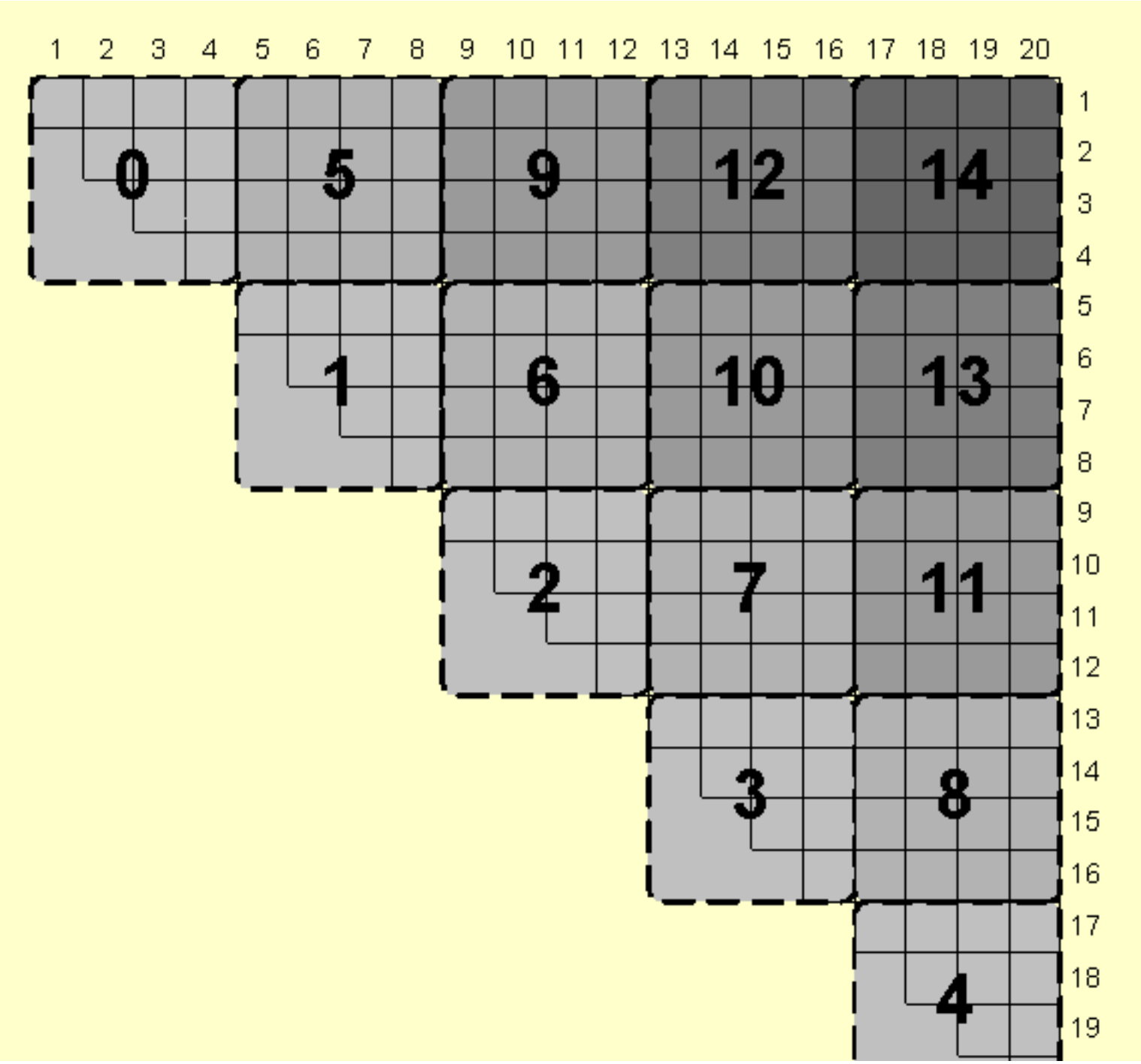


Figure 12 - Data decomposition of dynamic programming data space for matrix multiplication order problem.

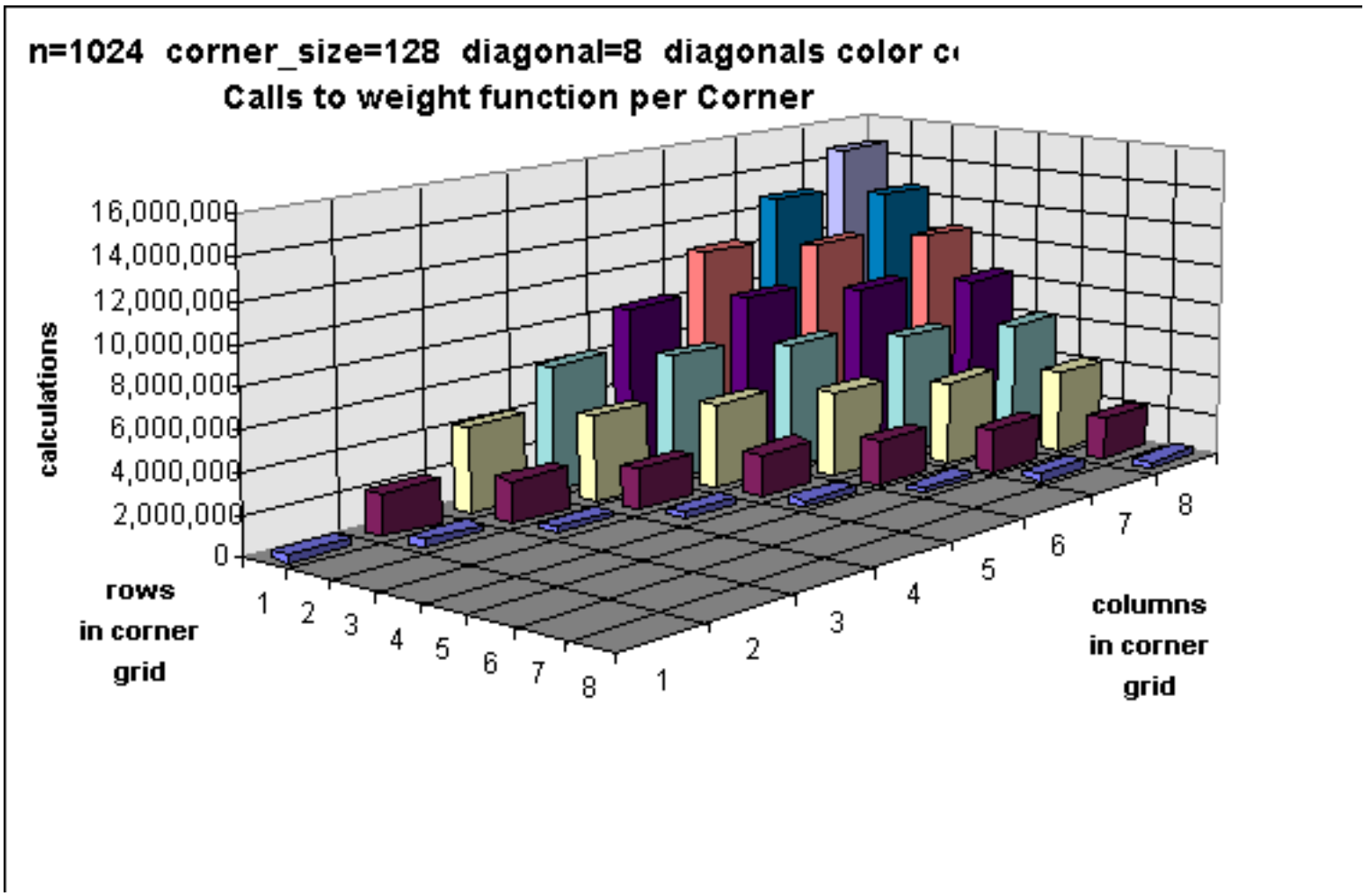


Figure 13 - 3D View of Corner Grid (n=1024, cs=128) showing the number of entries into the weight function per Corner Object.

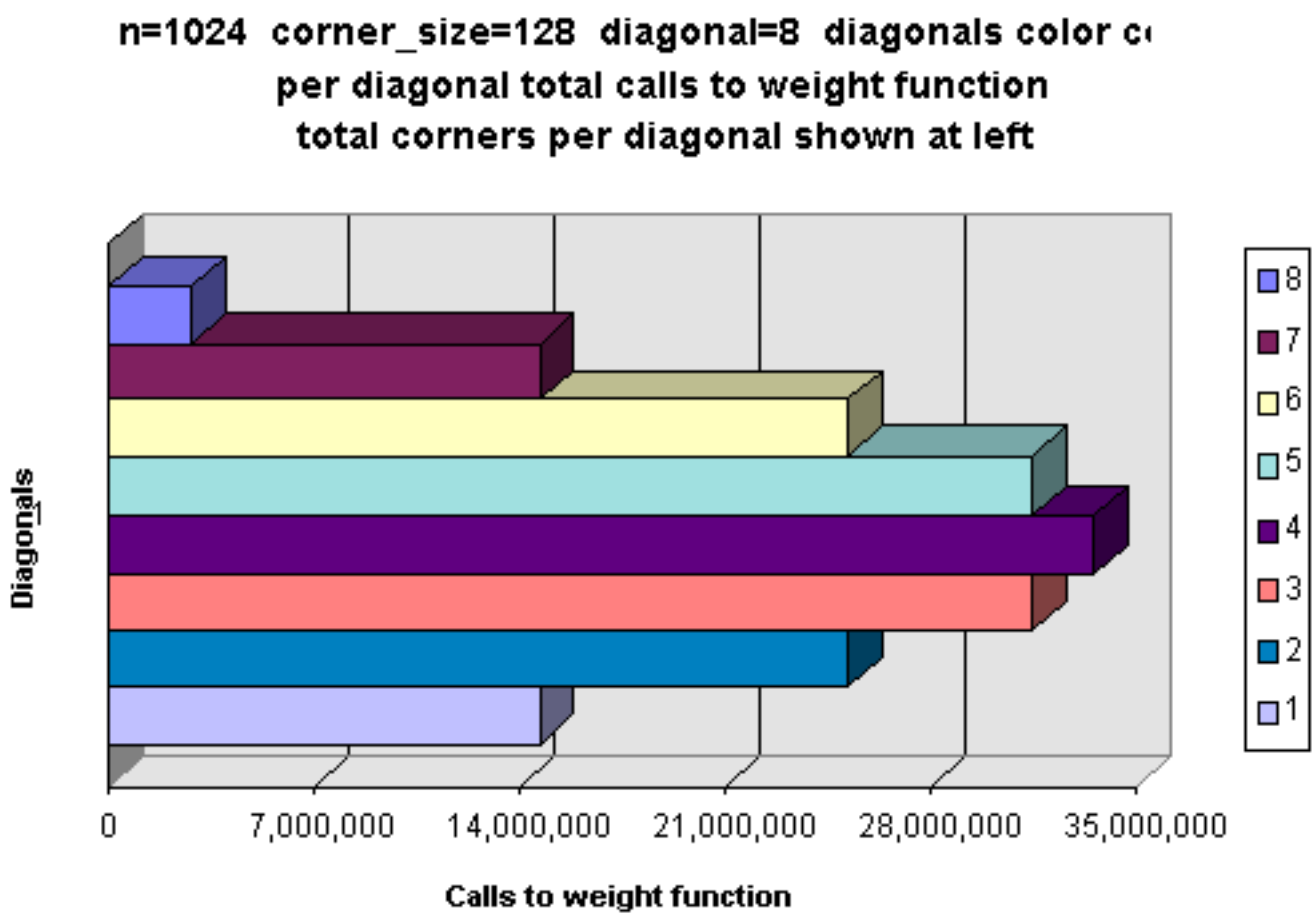


Figure 14 - Bar graph of the number of entries into the weight function per diagonal in the Corner Grid (n=1024, cs=128)

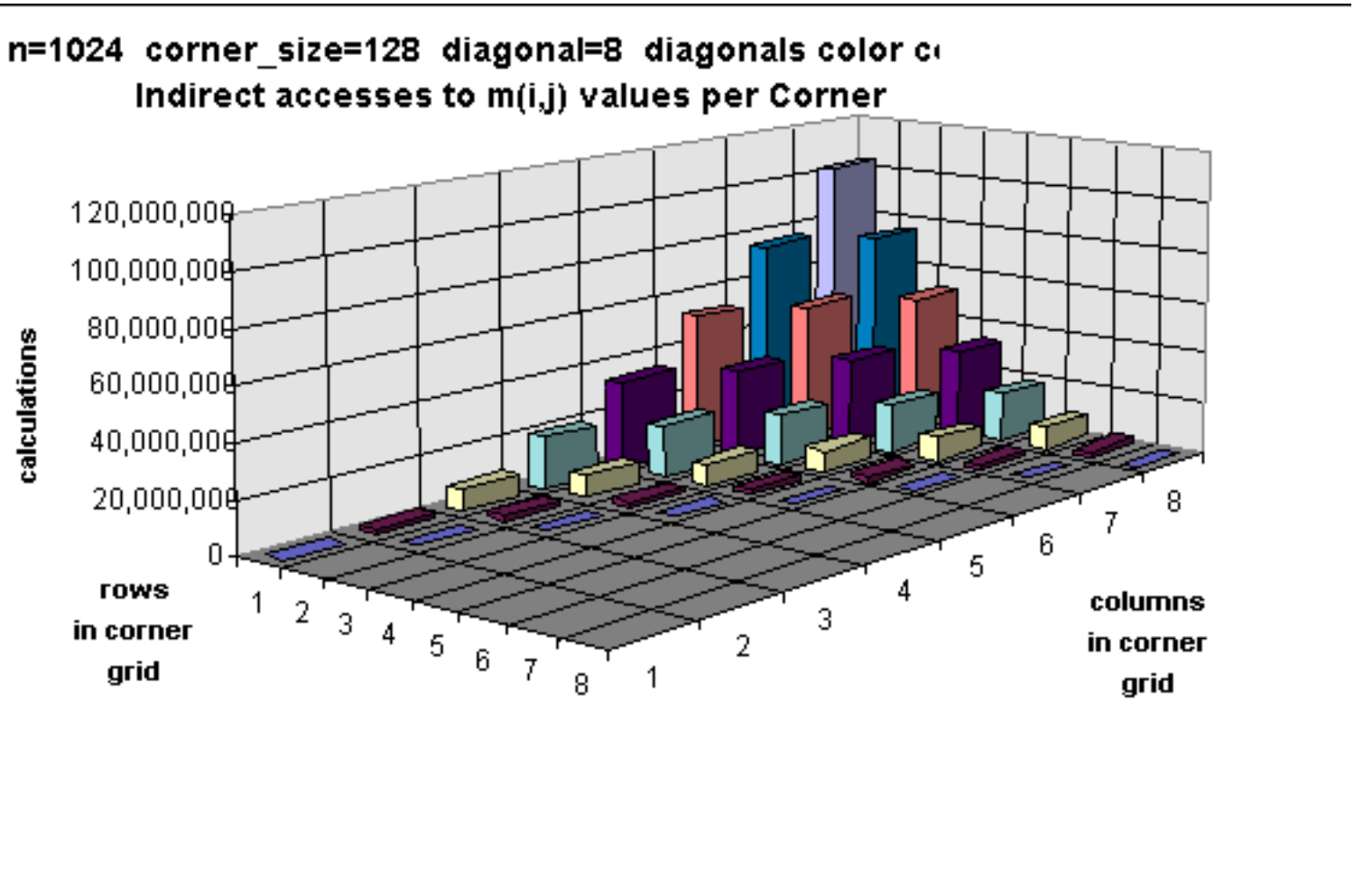


Figure 15 - 3D View of Corner Grid (n=1024, cs=128) showing the number of indirect accesses to m(i,j) values per Corner Object.

This page was last updated on February 18, 1997
Please send comments or questions to [Matthew Stevens](#).

Limitations

One limitation of the decomposition design in is that only one corner-diagonal can be calculated at a time. This is a problem because if the corners on a diagonal are each assigned to a processor for calculation, then the last corner to complete will delay the other unallocated processors. Also, the level of parallelism decreases from the center diagonal so that the final diagonal which has only on corner can only be calculated by one (1) processor. This limitation is compounded by the fact that this same corner is the most expensive to calculate. Figure 13 introduces an example of an arbitrary matrix chain of length n=1024. In this example, the corner size is fixed at 128 so that there are eight (8) corners along the first center diagonal. The diagonals are color coded through this graph and the next two. Figure 13 shows the number of calls to the weight function per corner to determine the costs, complete the corner, and fill in the S table. Notice that the number of calls per corner increase linearly between diagonals. This can be seen also in a different of the same data in Figure 14. Figure 15 shows a further limitation of the design. To understand this limitation fully, the software design must be explained first.

Caching Corner Data

In the software design, the corner data and behavior has been encapsulated in a distributed object. Corner objects are managed by Worker objects where one worker object is targeted to be executed on processor at a time. The Manager object directs all the Worker Objects and also manages a catalog of Corners already calculated. When a Corner needs to be calculated, if it is not part of the center diagonal, it requires values previously calculated in the corners to the left and down. Since these Corners might have been calculated on another processor, the remote corner data can be retrieved per method invocation on remote Corner object. This retrieved data is "cached" locally in the so that the current Corner can access data when needed. In this way, the Worker assigned the task offering the final Corner will have to have all other corner data cached. The replication of data is relatively negligible with respect to processing time. Memory could become a problem when n>4096. The amount of space required to store the corner data is on the order of n^2. The processing time is on the order n^3. The limitation shown in Figure 15 describes the compounding of the processing time due to the caching design and decomposition of the data in corners. Since the M table data is not captured in a single array but rather in a tree-like data structure where the nodes in the tree are Corner objects, accessing data in distant corners is expensive.

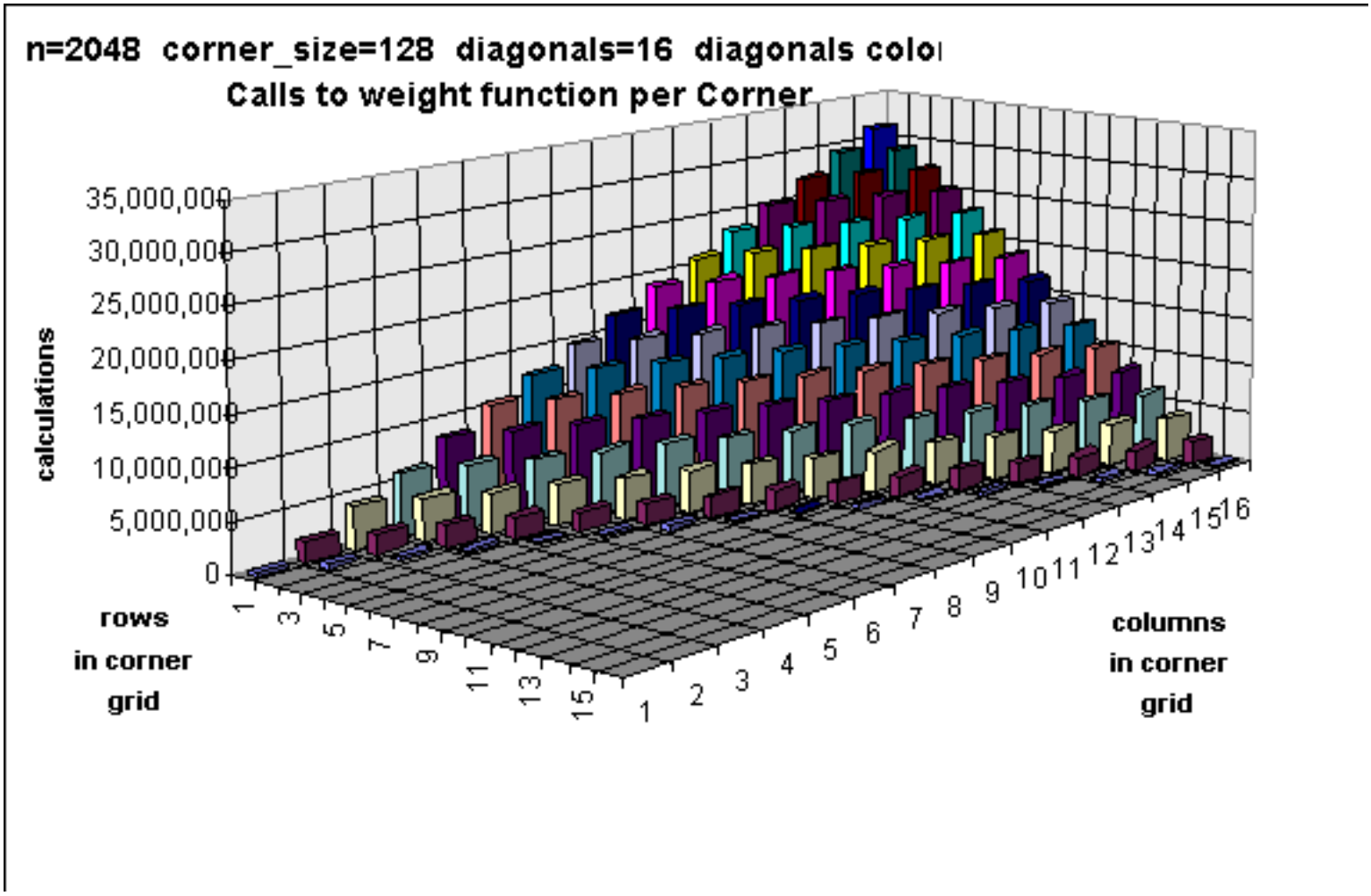


Figure 16 - 3D View of Corner Grid (n=2048, cs=128) showing the number of entries into the weight function per Corner Object

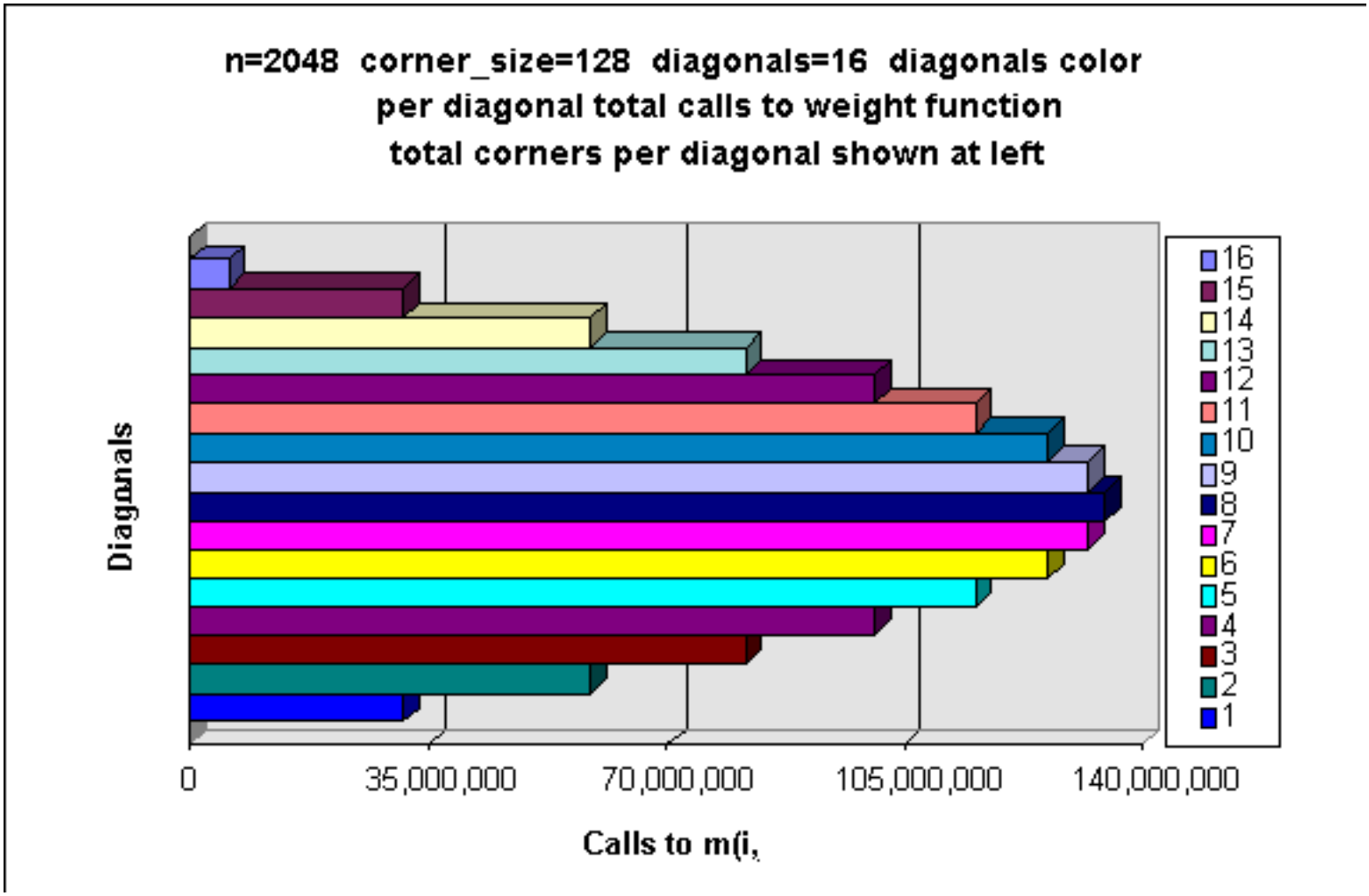


Figure 17 - Bar graph of the number of entries into the weight function per diagonal in the Corner Grid (n=2048, cs=128)

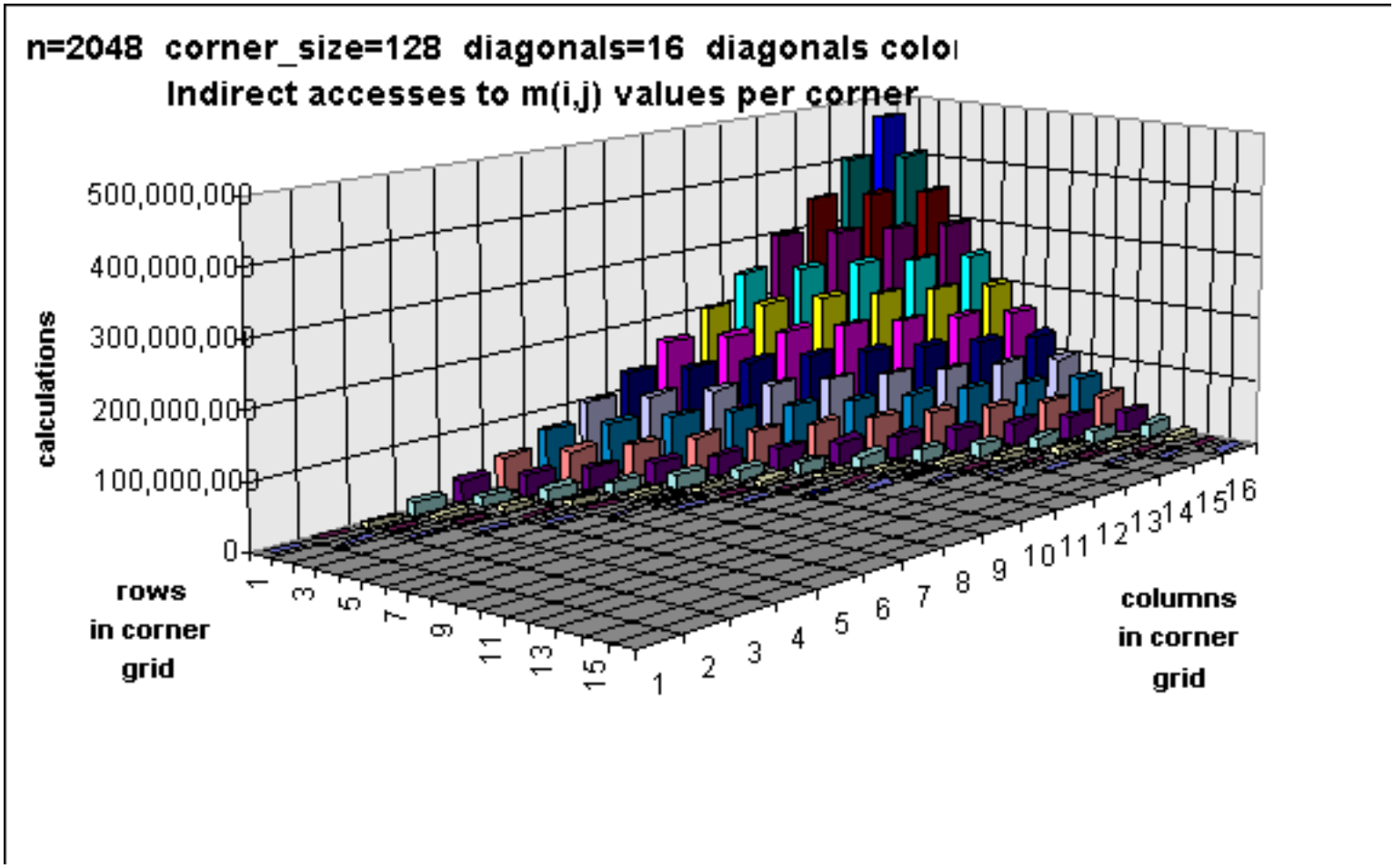


Figure 18 - 3D View of Corner Grid (n=2048, cs=128) showing the number of indirect accesses to m(i,j) values per Corner Object

The following code block shows the extra processing which is required to access local M data from related Corners. This function getMValue() will recursively call into Corners which are left or down from the current Corner. Also, if the desired data is in the current Corner, access must be made on the shared memory region. The data modeling is not at fault here. The performance problem can be solved using a better data structures.

```
MValue getMValue(MDataIndex across,MDataIndex down) // Real World Coordinates [1 N]

{

if(across < (the_i_value + 1 - the_corner_size)) // in left corner

return(the_left_corner->getMValue(across,down));

else if(down > (the_j_value + the_corner_size)) // in down corner

return(the_down_corner->getMValue(across,down));
```

```
else // value in THIS corner

if(the_busy_corner == this) // is THIS corner busy calculating

return(getshmMValue(across,down));

else

return((*the_M_data_ptr)[REAL_2_CORNER_SEQ_NUM(across,down,the_corner_size)]);

}
```

Figure 16, Figure 17, and Figure 18 show similar information for a matrix chain problem where n = 2048. The limitations are more evident in these graphs.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

CORBA Implementation

As show above, a object model consisting of Problems, Solutions (manager), Worker, and Corner objects was used. See Appendix A for all detailed code listings. The scenario for processing was very simple. The simplicity of the design shows the usefulness of the ORB architecture for doing parallel processing.

- Problem objects are interfaces to the source and destination data for a particular problem. In this prototype, only the Matrix Multiplication Problem was implemented. The model could be further expanded by deriving other specialized problem types like Traveling Salesman or Shortest Path.
- The Problem, Solution, and Worker objects are abstracted to the level of processes.
- The Problem, Solution, and Worker objects do not have to be collocated on the same processor but they can be in the same process. This fact was helpful during testing because the prototype could be runs as a single process to make initial debugging easier.
- The Worker objects were required to be in separate processes and on separate processors if parallel processing was to be harnessed.
- The Solution and Worker objects can also be specialized further to expand the capabilities of the prototype.
- Using the event driven capabilities of the ORBeline System, the objects can be started and stopped in any order. Workers can be added to the running problem as needed. New problems can be started when required. If any part of the system fails at either the O/S or hardware level, replicated objects and take over.
- Each object becomes part of the distributed logging sub-system to track the whole process.
- The order of events is as follows (nominally):
    1. The Logger object is started from the command line and its output is redirected to a file or viewed on the standard output. The Logger object registration is made global in the ORB (other objects can "bind" to it)
    2. The Solution object is started with its object name and corner size on the command line. The Solution object binds with the Logger. The Solution object enters its event loop waiting for remote method invocations and special ORB events.

```
The Worker objects are started with their object names and solution provider interface name which
they intend to connect to. The Worker objects registration is local and they bind with the remote
Logger. They try to bind to the remote Solution object or do a differed binding if needed. After
they successfully bind to the remote Solution object they call the "volunteer()" method to
register their availability.
```

1. The Problem object is started with its object name and any input parameters for the problem data. The Problem object's registration is local and it binds with the remote Logger. The Problem then tries to bind with the Solution object or do a differed binding if needed. After it successfully binds to the remote Solution object it calls the "getHelp()" method to register its problem. The input

data (matrix chain) is sent to the Solution object at this time.
2. Once the Solution object has a specific problem in hand, it will assign tasks to current volunteered Worker objects and to future Worker objects when they volunteer in the future. The task assignment relays the input data (matrix chain) to the Workers and sets the corner size.
3. The Worker objects will initialize its data structures for the current tasks by setting up its local Corner objects - all with empty M data. The Worker object will call the "taskReady()" method on the Solution object to let it know that the Worker is ready for calculating.
4. The Worker object is totally event driven on "taskReady()" messages. While there are more Corners to be calculated, Workers will be assigned Corners to calculate. The Solution object calls "getCorner()" method on a worker to retrieve the object reference to the remote Corner object.
5. The "getCorner()" method invocation is a blocking call - the caller can not handle other requests or events while its blocking. During the time, the Worker object directs its specified Corner object to cache any M data it needed at this time. A list of owners of already calculated M data is passed with the "getCorner()" method.
6. The Corner Object will make calls on remote Corner objects with the "getMData()" method. This call also blocks but is handled with concurrent processing. See Singlethreaded or Multithreaded Servers Section 4.5.4.1 and Shared Memory Section 4.5.4.2.
7. With this Corner object reference, the Solution object calls "calcData()" on the remote Corner object.
8. When the Corner object receives the "calcData()" request, the calculation is performed concurrently so that subsequent "getMData()" requests can be handled.
9. When a Corner object completes its calculation, it sends a "cornerComplete()" message to the Solution object and sends the resulting data to the Problem object.
10. When all the Corners are complete, the Solution object calls "endTask()" method on the Worker objects. At this point the Worker objects may re-volunteer.

Singlethreaded or Multithreaded Servers

At Object Expo 1995 in New York, Doug Schmidt, editor of the C++ Report, described a potential problem with non-multithreaded implementations of CORBA. The problem was that of deadlock in a distributed system. Deadlock occurs when object A makes a blocking request on a remote object B, which, in turn, makes a second blocking call on object A. This second call will not return because the object A is isolated/busy in the first blocking call. Doug Schmidt recommended a multithreaded implementation where a given object server could be used concurrently by more than one client object.

This is precisely the problem with the prototype faced. In lines 7, 8, & 9 above, the situation is described. In this case, a Corner object A may be trying to cache M data by requesting this data from a remote Corner object B. If another Corner object B' in the same Worker process is busy calculating data then ORB can neither deliver the request nor can the Corner object B service it.

The ORBeline System provided for full support for multithreaded objects with out any additional work other than linking executables with reentrant ORB libraries. However, this support was not available for

the SunOS operating system - just Solaris and NT. Also, I was unable to get the multithreaded version running on Windows NT. Instead, a design was used where Corner calculations would be done concurrently using Solaris and NT threads and for SunOS, standard heavy-weight processes with fork() system call. When a Worker process was interrupted with a request for a resident Corner object to "calcData()", a new thread or process would be started to perform the CPU intensive task while the rest of the Worker process could service "getMData()" requests. This lead to further design problems during thread completion determination. The forked process would send a SIGCHLD UNIX signal to the parent process when the calculating process completed. However, the Solaris and NT thread implementation required polling of the thread status to determine completion. Also, the forked process under SunOS needed a shared memory construct to pass the completed corner data back to the parent process. The shared memory implementation would have to be done in such a fashion so that as much of the code could be shared between Solaris, NT and SunOS.

Shared Memory

For the shared memory implementation, memory mapped files were used. As and example, when a one megabyte buffer was needed:

1. A temporary file was created and opened on /tmp file system.
2. The file was accessed out to the one megabyte point using lseek() system call and then a NULL "\0" was written. This increase the size of the file to one megabyte and by definition, cleared the values to NULL.
3. The mmap() system call mapped the file into the address space of the calling process and a pointer to the beginning of the buffer was returned. This buffer was inherited by the forked process and the changes made to this buffer in the child process were seen by the parent process.

The combination of the shared memory and concurrent process/thread design was successful although cryptic. This design also put some constraints on the rest of the system. The Solution object now required some state information about Corners and Workers. Also, only one (1) worker could be caching data at a time. This was the price to pay for portability and operation in a heterogeneous environment.

Distributed Logging

One of the most important requirements of a distributed system is logging - knowing what is going on in a vast system. In a distributed system, you don't have the ability to write to the standard output or read the system console. A time order sequence/stream of events in the whole system is needed. For this prototype, a Logger object was used which had one (1) method - "logMessage()" which accepted a single CORBA string as an argument. To enhance the information in the message, the Logger object multiply inherited from both its IDL implementation class and the Server Event Handler class which came with the ORBeline System. This allowed call backs to be assigned to almost every event that the Logger "server" object encountered.

- object bind/unbind
- method invocation

In this way, an object which binds to the Logger object has that event logged. If binding to the remote Logger object is made common to the constructors of all objects (Solution, Workers, Corners, Problems), then the whole prototype can be monitored. Also, with the per method callback, the "principal" information of the bound object is accessible (hostname, process id, interface name, object name, username, password, etc.). This allows for detailed logging with most of the effort being made under the covers. See Appendix B for a detailed look at a prototype log. These logs provided all the data for the rest of the graphs in this report.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Calculation Algorithm

The following code block shows the crucial parts of the MatrixMultCorner class that deal with concurrent process creation and calculating the M data. The portions of the code which are different for NT, Solaris, and SunOS are separated by the preprocessor directives. Notice the two entry points for the threads in NT and Solaris. Also, there is specialized processing for the Corner object which are on the center diagonal.

```
#ifdef WIN32

DWORD WINAPI MatrixMultCorner::_thread_entry(LPVOID thread_parm)

{

MatrixMultCorner::get_busy_corner()->_calcData();

return(0);

}

#endif

#ifdef SOLARIS

void* MatrixMultCorner::_thread_entry(void* thread_parm)

{

MatrixMultCorner::get_busy_corner()->_calcData();

MatrixMultCorner::the_thread_done = 1;

return(0);

}

#endif
```

```
void MatrixMultCorner::_calcData(void)

{

unsigned diagonal_count;

unsigned start_point;

unsigned end_point;

unsigned i_start_point;

unsigned i_end_point;

unsigned j_start_point;

unsigned j_end_point;

unsigned i,j;

if(the_corner_size != (the_i_value - the_j_value + 1)) // non-center diagonal

{

diagonal_count = 0;

start_point = the_i_value - (2 * the_corner_size) + 3 - the_j_value;

end_point = the_i_value + 1 - the_j_value;

i_start_point = the_i_value + 2 - the_corner_size;

j_end_point = the_j_value + the_corner_size;

for(unsigned diagonal = start_point;

diagonal<=end_point;

diagonal++,diagonal_count++)

{
```

```
if(diagonal_count < the_corner_size) // 0..(CS-1)

{

i_end_point = i_start_point + diagonal_count;

j_start_point = j_end_point - diagonal_count;

}

else

{

i_start_point++;

j_end_point--;

}

for(i = i_start_point,j = j_start_point;

i<=i_end_point;

i++,j++)

{

PValue p0p2 = getPValue(j-1) * getPValue(i);

MValue Mcache = 3999999999;

SValue Scache;

for(unsigned k = j;k <= (i-1);k++)

{

MValue q = getMValue(k,j) +
```

```
getMValue(i,k+1) +

p0p2 * getPValue(k);

if(q < Mcache)

{

Mcache = q;

Scache = k;

}

}

setshmMValue(i,j,Mcache);

setshmSValue(i,j,Scache);

}

}

}

else // center diagonal

{

diagonal_count = 0;

start_point = 2;

end_point = the_corner_size;

i_end_point = the_i_value + 1;

i_start_point = the_i_value + 3 - the_corner_size;

j_start_point = the_j_value + 1;
```

```
j_end_point = the_j_value + the_corner_size - 1;

for(unsigned x=(the_i_value - the_corner_size + 2);

x<=(the_i_value + 1);

x++)

{

setshmMValue(x,x,0);

}

for(unsigned diagonal = start_point;

diagonal<=end_point;

diagonal++,i_start_point++,j_end_point--)

{

for(i = i_start_point,j = j_start_point;

i<=i_end_point;

i++,j++)

{

PValue p0p2 = getPValue(j-1) * getPValue(i);

MValue Mcache = 3999999999;

SValue Scache;

for(unsigned k = j;k <= (i-1);k++)

{
```

```
MValue q = getMValue(k,j) +

getMValue(i,k+1) +

p0p2 * getPValue(k);

if(q < Mcache)

{

Mcache = q;

Scache = k;

}

}

setshmMValue(i,j,Mcache);

setshmSValue(i,j,Scache);

}

}

}

}

void MatrixMultCorner::calcData(const CornerList& corner_list)

{

MatrixMultCorner::set_corner_list(new CornerList(corner_list));

prepare();

MatrixMultCorner::set_busy_corner(this);

#ifdef WIN32
```

```
#ifdef SINGLE_THREAD

MatrixMultCorner::_thread_entry(NULL);

handle_calc_completion();

#else

MatrixMultCorner::the_thread_handle =

BEGINTHREADEX(NULL,0,

MatrixMultCorner::_thread_entry,

NULL,0,

&the_thread_id);

LOG_MESSAGE("Thread Created to handle corner calculation");

Dispatcher::instance().startTimer(CORNER_TIMER_INTERVAL,0,this);

#endif

return; // ending blocked request to client prevents deadlock

#endif

#ifdef SOLARIS

if(-1 ==

thr_create(NULL,NULL,

MatrixMultCorner::_thread_entry,

NULL,THR_BOUND,

&(MatrixMultCorner::the_thread_id)))
```

```
{

perror("Thread Create Failed");

exit(-1);

}

MatrixMultCorner::the_thread_done = 0;

LOG_MESSAGE("Thread Created to handle corner calculation");

Dispatcher::instance().startTimer(CORNER_TIMER_INTERVAL,0,this);

return; // ending blocked request to client prevents deadlock

#endif

#ifdef SUNOS

signal(SIGCHLD,MatrixMultCorner::child_signal);

if(-1 == (MatrixMultCorner::the_process_id = fork()))

{

perror("Failure on fork()");

exit(-1);

}

else if(0 != MatrixMultCorner::the_process_id)

{

LOG_MESSAGE("Child Process Created to handle corner calculation");

return; // ending blocked request to client prevents deadlock

}
```

```
else

{

//Dispatcher::instance(new Dispatcher);

MatrixMultCorner::_calcData();

exit(0); // signal parent that calculation done

}

#endif

}
```

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Results

This section describe the final results of the prototype, recommendations for improvements, and some presentation graphs. All results are based on average runs of test cases for four (4) problem sizes n = (512,1024,1536,2048). For each of the four (4) tests, three (3) different scenarios were executed and compared. The final results are shown in Figure 19.

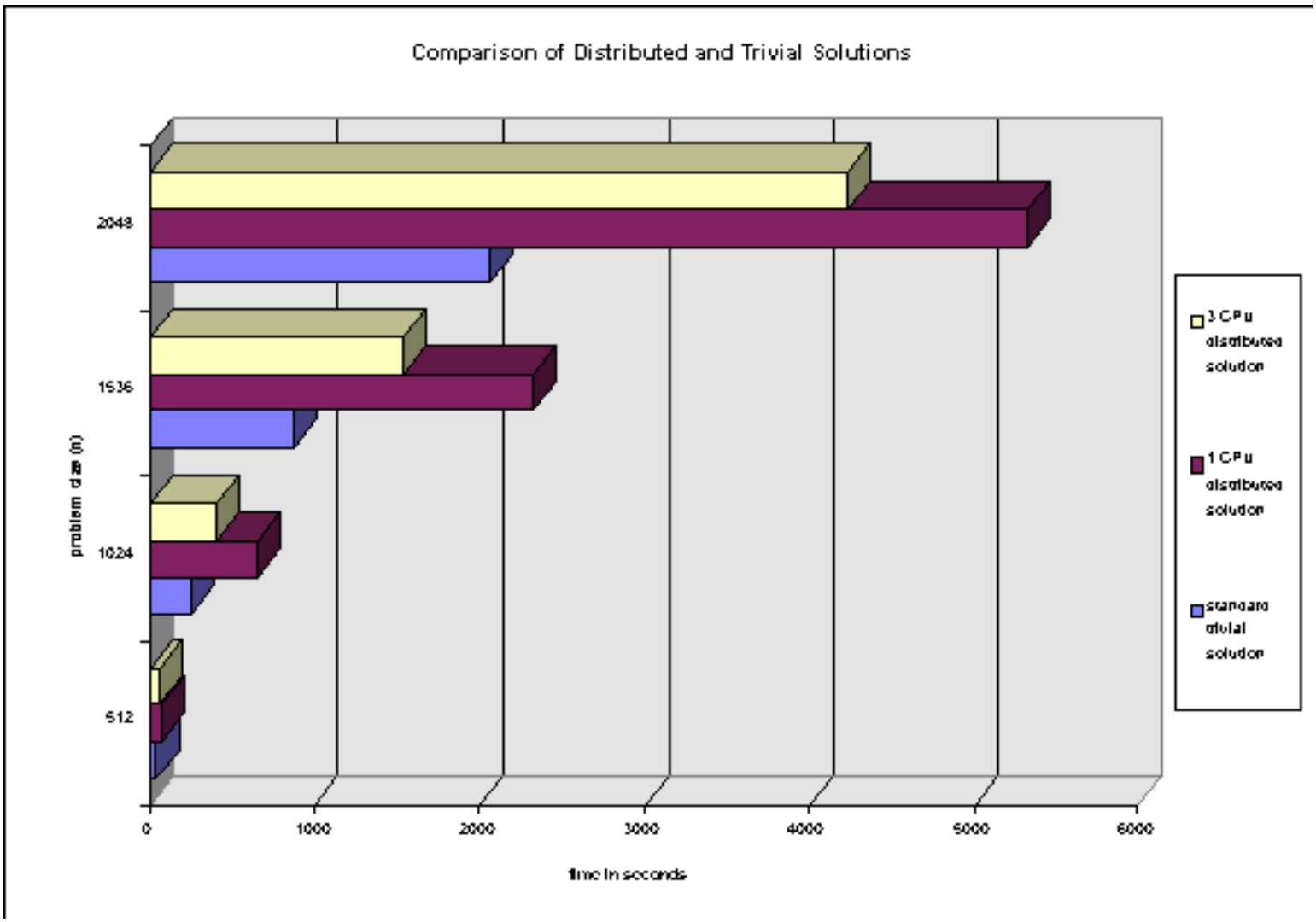


Figure 19 - Bar Graph comparing performance of distributed and trivial solutions to the matrix multiplication order problem.

1. Solve the problem using a direct C/C++ mapping of the textbook dynamic programming algorithm in a single UNIX process. (shown in cyan)
2. Solve the problem using the ORB prototype with a single Worker object with corner size = n (one process, one Worker, one Corner, one problem). This scenario shows the overhead associated

with the prototypes data structure and C++ coding. (shown in magenta)
3. Solve the problem as above but with 3 CPUs (three Worker objects) and the empirically derived optimal corner size (number of corners)

The best results were for the test where n=2048 and corner size=256. Here, a 40% efficiency was achieved. The research review of published results in similar prototypes was on the order of 80%-90% efficiency. I believe this discrepancy is due to three (3) factors.

1. The published values were achieved using prototypes built on single purpose parallel processing hardware with multidimensional interconnections which would tend to have greater efficiency than this prototype's experimental multicomputer.
2. The published prototypes utilities a more viable data decomposition which changed during execution to achieve greater parallelism during the entire run. This prototype's parallelism dropped off during final Corner calculations which took the most CPU resources.
3. As described before, this design had limitations in data structures and overhead due to heterogeneous requirements.

Figure 20, Figure 21, Figure 22, and Figure 23 show stacked bar graphs of the completion times of respective Corner objects assigned to processors. Each Graph has the 3 CPU and single CPU solution. In all cases, it is evident that the most detrimental aspect to this design was the drop off of parallelism during the last phases of the Corner calculations. The longest stacked bar denotes problem duration. All tests were completed on the SMS Ultra Sparcs.

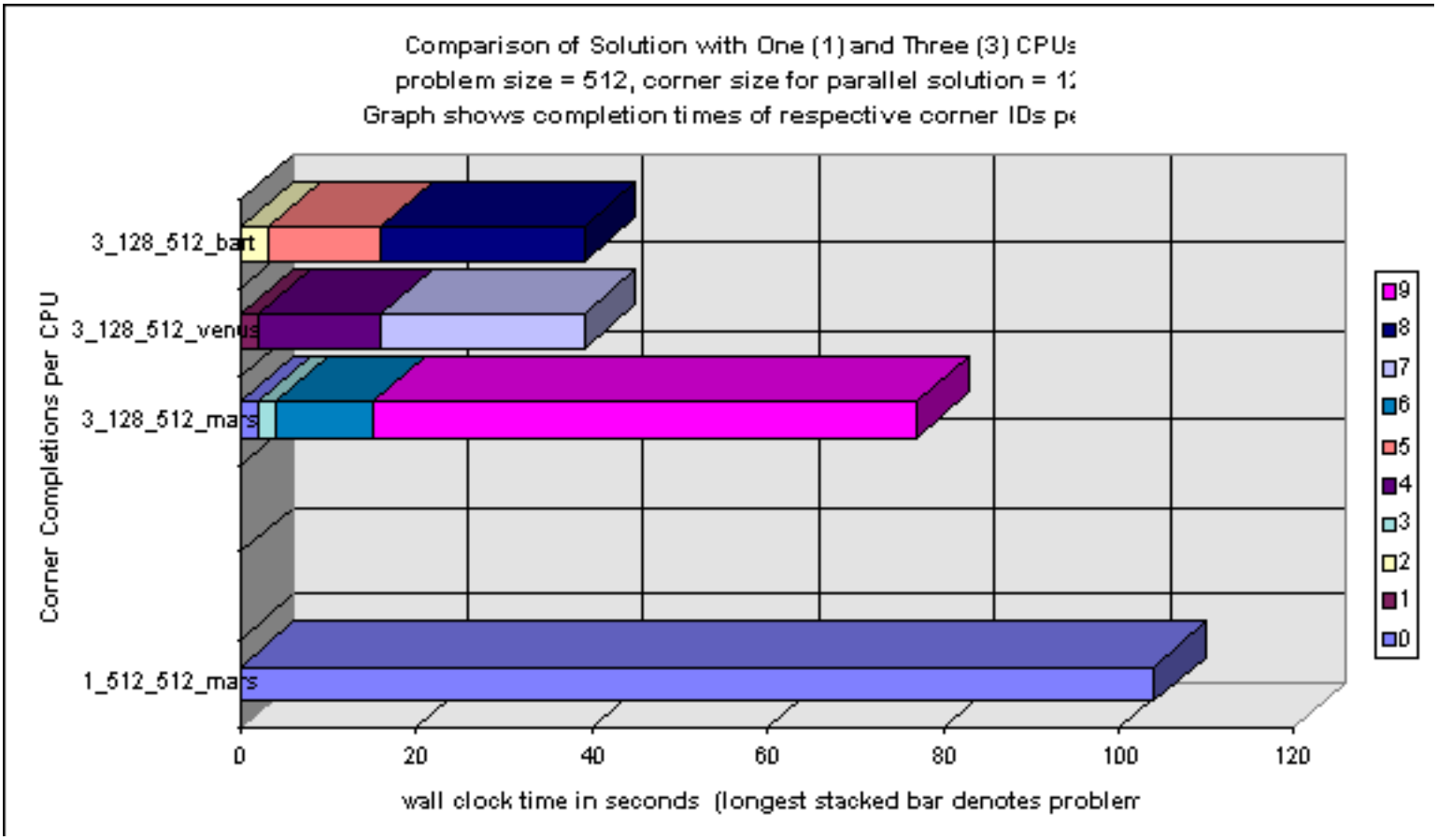


Figure 20 - Stack bar graph showing the wall clock performance of both the complete solution and individual corner objects broken down by CPU (n=512, cs=128,512)

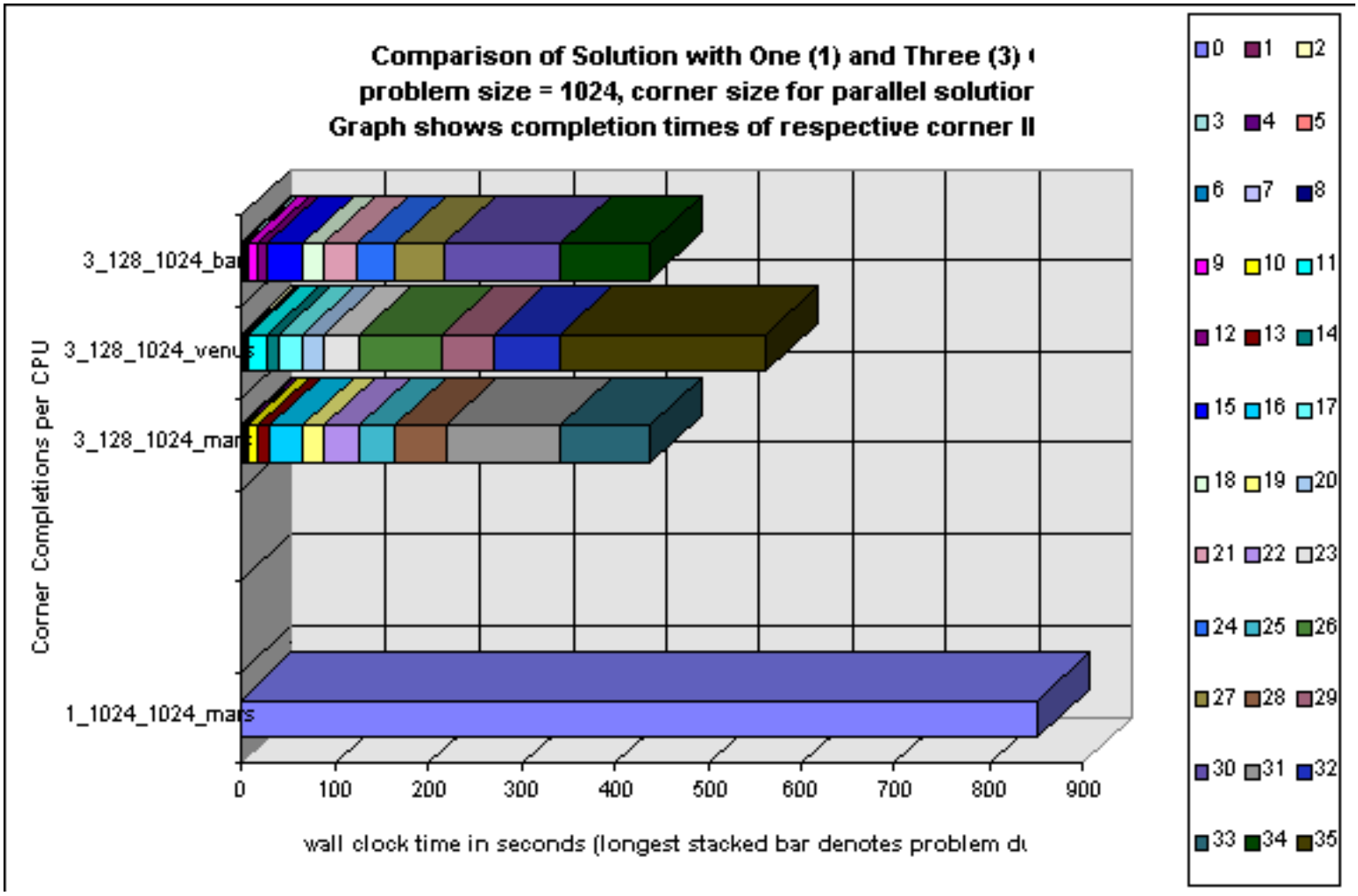


Figure 21 - Stack bar graph showing the wall clock performance of both the complete solution and individual corner objects broken down by CPU (n=1024, cs=128,1024)

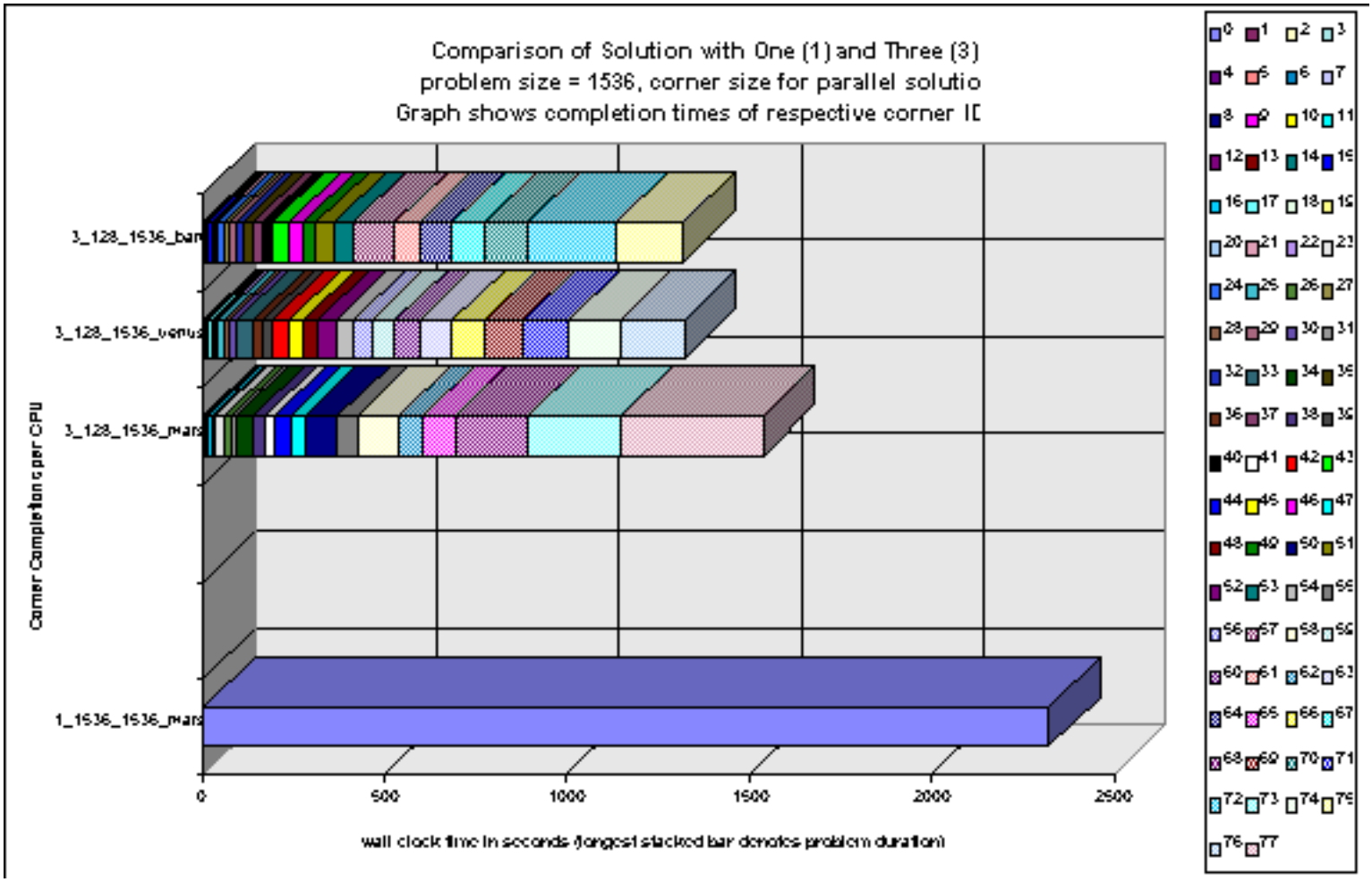


Figure 22 - Stack bar graph showing the wall clock performance of both the complete solution and individual corner objects broken down by CPU (n=1536, cs=128,1536)

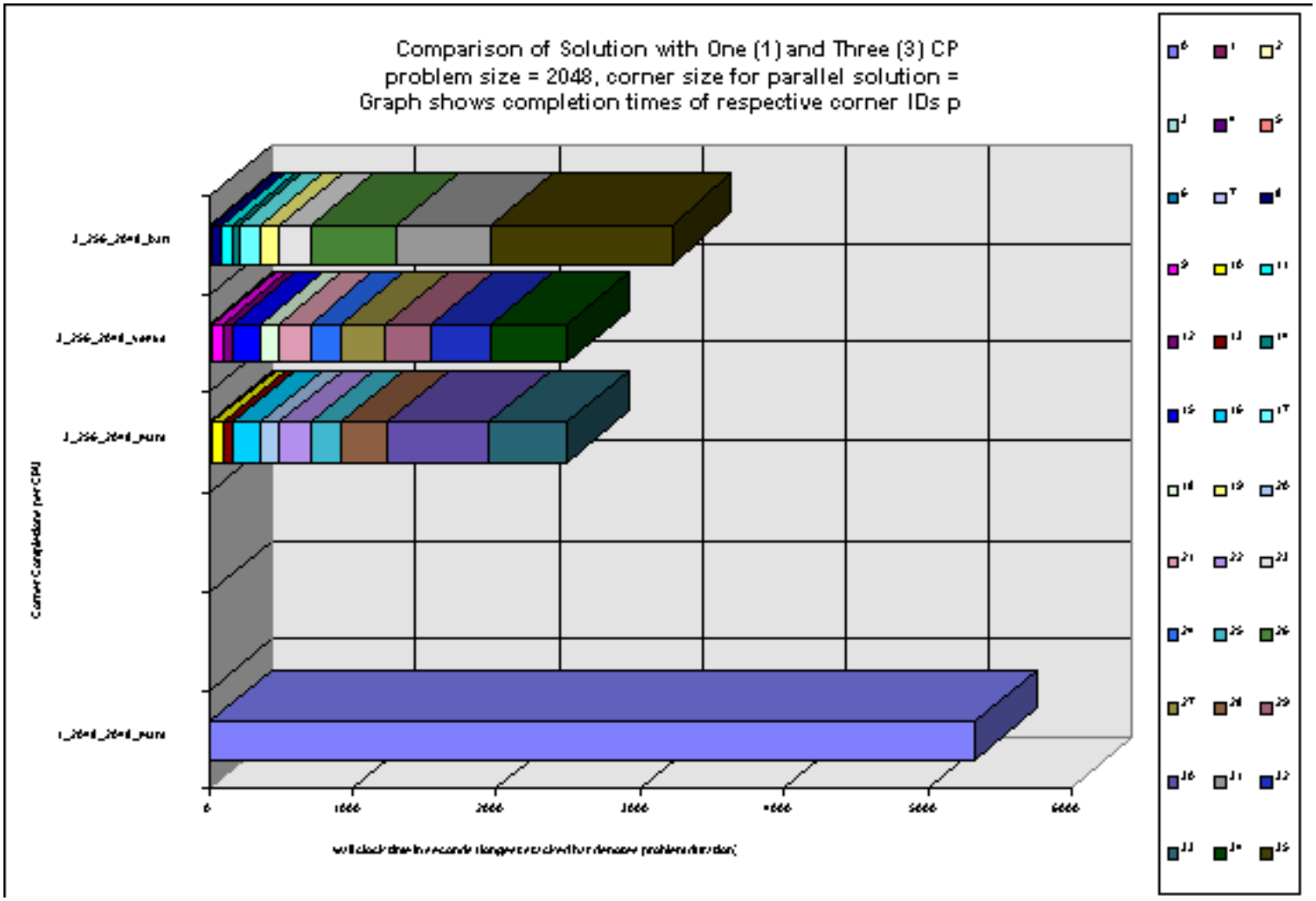


Figure 23 - Stack bar graph showing the wall clock performance of both the complete solution and individual corner objects broken down by CPU (n=2048, cs=256,2048)

Improvements

In addition to the improvements already presented above for improved data structures, better use of multithreaded environment, and more efficient data decomposition, the Manager/Worker scheduling aspects of this design need to be improved.

After all the effort made to make this prototype work on a heterogeneous multicomputer of various operating systems, processors speeds and interconnect speeds, heterogeneous operation yielded poor results because of poor scheduling algorithms. The system worked successfully over the heterogeneous multicomputer but the results were terrible. In most cases, the efficiency was not only poor but the wall clock time was worse than the single CPU case.

This result is not surprising because the scheduling logic (if you can call it logic) was too simple. As work became available and workers became ready, the calculations were assigned. The problem was that hard work was sometimes assigned to slow resources and faster resources were left idle. A simple remedy would be to rank workers according to some benchmark and evaluate calculation assignments

based on these benchmarks. A combination of this scheduling improvement, faster data structures and a higher parallelism during final stages through dynamic data decomposition should yield efficiencies closer the published values.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Conclusions

The success of the distributed object prototype using ORB technology shows that object oriented technology is the solution for distributed programming requirements. The thesis presented in this research, first stated in Section 1, that "Distributed object technology and concurrent/parallel processing are part of an abrupt evolution in information systems" and "Object technology will be the vehicle that allows information system providers to reap the benefits of distributed processing." has been supported in theory by this research and demonstrated in the prototype. This research makes no scientific estimation of the marketplace. However, in the months leading up to the draft of this paper, new technologies including the JAVA development environment have shown a skyrocketing in the market of distributed processing based on object frameworks and distributed objects.

The author would like to thank all family, friends, mentors, and supporters of this work over the last two years including Dr. Anthony Zygmont, Dr. Rick Perry, MRJ Inc., PostModern Computing (now Visigenic) Inc., and Infonautics Inc.

This page was last updated on February 18, 1997
Please send comments or questions to Matthew Stevens.

Footnotes

[1] H. Stephen Morse, "Practical Parallel Computing", Academic Press (AP) Professional, Massachusetts, 1994.

[2] Bruce P. Lester, "The Art of Parallel Programming", Prentice Hall, New Jersey, 1993.

[3] Andrew S. Tanenbaum, "Distributed Operating Systems", Prentice Hall, New Jersey, 1995.

[4] Eugene Hecht, "Optics", Addison Wesley, Massachusetts, 1987.

[5] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, "Computer Graphics - Principles and Practice", Addison-Wesley, Massachusetts, 1987.

[6] Doug C. Schmidt, "The ADAPTIVE Communication Environment - Object-Oriented Network Programming Components for Developing Distributed Applications", 12th Sun User Group conference, San Francisco, California, June 1993.

[7]ACE was designed to provide a framework for building very high speed distributed networking applications. Therefore, ACE concentrates on wrapping complicated UNIX System services in type-safe objects (IPC,Shared Memory, Dynamic-Linking, Threads, etc.). NIHCL was designed to simplify higher level application level programming.

[8] Keith E. Gorlen, Sanford M. Orlow, Perry S. Plexico, "Data Abstraction and Object-Oriented Programming in C++", John Wiley & Sons Ltd, England, 1990.

[9]A static class member is defined once for all instances of the class. It is not necessary to have an instance of the class to reference its static members.

[10]A const declaration specifies that the data can not be modified.

[11]All C++ classes have a primitive data member called "`this`" which is a pointer to itself.

[12]The copy constructor creates a new object from the same type argument object. Every C++ class has a copy constructor generated automatically. The standard copy constructor initializes every member variable of the new object with that of the source object.

[13]"Delphi, A System for Satellite Mission Planning and Scheduling, Preliminary Product Overview", Hughes Aircraft Company, Aurora, CO, 1993.

[14] Bruce Bohannan (Scientist/Engineer), "A Technical White Paper on the Hughes Inter-Process Communication Library (HIPC)", Hughes Aircraft Company, Aurora, CO, 1993, unpublished.

[15] Suresh Challa, "NetClasses - An Object-Oriented Communication Toolkit - Technology Overview", PostModern Computing (now Visigenic) Technologies Inc., Mountain View, CA, 1993.

[16]

```
/*************************************************************************/

/* ORBeline (c) is copyrighted by PostModern Computing (now Visigenic) Technologies, Inc */

/* Copyright 1993, 1994 by PostModern Computing (now Visigenic) Technologies, Inc. */
```

```
/* This code is the sole property of PostModern Computing (now Visigenic) Technologies, */
/* Inc. and is protected by copyright under the laws of the United */
/* States. This program is confidential, proprietary, and a trade */
/* secret, not to be disclosed without written authorization from */
/* PostModern Computing (now Visigenic) Technologies, Inc. Any use, duplication, or */
/* disclosure of this program by other than PostModern Computing */
/* Technologies, Inc. and its assigned licensees is strictly forbidden */
/* by law. */
/* All rights reserved. */
/*****************************************************************/
```

[17]scope delimiter operator

[18] The Common Object Request Broker: Architecture and Specification, Object Management Group TC Document Number 91.12.1, Revision 1.1, December 1991.

[19] Robert Orfali, Dan Harkey, Jeri Edwards, "The Essential Distributed Objects Survival Guide", John Wiley & Sons Inc., New York, 1996.

[20] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns - Elements of Reusable Object-Oriented Software", Addison-Wesley, Massachusetts, 1995.

[21] James O. Coplien, Douglas C. Schmidt, "notes

F="thesis_Object_I_O_.html#T1" TARGET="MAIN_FRAME">Object I/O</toc.gif 0100644 0000167 0000144 00000003514 06302364424 0013235 0 ustar 00mstevens perm 0000040 0000020 GIF87a# # ÷ c k k k k k s s! s)!s)!{1!{1){9){91{B1„B9„J9„RB„RBŒZBŒZJŒcJŒcRŒkR”kZ”sZ”{c”{cœ„cœ„kœŒkœŒsœ”s¥”{¥œ{¥¥„¥¥„--„--Œ-µŒ-µ”-½”µ½œµÆœµÎ¥½Î¥µÖ¥½Ö-½Þ-½Þµ½çµÆç½Æï½Æ÷ÆÆ÷ÆÎÿÎÎÕUUÿUU €UU€U∈Œ€UÕ€Uÿ€U ªUUªU€UªªUÕªUÿªU ÕUUÕU€ÕUªÕUÕÕUÿÕ