UNIVERSITY OF ALBERTA

An Appraisal of the Enterprise Model

BY

Ian Parsons ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Spring 1993

ISBN 0-315-82197-3

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Ian Parsons

TITLE OF THESIS: An Appraisal of the Enterprise Model

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1993

(Signed) [illegible]

Ian Parsons
6703-187 Street
Edmonton, Alberta,
Canada. T5T 2N1

Date: 10-25/93

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **An Appraisal of the Enterprise Model** submitted by Ian Parsons in partial fulfillment of the requirements for the degree of Masters of Science.

[illegible]

Dr. J. Schaeffer (Supervisor)

[illegible]

Dr. B. Cockburn (External)

[illegible]

Dr. D. Szafron (Examiner)

Dr. J. Hoover (Chair)

Date: Dec 16/92

Abstract

Enterprise is the latest offering in integrated programming environments for distributed parallel processing. Enterprise is intended for parallel programming on a network of workstations. The analogy of a business organization or enterprise is used to describe the various parallel constructs (line, department, division, individual, receptionist, representative, individual, and service) offered. Each *asset* or stand-alone module intended for one processor is created from the user's familiar sequential code and the graph that describes the parallelism. Enterprise is designed to use standard C code. The appropriate low-level communication and synchronization code is inserted by Enterprise; changes to either resources or the type of parallelism desired are easily accommodated and involve little user involvement.

This thesis presents four parallel algorithms that were developed using Enterprise: chaotic Gauss-Seidel, block matrix multiplication, transitive closure and alpha-beta search. The algorithms range from data-intensive (matrix multiplication) to computationally intensive (alpha-beta). Performance and ease of construction were two important metrics used in the evaluation. Performance was marred by the implementation of the communication server. Various experiments were done to isolate and identify the problem areas. Problems local to Enterprise are being addressed. The communication server problems were forwarded to ISIS (creators of the communication code) and may be addressed in the next release of ISIS. Several of the algorithms required user intervention to correct deficiencies in the current implementation of the Enterprise pre-compiler and run-time executive. Modifications to the Enterprise model and its implementation have resulted from this thesis. The implementation is now more robust and the model has been extended and simplified.

Acknowledgements

I would like to acknowledge the patience and forbearance showed by Steve Sutphen and Carol Smith who put up with my many questions about the workings of the ethernet, UNIX kernels and other arcane topics that crept into this evaluation. Also they were invaluable for suggesting further tests to isolate the quirks of the ISIS and NMP software packages.

I am grateful to authors of the executive, pre-compiler, and graphical interface, Pok Sze Wong, Enoch Chan, and Greg Lobe, who were always there when I was initially testing their products. Their quick and timely responses to either suggestions or questions about their work made my work proceed much faster than I anticipated.

I would like to thank the patience and continued good nature of Karim Makahani and Caroline Morris who maintain the student computer laboratories where I ran most of my tests. Unfreezing machines that I managed to corrupt, coordinating use where it wouldn't interfere with course work, and providing scarce storage space are above and beyond what their duties are and I am grateful for their assistance.

My thanks to Anthony Jones for suggesting the two slow converging problems for the Gauss-Seidel algorithm.

Finally, to my wife Edith and my two children Matthew and Ellen: I thank you for allowing me the freedom and opportunity to go back to university.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Being the first user of a brand new product is always an experience. It is similar to getting behind the wheel of a new car and taking the car out onto the highway. The driver has preconceived ideas about what the car can and should do. The car designers also have preconceived ideas about the typical driver. When these two groups of ideas collide there is a period of learning and growth before everyone adapts to the reality of the product.

Enterprise is a new collection of software tools intended to manage the low-level implementation details of coarse-grained parallel programming on a network of workstations. These software tools are presented to the user by way of a graphical user interface. The user, freed from the tedium and complexity of system dependent low-level details, can concentrate on developing the parallel aspects of the algorithm.

The complexity associated with developing parallel distributed programs is divided into two areas: the network and the program. The network complexity includes the heterogeneity of hardware and software. The program complexity involves deadlock, race conditions, message-passing, and non-determinism issues. The first two issues are possible in sequential multiuser programming (for example, database programming) but all of these issues frequently appear in parallel programming. This complexity affects the implementation of the Enterprise project in its performance and, more importantly, its usefulness.

The Enterprise project is the blending of three ideas or theoretical models of programming: the Enterprise model, the communication model, and the user's parallel processing model. This is not a smooth blending of ideas, although each model is effectively insulated from outside interference (in theory). With the implementation of the overall system, each model must adapt to the others - compromising principles. The success of all the models is measured by how little the implementation affects the final product in both efficiency and usefulness.

If the implementation of Enterprise requires constant user intervention, its usefulness will appeal to a much smaller group of experienced implementors. However, if the user is completely shielded from the complexity, the resulting distributed program will not always be efficient. Balancing these two equally valid states is neither an easy nor a clearly identifiable task. Permitting various shades of shielding allows both inexperienced and experienced parallel programmers to have exactly the type of environment necessary to quickly and easily produce distributed parallel programs.

This thesis is an evaluation of the state of the Enterprise project as of the spring of 1992.

1.1 Motivation

With the continual increase in the availability of low-cost and high performance workstations, there is a steady increase in the amount of unused computing cycles. Fully utilizing these spare cycles on a network of workstations in the form of a distributed program results in a low-cost supercomputer. There is a strong interest in harnessing these idle cycles, but there is a high cost associated with developing and maintaining such applications. Any tool that reduces these costs is desirable.

Compilers are essential tools designed to convert the user's high-level specification language into machine code intended for specific architectures (conventional or parallel). Similar tools to shield the general user from the complications of distributed parallel processing are necessary. Distributed parallel processing involves running processes on separate machines with communication between these different processes. Until software tools similar to compilers are developed for creating distributed parallel applications, the typical user is stranded, figuratively speaking, back in the dark ages of computing. The distributed code is hand-crafted to a specific instance of machines and networks. As such, it is difficult to maintain, modify, or adapt the developed code to other instances.

The type of parallelism suitable for a distributed network of workstations is coarse-grained. This means that the computational requirements should be large in comparision to the cost of exchanging information. This large granularity implies functional rather than instructional parallelism. That is, each processor must perform a complex series of instructions in parallel with other processors to offset the cost of communication.

In practice, it turns out that for most coarse-grained parallel applications, only a small set of constructs is needed to define the requirements for parallel coding. This set consists of replication of identical processes, restricted cases of recursion (an attempt to avoid deadlock or race conditions), a pipeline (each process does part of the work before passing it on to the next processor), or dividing the work into heterogeneous parts that are capable of being processed in parallel. These constructs are regular and well defined - ideal for automated code insertion by a *pre-compiler.*

The term pre-compiler is chosen because of the temporal ordering of its use. While both the conventional compiler and the pre-compiler tools use a grammar and symbol table to translate abstract templates to a specific reality, the distributed reality must be dealt with prior to dealing with the physical reality. The pre-compiler modifies the user's code to represent the distributed code requirements in a high-level formal programming language. Distributed processing is abstracted from the physical reality since properly developed communication software is not tied necessarily to a particular machine type or network type. Only after the pre-compiler is finished does a traditional compiler create executable code intended for a particular machine architecture.

It is difficult for the general user to write good parallel code even with these simple constructs to use as templates for code insertion or modification. Two reasons for this are inattention to detail and the human tendency to take shortcuts. Back in the dark ages, programmers hand-crafted machine level code. This is difficult and error-prone even for an expert. To create a wider community of programmers, the expertise of the machine-code implementors was abstracted into the formal high-level programming languages of today and the software tool, the *compiler*. Compilers are tools that provide the user with the experience of an expert code implementer. Compilers can provide tracing and debugging facilities with little effort on the part of the user. Similarly, by taking the expertise of a good distributed programmer and the attention to detail offered by automatic code insertion, a pre-compiler tool is created for parallel processing.

The pre-compiler does two things. First, the user is freed from the tedium of network communication details. Details like connections between processes, correct message formats, and error recovery are easily automated. Portability becomes much simpler since the pre-compiler takes advantage of the independence of the communication subsystem in both machine and network architectures. The traditional compiler creates the machine code for each required machine type. Second, the time spent debugging the communication code is eliminated. It is assumed, of course, that the pre-compiler does its work correctly. The tracing and debugging capabilities inserted by the pre-compiler are used to profile problem areas in the user's implementation or logic flaws within the algorithm itself. This makes the production of working programs both faster and easier for the user.

The advent of conventional compilers constrained the types of code that could be developed. The complexity of adding distributed processing requires limiting the types of automated code development. The truly dedicated and experienced programmer can craft the optimal distributed program but the cost is out of proportion to most needs. Portability also decreases quickly.

The advantage of the pre-compiler is that more people can easily become accomplished distributed parallel programmers. The dread of learning all the low-level communication protocols and the associated debugging skills are large impediments to the growth of this area of computing science. The real challenge is found in the creation of distributed algorithms but the current impediment of implementation discourages the trip from theory to practice.

Another low-level preoccupation with this type of parallel computation is the availability of machines. When should a process be allowed on a machine? Generally, the user does not own all the machines and access times are limited.

Conversely, when should a process retire from a machine? Not only may the access time expire, but the workload imposed by the parallel process may interfere with the machine owner's use. If a faster or better suited machine becomes available, how does the user migrate the parallel process to this new machine? All of these easily automated tasks can be handled by a run-time *executive* process that is reusable for many different parallel programs.

The concern with replication details is another easily automated low-level task. This task is responsible for balancing distribution of work requests to different instances of the same process. This could include adding or deleting new processors as they are needed or become available, forwarding work requests, and saving requests until processors become available. This general purpose task is best characterized as a *manager*.

The Enterprise project is a collection of software tools intended to aid the general user in bridging the complex implementation gap between theory and practice in coarse-grained distributed processing on a network of workstations. Enterprise provides the pre-compiler, runtime executive, and managers wrapped up in a graphical user interface. The user, supplying the sequential code, can test different parallelizing techniques quickly without having to deal with the tedious and error-prone details of interprocess communication.

1.2 Enterprise Overview

The Enterprise project is intended to take standard sequential C code and run it in parallel over a network of workstations [8]. The analogy that drives the program development is that of a business organization.

The different parallel constructs are couched in business terms. These constructs are called *assets* since they are of value to the user. An asset is a template of some specific parallelism required by the user. An example is the term *line*. A line is similar to an assembly line in that one asset does some work and then passes the partially completed work on to the next asset in the line. The pre-compiler takes the assets and the user code and produces the processes and communication structure that reflect this parallelism.

The assets can be *coerced* into other asset types depending on the user preferences. The coercion changes the code inserted by the pre-compiler only – the user supplied code is unchanged. This object-oriented design of Enterprise gives all the assets a simple external view while the assets themselves could be composed of complex parallel assets. This abstraction permits a clean implementation of the overall parallelization of the user's task.

The main user environment is the graphical user interface (GUI). Here the user draws the graph defining the parallelism, writes the sequential code for each node, initiates the pre-compilation and compilation of the nodes, and launches the Enterprise program. All of these *software tools* are initiated by the GUI. The user is shielded from the implementation details of all of these tools.

The tool, **compile**, builds each of the nodes into processes suitable for the different machine types. It accomplishes this by using the user-generated graph to direct the pre-compiler to insert the necessary communication code. The modified code is then conventionally compiled to the various target machines.

To launch the Enterprise program there is an Enterprise task, **run**. This task provides the interface between the user and the run-time executive. It provides a single entry-point for user interaction.

The run-time *executive* is a group of three tools hidden from the user. The first tool, **machine_mgr**, launches the user processes and manages machine access and availability. The second tool, **asset_mgr**, manages the replicated assets. A separate asset manager is launched for each group of replicated assets. The third tool, **monitor**, is a single process used to log messages sent between assets. This is useful when debugging or profiling run-time behaviour of the Enterprise program. These tasks are launched with every Enterprise program.

The GUI provides the user with a well-defined environment which is independent of any machine and network differences. The user need not know the details of the machine, network, or communication components to successfully create a parallel program. The more experienced programmer can remove some of the shielding to obtain a more efficient product. The GUI, launching of programs, run-time executive, and pre-compiler are part of Enterprise and are not directly modifiable by the user.

With Enterprise there are no new key words or routines to use when programming. The current programming language is C. The only restriction is that any pointers to memory locations cannot be passed in a message from one machine to another. This makes sense since, what would machine A do with a memory pointer from machine B?

The creation of the graph, compilation, and run-time steps do not need an integrated interface to run but this is not the purpose of Enterprise. Enterprise assists in quick prototyping of distributed programs. By providing a GUI, the user works from one place and, unbeknownst to him or her, the details of the pre-compiler and the launching of tasks are hidden from view. Creation of the graph file is as simple as selecting a menu item. Usually, the details of such topics are not relevant to the development of the distributed algorithm.

By using Enterprise, the user now becomes the chairperson or chief executive officer of an enterprise and outlines the problem while the well-trained management team delivers the final product. This is a vast improvement in both productivity and portability from the previous one-person show.

1.3 Scope of This Thesis

This thesis is an appraisal of the implementation of Enterprise as of the spring of 1992. Specifically, the implementation and execution of four parallel algorithms are examined with respect to the implementation considerations (ease of use, thoroughness of the pre-compiler) and the performance of the implemented program (speedup and robustness of the code).

The graphical user interface [19], the executive tasks [28], and the pre-compiler [7] have been implemented but not tested by a general user. The evaluation done as part of this thesis has and will cause changes in all components. Additionally, this work has uncovered flaws in the communication subsystem.

Four algorithms were implemented using Enterprise. They are intended to evaluate different aspects of Enterprise. The first algorithm is the parallel version of iterative Gauss-Seidel. This algorithm solves a family of linear equations. The parallel version of this code is non-deterministic in arriving at an answer. It is dependent on having all components running concurrently. This algorithm cannot be tested sequentially and reliability is a concern.

The second algorithm is parallel matrix multiplication. This is intended to test both the manager task when more requests arrive than there are processors available and how the efficiency of the communication subsystem in transmitting large messages. The modifications the user is required to make when changing between two different parallel implementations of the same algorithm are evaluated.

The third algorithm finds all possible connections between different nodes in a graph – the transitive closure. This algorithm is not easily amenable to the Enterprise model since the work, which must be performed in order, is generated by peer processes for other peers. The Enterprise model expects data to flow down the process graph.

The fourth algorithm is the alpha-beta game tree search algorithm. This recursive algorithm is intended to test and contrast the two assets representing replication and recursion. The user modifies the sequential code necessary for each implementation. The ease of making the modifications and the efficiency of the two parallel algorithms was to be compared. Only the version with parallel replication is presented due to the delay in implementing parallel recursion in Enterprise.

None of the above algorithms achieves outstanding performance; however, the actual implementation was easy and painless. An investigation as to why the performance was suboptimal centered on the message passing performance of the communication subsystem to identify any weak points in either Enterprise or the communication subsystem implementations.

The message passing performance of the communication system is critical to any distributed processing. Indeed, it is a major factor for the overall success of Enterprise. Enterprise shields the user from actual implementation of the protocol. If the performance is sufficiently poor, Enterprise will not be an efficient tool for producing quality distributed processing code.

Enterprise employs a manager to coordinate the activity of a replicated asset with the rest of the Enterprise program. What is the cost of the manager in terms of performance, machine resources, and efficiency?

1.4 Outline

The thesis is divided into six chapters. The second chapter details the current Enterprise model. The third chapter documents the implementation details as of the spring of 1992. The fourth chapter summarizes the theory, implementation, and some of the experimental results of four distributed algorithms. The fifth chapter examines the message passing performance of the communication system and the manager task of Enterprise. The sixth chapter is a evaluation of the implementation of Enterprise as of the spring of 1992 and potential research directions.

1.5 Summary

Enterprise is an example of a collection of software tools designed to quickly and easily prototype parallel algorithms for distributed processing. The time spent debugging the communication code is more usefully spent developing the distributed algorithm. The current implementation is close to working as designed but deficiencies which preclude the general release of Enterprise to the public still remain.

Some of the flaws in the current implementation are attributed to the fault that the model was not complete before the implementation was started. However, the model was not found to be incomplete until after the implementation. A continual source of minor and major flaws is the communication subsystem. The communication subsystem, ISIS, while a commercial product, is neither as robust nor fully tested as it should have been. Possibly, Enterprise is using it in a way that ISIS was not intended to be used, but some of the flaws uncovered are basic to the operation of ISIS.

Both the executive and pre-compiler were tested before they were finished and completely debugged. Many of the problems will have disappeared with the next version of these software tools. The Enterprise pre-compiler still has not approached the model's requirements. This is not the fault of the implementor since the complexity is not trivial. It does mean that the communication code that is inserted requires further modification to work correctly. This extra editing is not for the casual user of Enterprise but rather for someone who has used both Enterprise and the communication subsystem before and knows the algorithm and its implementation thoroughly.

The need for something like Enterprise to automatically insert appropriate code for the communication between processes is essential to move distributed processing from the perception of a black art to a logical and controllable science. The complexity still exists for the experienced programmer to extract the last dregs of performance. However, the hurdle for most users to reap the benefits of distributed processing is significantly lowered by using Enterprise to develop and run distributed code. Enabling more users to produce distributed code opens this black art to the light of knowledgeable, logical, and rational critical review. A drive in a car to someplace new is a broading experience; having a new car makes the trip enjoyable.

Chapter 2

The Enterprise Model

The Enterprise programming model is the latest in a series of programming models that have appeared in the past few years. Their purpose is to convert existing sequential programs to parallel programs or to develop parallel programs directly. The common goal is to aid the user in transforming an abstract theoretical model into a concrete application of workable code [7]. The low-level details, daunting to the casual user, are masked by abstracting the physical layer to a *virtual machine*. The low-level process communication and synchronization details are now hidden by a group of software tools.

These programming tools provide a mechanism for both process and message identification to aid in monitoring and debugging. This is important in producing error-free parallel programs. In short, these low-level tools aid the programmer to create and identify messages sent from one processor to another processor in some temporal order.

By concentrating at a higher level, the user has a more stable programming environment since the tools used to manage the low-level operations are able to run on multiple platforms. Pratt [24] points out that programmers often prefer a *familiar and convenient* programming environment with reasonable performance over an environment which gives maximum performance but requires a significant amount of additional programming effort.

Tools designed for parallel programming abstract the view of the underlying physical machine for the programmer. This permits programmers to concentrate on the parallel structure of their programs rather than dealing with hardware details. Such ease of programming does not come without a cost. Jones and Schwartz [13] point out that the masking of the hardware architecture in a distributed system comes with a performance penalty. The best possible performance in distributed parallel computing environments requires that the programmer have some control over the mapping between processes and processors. Ideally, a parallel programming tool should provide both options: automatic processor allocation for naive users, and full processor allocation control for advanced users.

An integrated environment for parallel programming provides the programmer with a set of tools to create, debug, and visualize a parallel program. These tools share a consistent user interface displayed in a uniform graphical display. A structural graph, in the form of a control flow graph or a data dependency graph, defines the parallel structure of a program. The run-time behaviour of the program is visualized by animating the defined structural graph. Visualization of program execution is particularly important to parallel programmers because it provides an easy way to identify performance bottlenecks of parallel programs.

In addition to the advantage of visualizing the execution of a parallel program, most of these environments, such as HeNCE [5], Paralex [3], FrameWorks [26], and PIE [17] separate the definition of functional modules and the definition of the parallel structure of a program. This option adds the flexibility to restructure a finished parallel program and to experiment with different forms of parallelism in order to achieve better performance since these are conceptual models rather than physical models, portability is improved.

One such conceptual model is object-based parallel programming. The idea behind the object-based model is similar to that of object-oriented programming. An object-based program is composed of modularly decomposed units. These units are independent, self-contained entities which may contain data, operations, or both [23]. The basic function of such an entity is to interact with other entities by sending and receiving messages. This model is ideal for a distributed network of workstations, the targeted architecture for Enterprise.

2.1 Enterprise Model

The Enterprise model has gone through several changes. This section briefly describes the state of the model as of the spring of 1992.

The Enterprise model, described in detail elsewhere [8, 7, 28], is built upon work developed for Frameworks [26]. The anthropomorphic principle behind Enterprise is a business. That is, the elements of the model are named after business concepts that describe the intended parallelism. Business enterprises are naturally parallel (the successful ones anyway). Various constructs from this business model are used to describe the intended parallelism. While this is a simple yet powerful model, there are limitations to the types of parallelism possible.

One such constraint is that there is a hierarchal ordering to the parallelism: there exists a process thread that proceeds from the beginning of the program until the end. This is consistent, in part, with the business analogy since arbitrary connections are not supported either – control and data flow through the established chain of command. Arbitrary connectivity or new paths are not possible on a dynamic basis. The chain of command is comprised of individuals with different duties; each individual exchanges messages either with their immediate supervisor or with individuals they immediately supervise. Control of the data is passed down the organization until an individual is reached who actually does the work. Coming back to the programming aspect, the entire program now either rewinds to the beginning or stops execution.

An Enterprise program is described by a directed acyclic graph. Each node represents an *asset*, the template for the type of parallelism selected. An arc represents the communication between processes. The user creates the sequential code for each individual asset. An important point is that no asset can call itself, except under special and controlled circumstances. This is intended to reduce or eliminate the possibility of deadlock or race conditions. Nodes selected for replication represent multiple instances of the same asset.

Sequential programs are hierarchal in nature. Using Enterprise to express the parallelism that is intended from within this sequential ordering is often straightforward.

The method for expressing the intended parallelism is graphical, either via a textual format or by drawing a structural graph on the screen. The implementation coding details are the task of the pre-compiler. The user is responsible for drawing the graph with the nodes and the connections between nodes.

Replication, expressed as a tuple of minimum and maximum values, is handled in two different ways by Enterprise. If the two values are the same, Enterprise will allocate all the requested number of processes at the program startup time. However, if the two values are not the same, Enterprise will allocate the minimum number of processes at startup time and dynamically recruit up to the maximum value from the remaining free processor pool as work becomes available at run-time. If a process is finished and there is no work waiting for it, Enterprise dismisses the idle process, freeing the processor for other work. At no time will the number of allocated processors fall below the specified minimum. If there are no processors available or the maximum replication is reached, the requested work must wait for an idle processor.

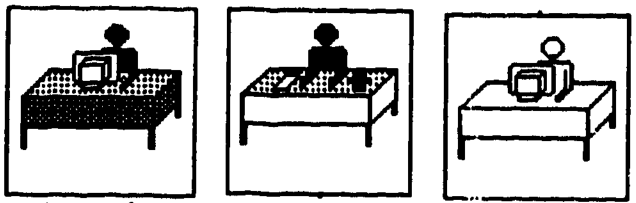
If a particular asset is already replicated, changing the number of members does not require a recompilation of the asset. The number of replications is a run-time concern. The amount of replication can be changed any time by the user prior to actually running the program.

Any asset identified as an individual (the simplest type of asset) can be *coerced* to other, more complex types. The user must supply the sequential code for each asset. If the user wants to change the parallelism of the program then the graph is redrawn. The user's code will likely require modification to reflect the change in parallelism. The changes involved, if the user designs the implementation correctly, are either consolidation or separation of asset source files. Each source file represents a portion of the functionality of the parallel algorithm. Usually, changing the parallelism of the algorithm involves only consolidating or splitting of the functionality of portions of the algorithm.

No arbitrary connections of nodes are allowed other than to simple nodes which are called *services*. A service is available to *all* processes from one logical site only. The graph representation of these services is a node that is separate from the main graph. In this way the user is aware of the peculiarities of services and the graph does not get cluttered with the connections of all nodes to the service. However, if the user wants separate, concurrent Enterprise processes (multiple businesses), then interaction is available via well-defined interfaces (eventual design for services).

The following sections describe the different assets and their potential uses. Figure 2.1 shows a schematic of five simple parallel constructs possible using Enterprise.

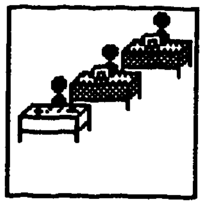
2.1.1 Individual



An *individual* is the base asset of Enterprise. The individual is responsible for executing some sequential code segment. A basic individual can be replicated or coerced into other assets. An individual is represented as a single node in Figure 2.1a. An Enterprise program starts as an individual. All leaf nodes in the Enterprise graph are considered individuals.

There are two special types of individuals. A *receptionist* is the entry point for a composite asset such as a line, department, or division. They cannot be replicated or coerced into any other asset. Representatives are the leaf nodes for a division. Representatives can be replicated or coerced only to a division. Reasons for these restrictions will become apparent shortly. Regardless of the type of individual, an asset classed as an individual has user code associated with it.

2.1.2 Line



The *line* asset is taken from the idea of an assembly line. There are n heterogeneous assets in a line composed of one receptionist and $n - 1$ assets (Figure 2.1b). Each asset calls the next asset in the line and replies (if necessary) to the previous asset. Each asset is assigned a particular piece of work and passes its completed work to the next asset in the line. A line has n different computations being done simultaneously. However, the speedup observed is bounded by the slowest asset. This asset will have messages waiting to be processed by previous assets and starving following assets. By replicating or coercing these slow assets, bottlenecks in computation are reduced or eliminated (Figure 2.1c).

The first asset in a line is the receptionist which shares the name with the line. It is the only asset that is externally visible. That is, an external asset may only call the receptionist of a line.

A line can be replicated as a single entity. Its members can be coerced into other lines, departments, and divisions as well as being replicated. The receptionist cannot be coerced since it defines the external view of the line.

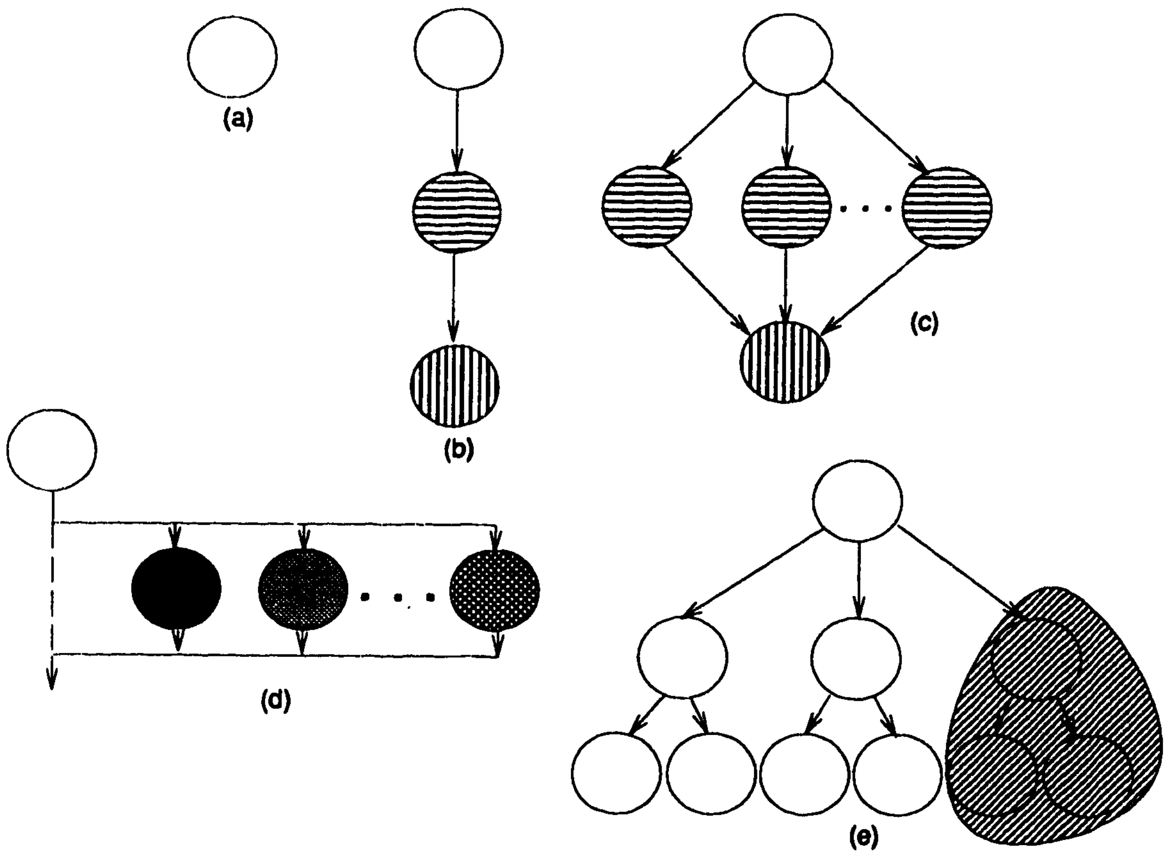
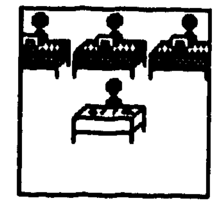


Figure 2.1: Basic parallel constructs of Enterprise.

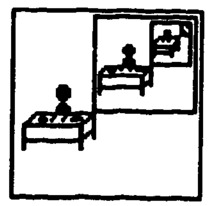
2.1.3 Department



A *department* has two distinct parts (Figure 2.1d). The first is a receptionist who is responsible for all requests directed to the department. The second part is the heterogeneous group of assets (individuals, lines, departments, divisions) that will process the requests of the receptionist in parallel. The receptionist processes the external asset's request and distributes the relevant work to its managed assets. The current implementation requires these managed assets to reply back to the receptionist who then calls or replies to the next asset in the graph. This restriction is being removed. While it does increase the complexity of the communication requirements somewhat, it does not compromise the basic tree structure.

A department can be replicated as a single entity. Its members can be coerced into other departments, lines, and divisions as well as being replicated. The receptionist cannot be coerced since it provides the external view to the department.

2.1.4 Division



The *division* asset is the only asset which can call itself (Figure 2.1e). This asset is intended for divide and conquer procedures with recursion carefully controlled. The user writes the serial code with a recursive part and a stopping criteria for the recursion.

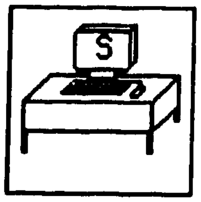
A division is considered by the user to be a block of homogeneous assets, all containing the same recursive sequential code. But within Enterprise, the internal and leaf nodes have different functionality representing the type of recursion. The internal nodes generating and receiving messages between the other nodes in the division represent the parallel recursion. The internal nodes perform only parallel recursion. The leaves of the recursion tree are called *representatives*. They receive messages from internal nodes and perform sequential recursion as specified in the user's code.

All these nodes are associated with distinct communication groups for the correct control and flow of data. The shaded part of Figure 2.1e shows the identification of a communication group. This permits the recursion to mimic the effects of the sequential recursion with the different levels clearly identifiable.

The user specifies the width and depth of the division - the limits of the parallel recursion possible. The width of each level in the recursion tree is controlled by the replication factor (permitting arbitrary fanout). The depth of the parallel recursion tree is the number of divisions created plus the representative asset.

Each level of depth of the parallel recursion is specified by another icon with the same name. Having the same name for the different asset icons reinforces to the user that these assets are identical, even though Enterprise assigns different functionality to the various nodes. In Figure 2.1e, the depth of this tree is three while the width is three at the first level and two at the second level. The width can vary at each level depending on the user's application. The replication factor specified is not the total replication for the level but rather the number of assets that communicate with one asset specified in the previous level. That is, the number of replicated assets at one level is the product of the replication factor at the current level and the replication factor of all the previous levels. Care must be taken that the number of processes does not exceed the number of processors available. The number of processes required in the example case is ten plus the managers for each processor group. The number of managers is unknown at this time since the division implementation is not complete.

2.1.5 Service



The *service* asset is similar to describing a business asset that does not consume or produce work for the business. An example is a blackboard or a clock on the wall. All processes may call any service to request something (a global variable, work-to-do item) or deposit something (modified global variable, work-to-do list). Having a service does enable some arbitrary connectivity. A service could be another Enterprise program but is currently implemented only as an individual process.

A service usually simulates shared memory in the distributed memory environment. The high cost of such simulation must be weighed against the need for such simulation.

2.2 Programming Model

The atomic element of programming with Enterprise is the user-supplied function or subroutine. The function's formal parameters define the composition of the message sent to it by other assets. The internal structure of the user's function reflects the parallelism selected. For example, in line parallelism the asset expects work to arrive via the formal parameter list; some processing of the work is performed by the asset; and modified work leaves by way of a function call to the next asset in the line.

Enterprise requires the user to supply the code segments necessary to express the parallel algorithm. The user-drawn graph designates to the pre-compiler and run-time executive both the parallelism of the algorithm and the connections required between the various code segments.

Enterprise supplies all the communication code necessary to implement the user's parallel algorithm. The run-time executive will launch and stop all the processes necessary to execute the algorithm.

2.2.1 Writing an Enterprise Program

The first thing the user must do to create any parallel program is to think about the following questions. How is the parallelism expressed in the algorithm – a pipeline, different tasks executing simultaneously, or divide-and-conquer? What are the different modules needed to express the algorithm? Under Enterprise, the user calls a function to express parallelism.

Next, the user needs to determine if the modules require shared memory (global variables). Can the algorithm be modified (if at all) to exist in a distributed memory environment? Acessing shared memory in a distributed memory environment is expensive. To simulate this requires either extra parameters being passed (if the globals are static over the entire run), or a *service* which will give the caller the current state of the global variables.

Once these design issues have been addressed, the implementation can begin. Enterprise implementation is based on an extension of the remote procedure call (RPC). A standard RPC has a process A issuing a request for work to process B. Process A then blocks until a reply (if any) is received from process B [27].

Hidden from the user, Enterprise transforms the user function call into an outgoing message and an incoming message (if necessary). The outgoing message is the list of parameters the user would use calling the function, while the incoming message is equivalent to the value returned by the function. The sequential form of the code would block until the function return.

Enterprise relaxes the necessity for blocking by queuing messages at the recipient and acknowledging only the receipt of the message. The process generating a work request which is not expecting a reply, blocks long enough to send the message; work then proceeds asynchronously in both processes. The work request that expects a reply operates with ***lazy synchronization.*** First, the process blocks until the receiver acknowledges the receipt of the message. Second, the calling process is allowed to continue until it accesses a memory location where the expected reply is to be placed. The process then blocks until the reply is received. The parallelism is found during the time spent before blocking.

However, messages are dealt with in a temporal order. A received work request must be completed prior to the receiver processing the next request. The receiver can act as a sender to subordinate processes with any number of requests being sent but all the expected subordinate replies must be received prior to replying to the original request.

The following simple example (Figure 2.2) demonstrates this. The value of x is packaged into a message and is sent to f while g continues execution until a is accessed. If f has responded with an updated value for a then g does not block but continues execution. However, if f has not responded, g must block until the reply is received from f.

The user writes the needed functions and support libraries. The algorithm, if possible, should be debugged sequentially to eliminate as many programming errors as possible. This also gives a hint as to the performance of the algorithm when there are limited machines available. Given a working or at least debugged group of functions, the user is ready to use Enterprise.

In Enterprise, the user draws the graph expressing the parallelism of the algorithm.

```
g()
{
    a = f(x);
       :
    b = a;
}
```

Figure 2.2: An example parallel program using lazy synchronization.

Each asset node is associated with a user-supplied source file. Based on the graph, Enterprise will insert the appropriate communication code and compile each asset into a stand-alone process for each of the desired target machines. When this compilation is done, the user is ready to run the Enterprise program.

2.2.2 Running a Parallel Program

The user's C code has been translated into a sequence of messages. The calling semantics of each function are translated into messages that are delivered to the process representing the sequential function. Any return variables or parameters passed by reference to this function are similarly packaged into a reply to the originator's message. Any error conditions that result in a function aborting are propagated to all the processes composing the Enterprise program resulting in a clean shutdown of the entire program (similar to what would occur in a sequential program).

To run the successfully compiled program the user optionally supplies a list of machine names suitable for execution of the program. The user selects **run** from the command menu and the Enterprise executive launches all the appropriate tasks on the available machines. When the Enterprise program finishes, the executive shuts down all of the processes.

2.3 Related Tools or Approaches

There are other products available which have similar functionality to Enterprise. Four of these are summarized and compared to Enterprise. They are HeNCE, Paralex, FrameWorks, and PIE.

2.3.1 HeNCE

Heterogeneous Network of Parallel Machines, HeNCE, [5] is a software package that supports the creation, compilation, execution, debugging, and analysis of parallel programs. It is the closest to Enterprise in functionality.

An acyclic graph, specified by the user, determines the parallelism of the program. Each node of the graph represents a subroutine or function of the program while an arc represents the data dependencies. HeNCE inserts the necessary communication code and runs the procedures on the machines in the network. HeNCE supports dynamically spawned subgraphs, pipelining, loops, fan, and conditional constructs. HeNCE supports the FORTRAN and C languages.

The parallel constructs defined in Enterprise, such as a *department* or *line*, can be constructed in HeNCE. Recursive parallelism, i.e. *divisions*, are not readily constructed unless the recursion is unrolled. HeNCE programs appear to be at a lower level in scope than Enterprise because the user has to specify a script to execute the various subroutines or subgraphs. The parameter passing of HeNCE is specified separately from the function parameter list. This permits several nodes to supply different parts of the formal parameters to a single node. Enterprise takes the parameters as specified in the function call and transports them to the remote process. Enterprise does not support multiple nodes supplying subsets of the parameters to a single node. Each Enterprise message supplies the complete parameter list as specified in the function definition.

HeNCE has a more advanced graphical interface than Enterprise. The animated debugging and display are quite sophisticated. A drawback of HeNCE is that the **makefile** (a file containing the instructions for compiling the modules) requires separate libraries or preprocessor directives that must be modified by the user. Enterprise allows the user to fill in a form and creates the correct **makefile** without further intervention.

In summary, while HeNCE is more advanced visually, the user must work harder to express the parallelism. This results in error-prone scripts and poor reusability when experimenting with different parallelism. The Enterprise model requires the user to modify the graph and to change the user-supplied source code when changing the parallelism.

2.3.2 Paralex

Paralex [3] is similar to HeNCE in that the user specifies a dependency graph and each node in the graph is a function. Multiple nodes may provide input to a single node as in HeNCE. The differences are that the output is limited to a single value (simple or complex). The user must code *filter* nodes that partition the function output for future nodes in the graph. This limits the portability and modification of the desired parallelism.

Paralex does not support recursive parallelism. Its fault-tolerance capabilities preclude access to any external device or file and no process may have a permanent internal state or external side-effects. The language supported is standard C with the intention to support C++, FORTRAN, and Lisp.

A unique feature of Paralex is the analysis of the parallelism of the final program. The nodes are grouped together when it is obvious that they must be run sequentially. This reduces the number of messages and hopefully speeds up the program. This condensation produces *chains* of nodes. The drawback to this occurs when multiple streams of data are flowing through the program. When this happens the grouped nodes cannot be condensed into one process.

Enterprise and Paralex both use the commercial communication subsystem ISIS to provide all communication needs. While Paralex has utilized a wide range of ISIS capabilities, Enterprise uses a limited subset. As a result, Enterprise does not have the fault-tolerance of Paralex. However, Enterprise is less committed to one particular communication subsystem.

2.3.3 Frameworks

Frameworks [26] is a graph-based parallel programming environment similar to the previous two models. The feature that is fundamental to Enterprise, the template, was developed here. The template concept is used for expressing input, replication, pipelines, and assimilating data flow from various nodes.

The communication subsystem used is the Network Multiprocessor Package (NMP) [6]. This package has some system specific limitations but has low processing overhead. It provides a minimal set of software tools necessary for distributed processing using Frameworks.

Frameworks is not discussed here since Enterprise evolved from it. Enterprise has clarified and extended the programming model. The graphical interface is more powerful. With the extension of the model and interface, the NMP communication package proved insufficient for providing the necessary software tools. Other communication subsystems were investigated. The communication subsystem is now ISIS [12] which provides system independent message processing and process groups for easier maintenance of replicated processes.

2.3.4 PIE

Programming and Instrumentation Environment (PIE) [25] for parallel processing is intended for a different architecture than the previous systems. It is targeted for shared-memory multiprocessor systems requiring medium to large grain parallelism. It provides a meta-language for modular programming. This meta-language provides the support, in a programming language independent fashion, to efficiently manipulate parallel modules, access shared data constructs, and observe the program status.

PIE uses an object-oriented approach to modularize parallelism. It has as its lowest level an *activity*, the sequential code. A *frame* is composed of activities that manipulate shared-memory directly. A *team* is a group of frames that cooperate together by interprocess and interprocessor communications. The *sensor* construct is used to monitor resource consumption or variables of the previous three parallel constructs.

PIE utilizes templates or forms that allow the user to quickly implement common parallel program techniques. For example, a master-slave, recursive master-slave, and pipeline are some of these techniques. The user is able to observe graphically both the construction and running of the parallel program. This is similar to the previous tools. Enterprise has only a static display of the connection graph. Again, like the previous systems, the user is shielded from low-level implementation details. The user is freed to concentrate at the abstract model level.

2.4 Summary

A brief introduction to the motivation of a tool like Enterprise has been presented. The Enterprise model has been summarized along with a generalized approach to developing and running code using the current implementation.

Four other software tools with similar functionality have been presented and compared to Enterprise. It is clear that Enterprise is not unique. The advantage of Enterprise is the degree of abstraction enjoyed by the user from the low-level implementations. The Enterprise model is the most recent and easiest to use of all these tools. It is not possible to say that any of these software tools produce extremely efficient code but the common goal of all of them is to enhance the productivity of the user.

Chapter 3

The Enterprise Implementation

We saw in Chapter 2 a summary of the Enterprise theoretical model. For a theory to be adequately tested, a reasonably complete implementation must be available. As with any theory, the implementation details are specific to a site or incarnation. This chapter answers how the theoretical model is implemented, in this case, on a network of homogeneous workstations. It does not address the performance considerations of Enterprise programs.

All three components of Enterprise (pre-compiler, run-time executive, and the ISIS communication system) have well-defined interfaces. Details of the pre-compiler are found in Chan's thesis [7], while Wong's thesis [28] documents the run-time executive. ISIS implementation details are summarized in the ISIS User Reference Manual [12]. What is presented here is a summary of the implementation of each component and its integration into Enterprise. For purposes of clarity, the tested implementation of Enterprise is referred to as version 0.5 while the newer version of Enterprise is known as version 0.6, unless otherwise stated.

These implementation details are broken down into five areas – the shell, communication server, the graph file, the compiler, and the run-time environments. These components are interdependent on one another to different extents and are difficult to cleanly separate. The first two are not applicable to this discussion since they are either a simple one time modification or not relevant to this thesis. Details of the first two areas are provided in Appendix B.

The last three areas are presented. First, the two versions of graph files are discussed. The first version is the tested version of the graph file (version 0.5). The second version (version 0.6) is the new, improved, and verbose graph file. It is intended to be produced by the graphical user interface (GUI) and used by the other Enterprise programs without any user intervention. Second, the Enterprise pre-compiler implementation details are discussed briefly. The pre-compiler uses the user-supplied graph file and sequential C source code files for each asset to produce the parallel Enterprise program. Third, the implementation of the run-time executive is detailed. This group of software tools is responsible for the launching and shutdown of an Enterprise program, monitoring messages and processors, and managing replicated assets. Finally, a summary of the implementation details for the version of Enterprise used for this thesis closes this chapter.

Each Enterprise program starts with the user doing a functional decomposition of the problem. These functions will determine what the graph will look like and what source code to develop. Enterprise is a tool used for the easy implementation of a parallel algorithm intended for a distributed network of workstations. Enterprise is not intended to parallelize the user's algorithm.

3.1 Graph File

The heart of Enterprise is the simple textual file known as the **graph file**. It documents the necessary connections between the various Enterprise processes and how to construct them.

Currently, there are two text file versions: the graphical user interface (GUI) version which conforms to the new model and the original version described by Wong [28]. The new graph file is intended to be created by the GUI interface and is complete (and verbose) while the older format is simpler for the user to encode.

The two graph file formats are presented next with some discussion of flaws and benefits of both. The same sample program, which is an example of a more complicated parallel program built up from simpler parallel constructs, is used in both cases. The graphical representation of the file is found in Figure 3.3. The program uses two services (Asset14, Asset15), two lines (Asset1, Asset2, Asset3, Asset13 and Asset10, Asset11, Asset12), two departments (Asset3, Asset4, Asset8, Asset9, Asset10 and Asset4, Asset5, Asset6, Asset7), and a division (Asset8). There are two replicated assets (Asset2 and Asset11) and the division has a width of three for each of its two levels. The program does not represent a real program but is an illustration of the Enterprise assets. The number of processors required is a minimum of thirty-one to to a maximum of thirty-four plus the communication server.

3.1.1 Old Graph File

The graph file format for the version of Enterprise used in this thesis is found in Table 3.1. There is one advantage to this format for the user: it requires the minimum amount of typing since only the information that is not defaulted to some pre-specified value needs to be included.

The ordering of the statements describing the graph is not important. However, the graph description must be first. Each asset (individual, department, line, or division) is completely represented on one line. The default asset is an individual. Hence, no individuals are described. This contrasts with the new version of the graph file, where placement of asset descriptions and complete descriptions of all assets composing the graph are critical to the successful use of Enterprise.

After the graph has been described, each asset that requires special libraries or machine placement instructions is listed with its name on one line and with some or

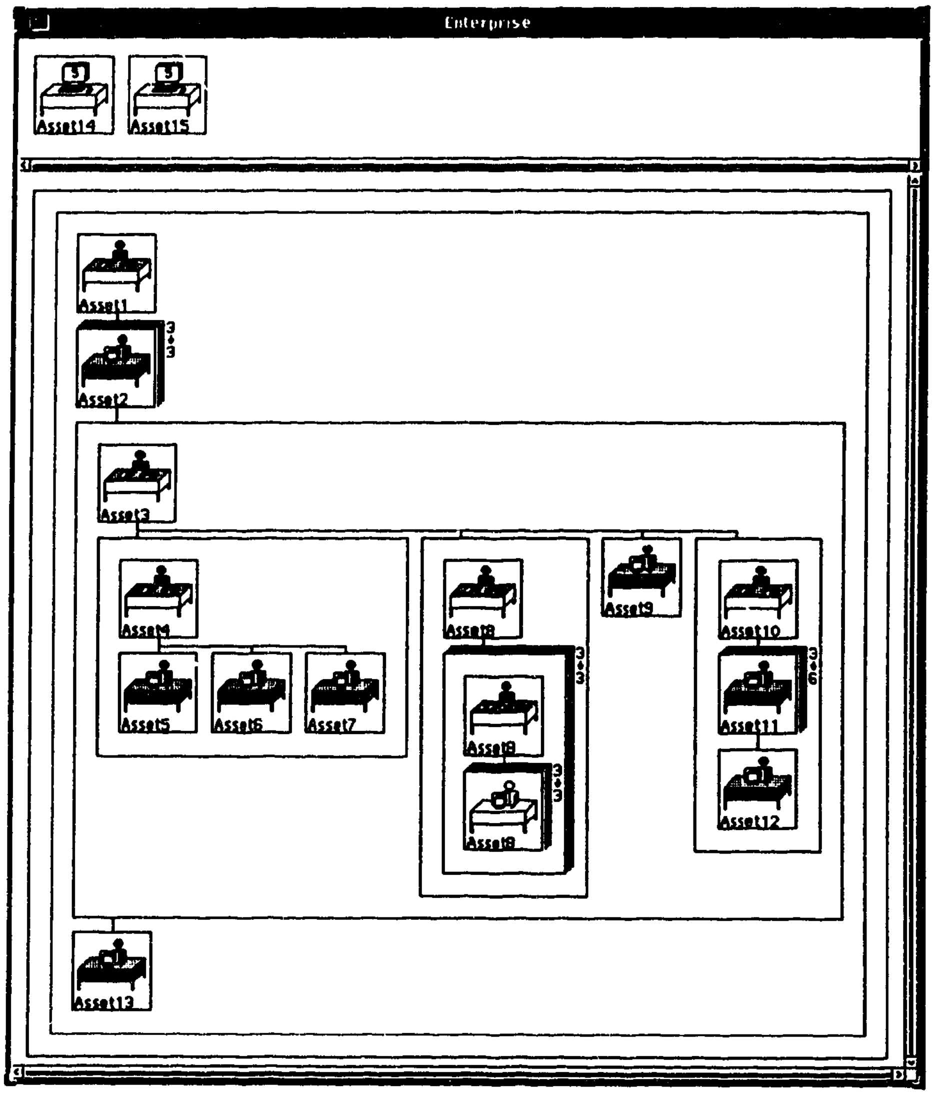


Figure 3.3: A large Enterprise program.

```
line 4 Asset1 Asset2 Asset3 Asset13
pool 3 Asset2
department 4 Asset3 Asset4 Asset8 Asset9 Asset10
department 4 Asset4 Asset5 Asset6 Asset7
line 3 Asset10 Asset11 Asset12
contract Asset11
division 3 3 Asset8
service Asset14
service Asset15
CFLAGS -DVERBOSE
Asset1
   include sass-lake
   exclude sundog
   library mylib.a -lm
Asset12
   library mylib.a -lm
   exclude sundog
Asset13
   library myotherlib.a mylib.a -lm
```

Table 3.1: Old graph format for a sample program.

all of the **include**, **exclude**, or **library** statements that apply to that asset following on separate lines. The default is the default system and communication libraries.

In the example given, CFLAGS has the compiler variable VERBOSE set *for all assets*. Asset1, Asset12, and Asset13 need the standard math library (*-lm*) and the user's library *mylib*. Asset13 also needs the user library *myotherlib*. Asset1 and Asset12 are not to be run on the machine **sundog** while Asset1 should be placed on the processor **sass-lake** if possible. All other processes can be placed on any machine present in the machine file and are to be linked with the standard C libraries and ISIS libraries.

The division asset has only a depth and breadth parameter following it. Varying the breadth parameter at each level of recursion should be a user option. The next version of Enterprise allows this.

Assets that require variable replication are identified by the key word *contract*. Contracts start with one member and expand to use all of the free machines supplied. There are no lower or upper limits applied to contracts. The term contract has been dropped in favour of a pool with variable lower and upper replication values. This simplifies and enhances the model.

One problem occurs in compiling with the old graph file. The CFLAGS option works well but is a global feature. It may be necessary to have separate compile options for each separate architecture. This has been changed in the new version. For testing purposes, a homogeneous collection of machines is assumed.

3.1.2 New Graph File

The new graph file (Table 3.2) is a textual summary of a depth-first search of the Enterprise graph. Each asset is described along with a listing of the machines it is preferred to either run on or avoid. The compiler flags are local to the asset, as are the link options. This results in a dramatic increase in the verbosity of the graph file. The local information is detailed for each asset. The user is not supposed to see this file or, for that matter, touch it. The graphical user interface (GUI) is solely responsible for translating the desired operations to the appropriate textual format. This increases the flexibility for the system to respond to site and target architectures.

Part of the file in this implementation is not used. The machine-independent flags DEBUG/NDEBUG (enable/disable debugging) and OPTIMIZE/NOPTIMIZE (enable/disable optimization) are ignored by the new version of the pre-compiler.

Replication is defined by giving minimum and maximum bounds on the size of the pool. If the lower limit is set to zero, one process is created upon startup and additional processes are started up as required until the maximum is reached. If the maximum limit is zero as well, additional processes are created until all the available machines are used.

Divisions are no longer specified with only a depth and width parameter. Each level of the parallel recursion is specified, with the last level having the special asset name, representative. This indicates where the parallel recursion stops and the sequential recursion begins. Each level is specified separately since the user may sometimes want different branching factors for different levels. The important issue in the new division format is that the source file for a division asset remains the same, regardless which level of the recursion tree is being compiled.

3.2 Source Code Caveats

In addition to the graph file, the next files created are the source code files for each Enterprise asset. Each asset is viewed as a function call under Enterprise. The user can use this asset code to test each function sequentially. In fact, the user can usually construct and test a sequential version of the program by just compiling the various asset code fragments. This allows another level of debugging before complicating the program with parallelism.

The user's code must not contain the following code fragments in order to be parallelized by Enterprise. They are as follows:

- Do not use register variables as parameters in function calls that are intended to be Enterprise assets. This may change in later versions since this is a problem of the pre-compiler not the communication subsystem.

- Delete all forward declarations of functions bearing the Enterprise asset name. A forward declaration indicates to the compiler that the function, for example

```
double foo();
```

```
Asset1 line 1 1 ORDERED DEBUG NOPTIMIZE 2
CFLAGS -DVERBOSE
EXTERNAL mylib.a -lm
INCLUDE sass-lake
EXCLUDE sundog
Asset2 individual 3 3 UNORDERED NDEBUG OPTIMIZE
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
Asset3 department 1 1 ORDERED DEBUG NOPTIMIZE 4
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
Asset4 department 1 1 ORDERED DEBUG NOPTIMIZE 3
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
Asset5 individual 1 1 ORDERED DEBUG NOPTIMIZE
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
Asset6 individual 1 1 ORDERED DEBUG NOPTIMIZE
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
Asset7 individual 1 1 ORDERED DEBUG NOPTIMIZE
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
Asset8 division 1 1 ORDERED DEBUG NOPTIMIZE 1
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
```

continued on next column

```
Asset8 division 3 3 ORDERED DEBUG NOPTIMIZE 1
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
Asset8 shadow 3 3 ORDERED DEBUG NOPTIMIZE
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
Asset9 individual 1 1 ORDERED DEBUG NOPTIMIZE
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
Asset10 line 1 1 ORDERED DEBUG NOPTIMIZE 2
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
Asset11 individual 3 6 ORDERED DEBUG NOPTIMIZE
CFLAGS -DVERBOSE
EXTERNAL mylib.a -lm
INCLUDE
EXCLUDE sundog
Asset12 individual 1 1 ORDERED DEBUG NOPTIMIZE
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
Asset13 individual 1 1 ORDERED DEBUG NOPTIMIZE
CFLAGS -DVERBOSE
EXTERNAL myotherlib.a mylib.a -lm
INCLUDE
EXCLUDE
Asset14 service DEBUG NOPTIMIZE
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
Asset15 service DEBUG NOPTIMIZE
CFLAGS -DVERBOSE
EXTERNAL
INCLUDE
EXCLUDE
```

Table 3.2: New graph format for a sample program.

returns a value that is of type ***double***, not the default type of ***int***. Enterprise redefines the function type and the forward declaration will confuse the conventional compiler which will report loader errors. The user will need these forward declarations for only the sequential version.

- Run the source code through a program like **indent** or resist the temptation to write code of the format

  ```
  if (bar == 0) return;
  ```

  Enterprise will remove the conditional statement and the return statement when modifying this code fragment. The Enterprise code will consist of just the distributed return statement without the conditional statement. Again, this is a problem with the tested version of the pre-compiler.

- Avoid using an individual element of an array as objects of a future return variable! Using an array element usually implies the messages are intended for different memory locations and are, as a result, non-blocking. The tested version of the pre-compiler does not support this yet. The user, however, may try to use a simple variable as a return variable and then assign the value to the array element. The drawback to this is that the program will block immediately waiting for the message to be returned. If the user wants non-blocking return variables, then the code must be modified by the user to use individual array elements. This is recommended only for users who are familiar with the communication subsystem and the tested Enterprise implementation.

There are currently some unresolved faults with the communication subsystem. Enterprise uses the ISIS library function **bcast_l()** to send a message to another asset. The size of the local stack must be increased by 2,516 bytes after any declaration of a return variable. This empirical finding holds if and only if the return variable (structure) is less than 2,516 bytes in size. Enterprise can be modified to introduce a dummy variable to make up this deficiency in ISIS. ISIS is not going to rectify this problem in the near future. Moving to another communication subsystem would eliminate this problem (and probably create new problems).

3.3 Compiling an Enterprise Program

The creation of an Enterprise program has several steps. At this point though, the user should have a graph file and the associated source code for each asset. The graph file is used by both the pre-compiler and the run-time executive.

The program **compile** [28] reads the graph file. From the graph, it is able to create the binary stand-alone processes for each asset. These stand-alone processes compose an Enterprise program.

After the graph file is read by **compile**, it creates the source code file for the distributed process. This source code file has the character string **_ent_** prefixed to the asset file name. **Compile** then inserts the necessary ISIS code for handling communication between processes. This includes code for attaching the process to any *service* process groups. (Recall that *services* can be connected anywhere in the graph). Replicated assets have code inserted for communicating with the replication manager of that process group. Finally, code for attaching any callee's process group (the processes connected to this node further down the graph file) is inserted. After this preamble is written, the asset source code is appended to this file. The next stage of the process is modifying the new source file by inserting the appropriate code for processing and sending messages within the user supplied routine.

The pre-compiler has two passes. It is a modified version of the GNU C compiler (gcc). It parses the asset source file searching for other asset function calls. The specific assets looked for are determined by the graph file. It then inserts the necessary ISIS function calls for decoding and encoding messages. Also, it inserts code for *lazy synchronization* of return variables. The pre-compiler creates a **.sym** and a **.chk** file corresponding to each asset. The next pass inserts the necessary variables used for *lazy synchronization*. The second pass also checks the consistency of the calling structure of the assets. That is, if a function g calls f with two parameters then f had better expect two parameters and the two parameter types must match.

The variables that are passed as parameters to another process must be pre-declared (static) variables. The tested state of Enterprise does not permit using dynamically declared variables (pointers). If the user wishes to use dynamically declared variables, the source code from the pre-compiler must be modified by the user and then conventionally (read manually) recompiled. This is only for users who are familiar with both ISIS and Enterprise programming. The use of dynamic variables will be possible in Enterprise version 0.7.

3.4 Running an Enterprise Program

After the user has successfully compiled the various processes that represent the assets in the graph file, the user is ready to run the Enterprise program.

An important question at this point is whether or not to dynamically launch the assets when needed or to statically launch all processes before starting up the Enterprise program. The dynamic launching argument stresses the sequential flow of the user's program. The beginning assets could be finished before later assets are needed. This results in better processor usage and possibly reduced numbers of processors needed. The main drawback is the implementation. A call to a non-existent process will not trigger a program error but, rather, signals the Enterprise executive to launch the missing process. However, when a process exits, there exists a constraint in that the internal state of that process cannot be required for the next use of the process.

Statically launched processes require the maximum number of processors available at run time (excluding the variable replication). If there are not enough processors, multiple processes occur on a single processor. This is not conducive to good speedups.

Static launching is, however, easier to implement in the executive. The tested version of the executive statically launches the assets. Variable replicated assets have the minimum number of processes launched.

Enterprise requires a list of machines that are available to run the program. The user must make sure the ISIS server is up and running on the machine specified as the communication host. The binary files (**run**, **asset_mgr**, **machine_mgr**, and **monitor**) composing the run-time executive must be symbolically linked to the local directory. In addition to the user's asset binary files, two user supplied files are needed – the graph file and the machine file.

The next section discusses the machine file. Following that, the next three sections detail the Enterprise launching process (**run**), the machine manager (**machine_mgr**), the asset manager (**asset_mgr**), and the monitor (**monitor**) programs.

3.4.1 Machine File

The machine file lists the machines available for running the Enterprise program. The available machines are listed in an ASCII file with one machine name per line. The environment variable **mach_file** can define the file name; otherwise, the local directory is searched for a file named **mach_file**.

The usable machines can be further restricted by using the keyword **EXCLUDE** in the graph file – that machine will never be used for that process. If the load average on an **INCLUDE** machine is above some critical value then that machine is not used until its load average has dropped to some prespecified minimum value.

Machine names on a network are unique. There are long and short formats for describing a machine name. The long format has the machine name and the symbolic Internet address appended to it. An example is **silver-vly.cs.ualberta.ca** where **silver-vly** is the machine name. The short form is just the machine name. Depending on the way the host names are bound, either the short form or long form of the host names are needed. This is system dependent.

3.4.2 Launching Enterprise Programs

To launch an Enterprise program from the command line the user types

run [*command-line-parameters*] *graph-file-name* [*command-line-parameters*].

The run command command-line parameters fall into three groups. First are the optional command-line parameters for the **run** program. There are two useful **run** parameters: *-l* for logging to a file and *-d* for monitoring executive duties. Second is the mandatory filename of the graph describing the Enterprise program. Third are any optional command-line parameters intended for the user's program. Redirection of output will result in a merger of both the user's and run-time executive outputs.

The **run** command starts the **machine_mgr**, **asset_mgr**, and **monitor** processes. Only when all the processes have successfully connected to ISIS will the distributed program be started with the appropriate command-line parameters. If the user wishes to abort the process, sending a QUIT signal to **run** will cleanly abort the entire program.

If Enterprise programs are run from a script, separating the run command by a sleep of ten seconds is recommended to allow for all processes to complete properly and allow ISIS to complete any bookkeeping it needs. This is not a problem with **run**. This is not a good solution to the problem, but it does allow unattended runs to complete. The script fails more often than not if the sleep command is not inserted.

The CFLAGS option in the old graph file must be commented out before running the Enterprise program. The current version attempts to launch CFLAGS as an asset. When it fails to do this, the entire Enterprise program aborts. This has been corrected in the version 0.6 of Enterprise.

3.4.3 Machine Manager

The machine manager [28], **machine_mgr**, is responsible for launching all processes needed by the Enterprise program. It requires a machine file containing the list of available machines and the graph file. The machine manager selects the machines from the machine file that have load averages less than some predefined threshold. The graph file is used to select the processes to run and to determine where they should be placed.

If a particular process has been specifically designated for a processor (INCLUDE), an attempt will be made to place the process on that processor. If the machine has already been taken by another process, it will place the spurned process elsewhere. If the keyword EXCLUDE was used, the process will not be placed on that processor under any circumstances.

The machine manager recruits new processors from the free processor list it maintains when asked by the asset manager during the user's program execution. If there are no free processors, the recruitment fails. This failure to recruit is not a critical failure since the user has specified that a variable number of processors is acceptable in the first place in the graph file. The startup mechanism has provided at least the minimum number of processors for this asset. The asset manager will queue the request until a processor does become available. The machine manager will maintain only one process per processor except for the user's machine.

If there are no free processors at the startup of the Enterprise program, the program can get into serious difficulties that are not really the user's responsibility. A partial solution allows multiple processes on the user's processor. This is not effective parallelism but it does allow the Enterprise program to start. The machine manager prints a message warning the user if this happens.

A cost incurred with the machine manager is the number of process slots taken by the ISIS server processor. The machine manager is currently running on the ISIS server processor. The manager uses the **rsh** command to launch all remote processes. This takes up two process slots on the server machine for each process launched. The kernel on the server must have extra process slots if larger Enterprise programs (many assets) are to be run.

For example, a program has a line of two assets with the second asset of the line having a fixed replicated factor of ten. This program requires six slots for the ISIS server, one for **run**, two for starting asset, two for the asset manager responsible for the replicated asset, two for machine manager, and twenty for the replicated asset. Thus, a total of thirty-three slots are taken for one Enterprise program. Typically, on the laboratory network machines used for experiments in this thesis, there are twenty-five to thirty process slots taken up on an idle machine. With a total of only seventy process slots typically available, there is little room for multiple users of Enterprise. The processors executing the asset processes do not have this large demand of system resources. The main demand derives from the Enterprise executive and its location on the server processor.

The version 0.6 of Enterprise has the machine manager process installed on the user's processor. This allows the ISIS server to process many users with minimal impact on the communication server processor.

3.4.4 The Asset Manager

The asset manager [28], **asset_mgr**, becomes the receptionist for one replicated asset. This is not the same as the department receptionist which is defined by the user. The asset manager is automatically spawned once for every replicated asset identified in the graph. It is responsible for receiving and forwarding all messages intended for the replicated asset to an individual worker.

If a worker process is unavailable, the asset manager stores the message in a queue for forwarding when a worker does become available. If the asset has variable replication, the asset manager attempts to recruit a new processor from the machine manager. If no work is available, the asset manager is able to dismiss idle processors. This frees up processors for other duties. The cost involved with variable replication is substantially higher than with fixed replication. The additional time needed for recruiting and setting up the new process is substantial (10-30 seconds). This includes launching the process, connecting to the ISIS server, and joining the various process groups. This cost is hidden when using a fixed replication factor since the startup times are done in parallel on each processor.

By specifying the *-d* parameter on the command line, more information is displayed relating to the asset manager duties.

3.4.5 The Monitor

A monitor process is launched with every Enterprise program. If logging (*-l*) is requested, then messages due to machine manager duties are logged to this process. If the asset manager details are needed, one can use the debugging option (*-d*). The logging file is the graph file name appended with **.log**.

3.5 Summary

The implementation details for the tested version of Enterprise have been presented. Most of the undesirable features have been modified or eliminated in the version 0.6.

Some of the startup costs for using Enterprise are a one-time-only cost. The additional environment variables and the symbolic links can be set up using a shell script.

The awkwardness of the compiler and graph file has been reduced. This includes the reduction in the number of files, controlling the degree of compilation, and the more detailed description of each asset.

The Enterprise environment has been improved by the development of the GUI controlling the user's interaction with the specifics of the implementation.

Chapter 4

Performance Evaluation

The performance of Enterprise can be measured on many levels. Chapter 3 has gone over the tasks the user must perform to install, compile, and run an Enterprise program. This was done at a general level, without any specifics for a particular parallel algorithm. However, specifics are an important performance metric. The Enterprise model is just that, a theoretical description for coarse-grained parallelism. Its performance on a network of workstations can be measured by implementing different parallel algorithms. For example, measuring how easy the parallel algorithm is to implement using Enterprise, the speedup, and assessing how different parallel templates affect the user's algorithm are some important performance characteristics.

To measure these subjective and objective metrics, four parallel algorithms have been implemented using Enterprise. Each algorithm is intended to exercise some aspect of Enterprise and provide some useful data on the Enterprise model and its current implementation.

The first algorithm is a parallel version of the Gauss-Seidel algorithm. Gauss-Seidel is an iterative algorithm for solving families of linear equations. The parallel algorithm is non-deterministic regarding the solution reached. It is implemented as a line with one member replicated. This was the first algorithm implemented using Enterprise for this thesis. Details relating to the creation of the Enterprise program from the algorithm are typical of all the presented algorithms.

The second algorithm is block matrix multiplication. This data intensive algorithm is used to contrast two different parallel constructs. The first construct, a line of three assets, has the centre asset replicated. The centre asset does the matrix-matrix multiplication. This centre asset separates the creator-of-work and the assimilator assets. The second construct, a department, combines the first and last asset functions into one asset while replicating the multiplication asset.

The third algorithm originates from graph theory. It seeks to find all nodes in a graph that are connected to one another. The particular algorithm chosen does not fit well with the Enterprise model. The implementation uses a constant polling stage which generates unnecessary traffic over the network. Despite this flaw, the program shows speedup performance using Enterprise.

The fourth algorithm is a game tree search. It was intended to contrast the difference between a department or line implementation of a recursive algorithm and the use of the Enterprise parallel construct of a division. The division was not ready in time for this thesis; only the department results are presented.

Each algorithm is presented in the same format. First, the theory behind the algorithm is presented. Second, the implementation details are described. Third, the experiments themselves are detailed. Fourth, a discussion of the observed results is presented. Finally, each algorithm is summarized as to its implementation and performance under Enterprise.

The performance of all the algorithms is marred by the performance of the communication server. The results and nature of this work uncovered two basic flaws in the implementation of the server. These problems are discussed in detail in Chapter 5.

The first and most serious flaw of the communication server is the failure to release or delete a message once it has been consumed. An unnecessary and large portion of the local memory on a processor is wasted preserving these consumed messages. The second flaw, which is related to the first one, is that as the list of unconsumed messages (in this case undeleted) increases in length, there is a sharp one-time increase in the delivery time of a single message. These two fundamental flaws of the communication server limit the performance of all the tested parallel algorithms.

All experiments were run in a laboratory of twenty SUN IPCs. Each machine has 12 megabytes of local main memory and 32 megabytes of local virtual memory. Communication between processors was via ethernet. The UNIX operating system was SUNOS version 4.1.1.

4.1 Chaotic Gauss-Seidel

The Gauss-Seidel algorithm, found in many matrix algebra texts, is an iterative procedure used to solve the family of equations

$$Ax = b$$

where A is the $N \times N$ coefficient matrix, x is the $1 \times N$ vector of unknowns, and b is the $1 \times N$ vector of constants [2]. Two sufficient conditions for convergence are either that matrix A is strictly diagonally dominant [10] or that matrix A is a positive definite symmetric matrix [2]. If either condition is met, the solution will converge regardless of the starting point.

The drawback to this procedure is that, while convergence can be guaranteed, the rate of convergence depends on how much smaller the spectral radius (magnitude of the maximum eigenvalue) of the matrix C is less than one [14]. C is defined as

$$C = (I - L)^{-1}U$$

where I is the identity matrix and L and U are derived from A,

$$A = I - L - U,$$

where L and U are the lower and upper triangular matrices with null principal diagonals respectively. The closer the spectral radius is to one, the slower the convergence. Three problems were examined with different convergence rates. The first is a diagonally dominant dense matrix which converges quickly. The second problem is a tridiagonal matrix with a spectral radius close to one. The third problem, a banded matrix, again has a spectral radius close to one. The last two problems have their convergence rates controllable by the user.

A technique for speeding up the slow convergence is successive over-relaxation (SOR). It provides an estimate for the predicted answer by taking advantage of the previous iteration. It extrapolates a new point using the current answer and the previous estimate. The formula used is

$$x_i^{\text{new}} = \omega x_i^{\text{new}} + (1.0 - \omega) x_i^{\text{old}}$$

where $\omega \in R$. For simplicity, ω was set to 1.5. In a more rigorous approach, an eigenvalue analysis would be used to determine an appropriate value for ω.

4.1.1 Design and Implementation

Baudet suggested various ways of parallelizing this algorithm in his Ph.D thesis [4]. He proposed that less synchronicity between parallel components will result in better speedup since processors do not have to wait for slower processors. The drawback is the non-determinism of the answer. This is readily apparent from the experiments. The answers were not the same from run to run, but they were similar.

To parallelize this algorithm, the answer vector is divided into independent blocks, each processed by one processor. Hence, the number of blocks is the degree of parallelism. Relaxing the requirement that all blocks sent out must return before the next round of work improves performance. Now, idle processors are utilized rather than waiting for the slowest processor to finish. But, increasing the number of blocks increases the observed non-determinism between runs. The non-determinism of the answer complicates the recognition of a valid solution. The solution error is determined locally for a segment of the solution vector. Only that segment is modified by a processor. No one processor contains the entire coefficient matrix so a global error cannot be calculated. The global error is derived from all the local errors which are based on different global solution vectors.

Estimating the global error of the solution is difficult when synchronization is relaxed. At given iteration, n, the answer vector is composed of only the returned blocks from iterations $< n$. A processor requesting a new solution vector gets only a snapshot of the answer composed of blocks of varying degrees of iteration ($< n$). A processor that returns a new solution for its block will likely have portions of the original solution vector used to derive the new solution changed while the block was being processed. Sequentially checking the experimental results confirms that the answers are acceptable, but the reported error estimates are higher than the actual estimated global error. All the errors reported fall below the user-supplied tolerance.

The local error for a block is determined during the run by calculating the estimated error of the block between two successive iterations. The local errors of the other blocks are ignored. The global estimated error is the sum of these local errors. The error is initially set to some arbitrarily high value so that all blocks must be examined before any answer is deemed accurate enough.

Writing the serial version of this algorithm is easy. However, to break up the work into modules that can be separated on to different processors is more complicated. The program is divided up into three sections. The first section receives input from the user and then divides up the work and distributes it to the processors. This is an abstract concept only. In reality, the program appears to repeatedly call a function in a loop. This program exits after scheduling all the work.

The second section acts as shared memory. It keeps the current estimate of the x vector and its error along with the size of the vector. This section responds to requests from other routines to either initialize, update portions of the vector, or return the current state of the vector. In Enterprise, this section is implemented as a *service*.

The third section does all the real work. It asks the service for the current state of the x vector. It then iterates its section of the vector until the user-supplied tolerance is met for one of three conditions. First, if the vector has met the error tolerance overall, the program exits. Second, if the group of elements that a particular replicated part of the program is working on has an error estimate less than the user-supplied tolerance, an update is sent to the service. Third, if the change in the error falls below the user-supplied tolerance, the service is updated. In turn, the service replies with the new estimated solution it has received in the interim from all the other processors. The code then loops back and repeats until the exit condition is met.

The workload on this third set of modules is uniformly distributed. The problem comes in finding enough work for the processors to do to make the communication cost worthwhile. From preliminary experimental data, the program exhibits speedup of 1.8 with problem sizes of only 500 elements and 2 processors. There is, conversely, a slowdown exhibited for the larger sized problems. For example, a problem size of 1,500 elements and 2 processors has a speedup of 0.4. This is due to page faulting, which will dominate the timing results. In this case, page faulting is much more costly than the communication delays.

Figure 4.4 shows the graph describing the parallelism. The parallelism is not obvious at first glance. The algorithm is divided into two distinct assets (**Baudet** and **GaussS**). However, the replication factor is the number to the upper right of the icon representing the asset, **GaussS**, and indicates the degree of parallelism desired. The service (**XVector**) is separated from the graph but is actually connected to all processes in the graph.

Fixed replication is used since all the segments must be running simultaneously in order for the program to finish. All blocks must be processed at least once to get the error estimate below the user-supplied tolerance. A processor gets a block of the solution vector to work on only when it starts up. There must be a one-to-one

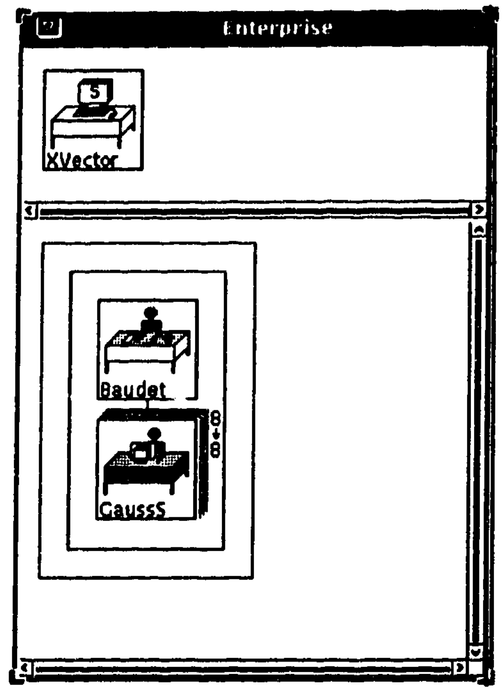


Figure 4.4: The Enterprise graph for chaotic Gauss-Seidel.

correspondence between processors and blocks.

4.1.2 Experiments

The problems were created by the user specifying a size and a blocking factor. The input will, depending on the compilation flags (CFLAGS), create one of the three problem matrices: a dense matrix, a tridiagonal matrix, or a banded matrix.

After the code was tested, a number of runs were generated in order to determine the running times. Details of the sequential experimental results can be found in Appendix A Table A.10. The parallel results are in the Appendix A Table A.11. Speedups are seen in Figure 4.5. It was quickly apparent that the sequential version took a large penalty with the increasing size of the problem. In fact, the penalty became infinite when attempting to solve a problem of size 2,000: the process consumed all available memory on the processor. The user times showed a steady, smooth increase and it was possible to extrapolate to a problem size of 2,000.

Due to message deallocation problems (discussed in Chapter 5.2.3), the parallel version for tridiagonal and banded matrices failed to complete. The slow convergence resulted in many messages being sent over the net requesting updated versions of the solution vector. The CPU load was reduced since only one or two iterations are necessary to meet the local error tolerance. The communication costs quickly soared.

Using SOR for the dense matrix problem gave poorer performance than without it. This problem converged quite rapidly without SOR, indicating a spectral radius

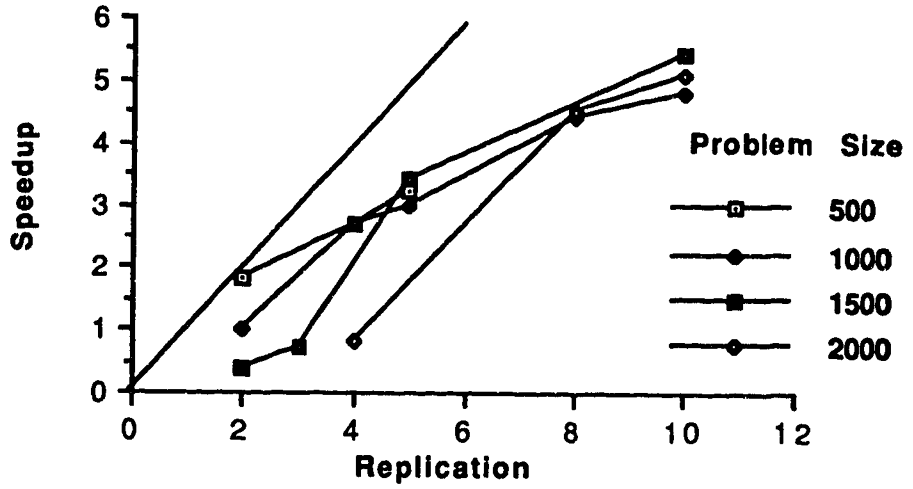


Figure 4.5: Speedup of chaotic Gauss-Seidel.

closer to zero than one. Good results were obtained by setting ω to one which had the same effect as not having SOR at all.

A pleasant surprise is the large matrix sizes that are solvable. Results for matrices with dimensions as large as 3,000, not obtainable using the serial version, are now possible by taking advantage of the local memory at each processor.

The speedup graph (Figure 4.5) shows that, with an increasing replication factor, the speedup increases. The replication factor is not the total number of processors used. The total number of processes used is the replication factor plus four (one each for the server, **Baudet**, the asset manager, and **XVector** processes). The respectable speedup of about five for a replication factor of ten is due to the chaotic nature of the work. There are no synchronization points and the work is evenly distributed to all processors with the **GaussS** process. The only bottleneck is the network itself when messages are being passed to and from the service to the replicated members. These are the best observed times of five runs. If the network is busy the speedups are lower. The observed times on different days give different elapsed times.

4.1.3 Discussion

This was the first algorithm developed using Enterprise for this thesis. The preliminary version of Enterprise tested had several flaws which were quickly eliminated. The most frustrating of these was the failure to return control to the user when the program was finished.

All the program runs exhibited some degree of non-determinism when attempting to stop correctly. Sometimes the program would relinquish control back to the shell after the program exit. Other times the program would finish but control would remain with Enterprise indefinitely.

Having a non-deterministic program confused the issue somewhat since the failures were non-deterministic. Converting the algorithm to a deterministic version was easily done by having each module solve its section, update the service, and return control to the calling program. The calling program would, after all the work sections returned a value, re-issue the work requests until all the processes were satisfied with the overall error.

This time the code worked correctly with the replicated members called in bursts of the user supplied block size. However, the running of the code is still non-deterministic as far as the successful return of control to the user at the end of the run. The explanation supplied is that the loss of packets by the communication server causes certain processes to miss their stop message. The packet loss explanation is likely since the program runs and stops correctly using five replicated members all the time. Using a fixed replication factor of ten, the program stops correctly two out of three times. The size of the messages this chaotic program generates is two to sixteen kilobytes. The non-determinism of stopping increases with increasing message size which indicates packet loss as a prime candidate for the source of this error.

Subsequent modifications to the Enterprise executive to overcome this fault have all but eliminated the non-deterministic stopping. The solution was to force a reply to the calling process for an abort or stop message.

4.1.4 Summary

Chaotic Gauss-Seidel yields a speedup of 5.4 using 10 processors for larger problem sizes. The implementation code did not require further modification after the pre-compiler stage. Non-determinism remains a problem for determining an exact answer. Slowly converging problems cause a large number of messages with respect to the computational requirements. While this is not efficient, the large number of messages revealed a flaw in the deallocation mechanism of messages in the communication subsystem. When this fault is fixed, the problems can be re-tested to clearly evaluate the effect of slow convergence using this algorithm.

4.2 Block Matrix Multiplication

Multiplying two matrices sequentially has a well-known traditional algorithm (Figure 4.6). However, one parallel technique is to split the two matrices into compatible blocks then multiply these smaller blocks independently. The product of this multiplication is then added to the existing answer matrix.

Two matrices $A \in R^{M \times N}$ and $B \in R^{N \times O}$ can be divided into blocks of submatrices to solve

$$C = AB$$

```
/* $A \in R^{m \times n}$ */
/* $B \in R^{n \times o}$ */
/* $C \in R^{m \times o}$ */
int i, j, k;
for i = 1 to i $\leq m$ increment by 1 do
    for j = 1 to j $\leq o$ increment by 1 do
        $C_{ij} = 0$;
        for k = 1 to k $\leq n$ increment by 1 do
            $C_{ij} = C_{ij} + A_{ik} * B_{kj}$;
        endfor
    endfor
endfor
```

Figure 4.6: Traditional sequential matrix multiplication.

where $C \in R^{M \times O}$ by using the formula

$$C_{ij} = \sum_{k=1}^{N/b_N} A_{ik} B_{kj}$$

where $i = 1 : M/b_M$, $j = 1 : O/b_O$ and b_M, b_N, and b_O are the number of processing blocks of each dimension [11]. The advantage in a parallel environment is that these blocks of submatrices can be multiplied in parallel by a conventional sequential method. The addition or merging of these matrices can be done by another processor. The cost of communication of the blocks is balanced against the computational price of the matrix multiplication.

4.2.1 Design and Implementation

Two versions of this algorithm were developed. The code is almost identical but the effects are quite different. The pseudo-code for the generator, multiplier, and collector tasks are given in Figure 4.8. The sequential matrix multiplication code (found in any matrix algebra textbook [2]) is given in Figure 4.6.

The first version is a line of three processes (Figure 4.7): a make-work process (**Block**) which divides up the two matrices into blocks and distributes the work, a process which multiplies the two blocks (**Multiply**), and a collector process (**Collector**). The collector process merges the submatrix into the larger global matrix, performing the necessary additions. No process expects a reply to any message. The second process is replicated to parallelize the program.

The number of messages is a function of the size of the two matrices. The dimension divided by the block size gives the number of messages. The block size used is a 100 by 100 matrix. To multiply two matrices of this size takes about one to two

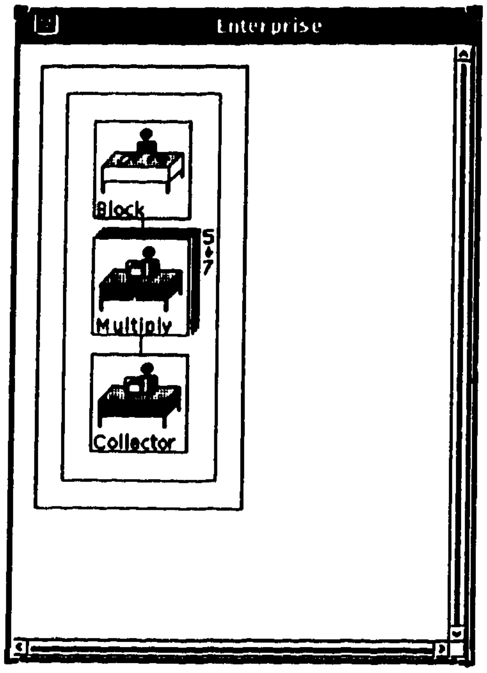


Figure 4.7: Enterprise graph for parallel block matrix multiplication – line of three.

seconds of processor time to calculate, giving a good balance between message passing and processing. As discussed earlier, the number of messages is a problem with the first process. Multiplying two 700 by 700 matrices with the previously stated block size generates 343 messages of size 160,032 bytes. These messages are generated much faster than they are consumed, which overloads the asset manager queuing space (ie. using virtual memory which results in page faults). Experimentally, the collector process had no apparent problems keeping up with ten processors.

The second version was a department of two processes (Figure 4.9). The three components are still present, however, this time the make-work process (**Block**) acts as the collector as well. The multiplication process (**Multiply**) returns the product matrix to the caller. The caller process would generate some work and then wait for the products to return. It was found empirically, that if the two matrices were completely divided up and sent off before any collection was done, the memory requirements quickly flooded the system resulting in extremely poor execution times or the program aborting due to insufficient memory. The user-code for **Block** was modified to only generate the work necessary for the inner loop of the algorithm. When the results were returned the next batch of work was generated. This synchronization permitted the programs to run using large matrices but the cost demonstrates the trade-off in performance.

The user also becomes intimately involved with the specific amount of parallelism. The number of return messages (array of blocks) must have space allocated for them. If the number of parallel tasks increases, the source code must be recompiled since only statically allocated memory is allowed for return variables. The return variables

```
/* Initialize matrices A ∈ $R^{M \times N}$ and B ∈ $R^{N \times O}$ */
A = InitializeMatrix($M$,$N$);
B = InitializeMatrix($N$,$O$);

/* Generate the work */
for i = 1 to $N$ / BlocksOfRowsOfA increment by 1 do
    for j = 1 to $M$ / BlocksOfColumnsOfA increment by 1 do
        for k = 1 to $O$ / BlocksOfColumnsOfB increment by 1 do
            /* Send a block of matrix $A$ and $B$ to a processor */
            Send-message(Proc$_{Multiply}$, $A_{ij}$, $B_{jk}$, i, j, k,
                BlocksOfRowsOfA, BlocksOfColumnsOfA, BlocksOfColumnsOfB);

/* Perform the matrix multiplication with the sub-matrices */
do while (NOT ENDOFMESSAGES)
    Receive-message($A$, $B$, Arow, ACol, Bcol, $m$, $n$, $o$);
    /* Traditional sequential matrix multiplication (Figure 4.6) */
    id_tag = {Arow, Acol, Bcol};
    Send-message(CollectorProcessor, C, id_tag, $m$, $o$);
enddo

/* Collect the work */
do while (NOT ENDOFMESSAGES)
    Receive-message(CBlock, id_tag, RowsOfCBlock, ColumnsOfCBlock);
    UpdateCMatrix(CBlock, id_tag, RowsOfCBlock, ColumnsOfCBlock);
enddo

/* Do the matrix multiplication on the sub-block */
do while (NOT ENDOFMESSAGES)
    Receive-message($A$, $B$, Arow, ACol, Bcol, $m$, $n$, $o$);
    /* Traditional sequential matrix multiplication (Figure 4.6) */
    id_tag = {Arow, Acol, Bcol};
    Send-message(CollectorProcessor, C, id_tag, $m$, $n$, $o$);
enddo
```

Figure 4.8: Pseudocode for the generator and collector in the block matrix multiplication algorithm.

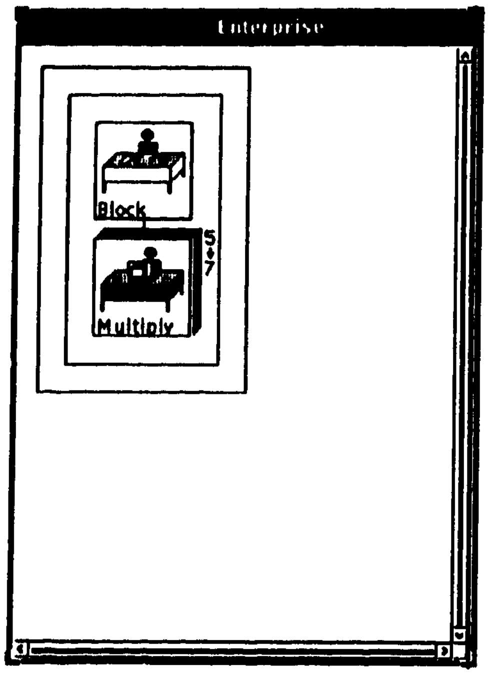


Figure 4.9: Enterprise graph for parallel block matrix multiplication – department of two.

must be statically declared not dynamically declared, but this is a problem of the pre-compiler not the communication subsystem.

The pre-compiler tested was not able to handle array elements as return variables. The code needed to be further edited by the user to successfully compile. This involved creating an array of message tokens and inserting the proper return variables in the message passing function. This is to be fixed in the next version of the pre-compiler.

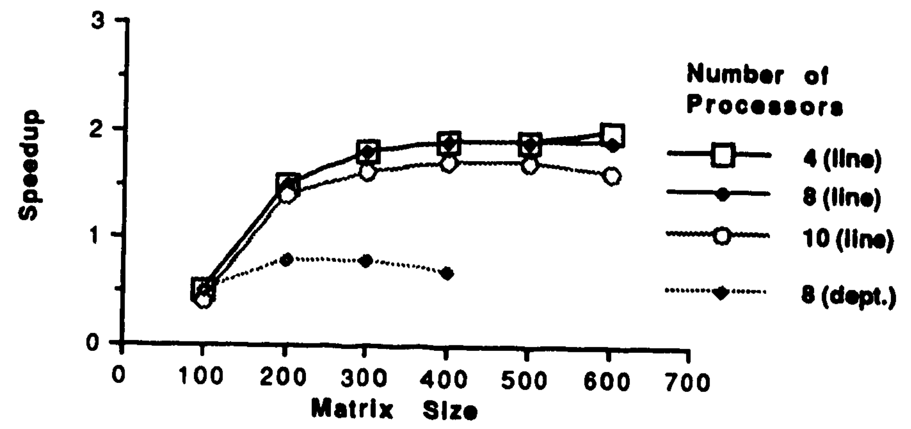
The problem still remains of unconsumed ISIS messages. Unconsumed messages are queued messages waiting for processing; it is faster to generate messages than to consume them. This extra baggage taxes the system resources of the server. The maximum problem size solvable is smaller than it need be due to resource contention from the unconsumed messages.

4.2.2 Experiments

The experimental data is found in the Appendix A Table A.12 and A.13. The speedups are shown in Figure 4.10 for both the line and department experiments.

The optimal block size was established to be a matrix of 100 by 100 elements (1.8 seconds user time). This permits a good balance between communication costs and processor utilization. Larger block sizes showed a slower consumption of outstanding messages, while smaller block sizes resulted in processor under-utilization. Messages could not get through fast enough.

The line configuration multiplied two square matrices of size 100, 200, 300, 400,



Figur^ ^.10: Speedups observed for block matrix multiplication - line of three and depar\ ^nt of one.

50(· · ·'0 together using 4, 8, and 10 replication factors. Larger problem sizes did not [illegible] .ce due to failure of the memory system. The results reported are the best elapsed times of at least four runs.

Consolidating the creator-of-work and the collector, then repeating the experiment using a replication factor of eight, resulted in the maximum problem size lim' d to only 400.

4.2.3 Discussion

Figure 4.10 shows that the speedup using a line decreases when the replication factor is increased. While the performance is not good, the observed decrease is puzzling at first.

One explanation is offered. The asset manager is receiving and forwarding large messages. When there are many large incoming and outgoing messages, physical contention for the bus is evident. These large messages interfere with the smaller messages of idle processors checking for more work. This results in more idle time and hence poorer performance.

Closer inspection of the results identified the problem as a failure to deallocate messages properly in the communication subsystem. The generator-of-work (**Block**) quickly runs out of main memory.

A brief example will demonstrate this. Using the previous example of multiplying two 700 by 700 element matrices with a block size of 100 by 100 elements, 343 messages of 160,032 bytes were generated. The memory requirements for the main process and replication manager are a maximum value of 54 megabytes. Given that the memory of the machines used in these experiments is only 32 megabytes it is obvious that the program will fail. The consumption of messages (excluding the deallocation flaw) should balance the creation of messages. When the communication subsystem fault is patched, these experiments should be re-run to see the true effect of generating an unconstrained number of messages. That is, the asset manager can still be overwhelmed by queued messages.

A minor annoyance is that the asset manager cannot be specifically placed on a particular processor. The asset manager is part of the executive and should run on any machine but, in this case, having a machine with more main memory would speed up message forwarding. Any time a processor generates a number of page faults, there is a significant performance degradation.

Consolidating the generator and collector of blocks of data multiplies the effect of this memory problem. The maximum problem size reduces from 600 by 600 to 400 by 400. Resource contention reduces the speedup to less than one. Even with the communication patch the observed speedup (if any) will be less than any speedup obtained with the line. The task of collection overlaps the task of message generation and should not be included in the same sequential process. Figure 4.10 shows the slowdowns for a department of one with a replication factor of eight.

4.2.4 Summary

Speedup is observed for the line implementation of this algorithm. The speedup is limited to a maximum of 2 because a flaw in the communication subsystem prevents it from releasing consumed messages.

Having the generator and collector of work in the same process results in a slowdown. This is not unexpected since the two tasks overlap.

Changing the code between the two implementations involves modification of the multiply routine to return the product of the two matrices to the caller rather than sending a message to a different process. The main task needs the collector code included with the generator code. The main loop generates a set number of messages and then waits for their return. This cycle repeats until all the work is sent out. This is not efficient since the number of return variables for the collector must be predeclared at compile time.

4.3 Transitive Closure

Given a graph G with n vertices (v) and m edges, there is associated with G a connectivity matrix C. The elements of C are defined as,

$$c_{ij} = \begin{cases} 1 & \text{if there is a path of length zero or more from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

for $i, j = 0, 1, \ldots, n-1$. C is also known as the ***reflexive and transitive closure*** of G. To compute C the adjacency matrix A for G is needed. The elements of A are defined as,

$$a_{ij} = \begin{cases} 1 & \text{if there is a path of length zero or one from } v_i \text{ to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

By repeatedly performing boolean matrix multiplication of the matrix A, C is derived in $\lceil \log(n - 1) \rceil$ matrix multiplications. Akl [1] describes an algorithm for a cube-connected SIMD computer that has complexity $O(n^3 \log^2 n)$. Using matrix multiplication, the load balancing of the algorithm would be evenly distributed.

This boolean matrix multiplication is a variant of the block matrix multiplication algorithm discussed earlier (Chapter 4.2), hence it was not implemented. LeBlanc and Markatos [15], in their study of shared memory versus message passing, use another version to calculate the transitive closure of G. This algorithm has definite load imbalances that are controllable by the user.

This algorithm is chosen for implementation for two reasons. First, the previous two algorithms divide the workload evenly amongst all replicated processes. Using an algorithm with controllable load imbalances would be a good test of the pool implementation. Second, this algorithm broadcasts work to peers, which is not easily implemented using the Enterprise model.

4.3.1 Design and Implementation

Figure 4.11 shows the transitive closure program set up as a line. The first asset (**StartUp**) initializes the adjacency matrix by sending a message to the **AdjMatrix** service. It then initializes the shared memory asset (**RowVect**) with a message and finally, divides up the work for each individual in the **TransClos** asset. Each individual of **TransClos** requests its block of the adjacency matrix from **AdjMatrix**. This could represent the data coming from another Enterprise program. Each row of the adjacency matrix is posted on **RowVect**, with all processors who execute **TransClos** busy-waiting until all of **TransClos** is finished with that row before a new row is posted. The completed work is then sent to **AdjMatrix** for potential delivery to another Enterprise program.

The pseudo-code for the algorithm is found in Figure 4.12. $M \in R^{N \times N}$ is the global boolean adjacency matrix that is distributed amongst the t processors (P) in blocks of rows of length N defined by *start* and *finish*.

The actual implementation was done using a busy-wait process. This was necessary because of the communication model upon which Enterprise is based. A node can either send a message and block for a reply or send a message without expecting a reply. The receiver of the message, however, must reply back to the user before responding to any further messages. This eliminates any chance for queuing a message at a server that will be responded to when the conditions are correct (i.e. new work is available). The caller must repeatedly send messages to the callee, requesting if new work is available. The callee responds negatively - hence the busy-wait. System resources (network, CPU) are tied up doing useless work. More importantly, it imposes

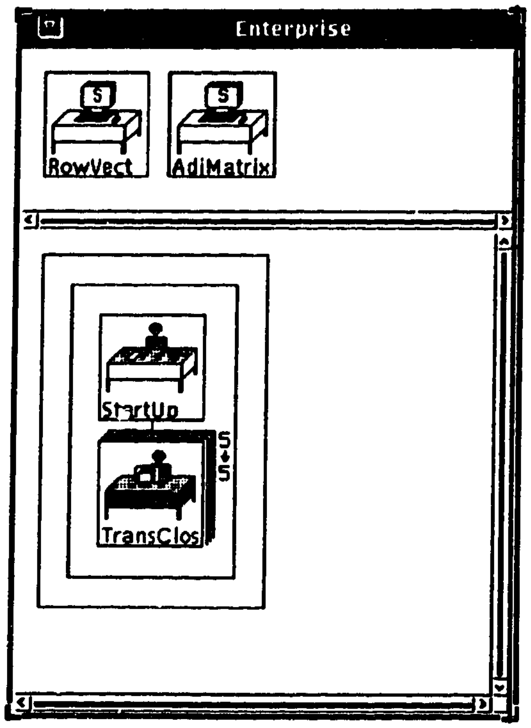


Figure 4.11: Enterprise graph for parallel transitive closure.

an unnecessary load on the network which will slow down overall throughput. This needs to be changed in the next version of Enterprise if peer-to-peer communication is desired. How to do this is beyond the scope of this thesis.

Both the supplier and consumer of work must use busy-wait. The supplier cannot contribute new work until all other processors have consumed the current work. The consumer must wait until new work is supplied. If the work is not evenly distributed, several processors will generate larger quantities of these useless messages, slowing down the overall response of the system.

An annoying and unexpected adaptation from serial to parallel was found necessary. The first asset must have a sleep statement inserted just before the return statement of the asset. The reason for this extra step is the shutdown of the overall Enterprise program, initiated by the return of the first asset. All the other assets are sent finish messages. These assets will process the finish message and propagate the finish message to their children. The problem lies with the services. They are not in this hierarchal order and after the first asset successfully finishes the program responsible for launching the assets, **machine_mgr**, sends a finish message to the next asset in the Enterprise graph. When the reply to this message is received, the services are sent the finish message.

The finish message is propagated down the graph. The important thing is that these messages generate a reply before successfully propagating the finish message to the child assets.

```
/* Globally, the adjacency matrix is $M \in R^{N \times N}$ */
/* Locally, the adjacency matrix is $M \in R^{(finish\text{-}start) \times N}$ */

/* Receive the startup message */
Receive-message(start, finish);

/* Get the portion of adjacency matrix to be processed locally */
Send-message($Proc_{AdjMatrix}$, start, finish);
Receive-message(&$M$, &$N$);

/* Process the local adjacency matrix */
for $i$ = 1 to $N$ do
    /* Attempt to post the current row for all to work on */
    if $i \geq$ start AND $i <$ finish then
        Accepted = FALSE;
        do while (Accepted == FALSE)
            Send-message($Proc_{RowVect}$, $M_{i-start}$, $i$, $N$);
            Receive-message(Posted Row, $N$, &Accepted);
        enddo;
    /* Request the current row to work on */
    else
        Accepted = FALSE;
        do while (Accepted == FALSE)
            Send-message($Proc_{RowVect}$, $i$, $N$);
            Receive-message(Posted Row, $N$, &Accepted);
        enddo;
    endif
    /* Process the local adjacency matrix */
    for $j$ = start to finish do
        if (Posted $Row_j$ == TRUE) then
            for $k$ = 1 to $N$ do
                if (Posted $Row_k$ == TRUE)
                    $M_{(j-start),k}$ = TRUE;
            endfor
        endif
    endfor
endfor
```

Figure 4.12: Algorithm used for transitive closure.

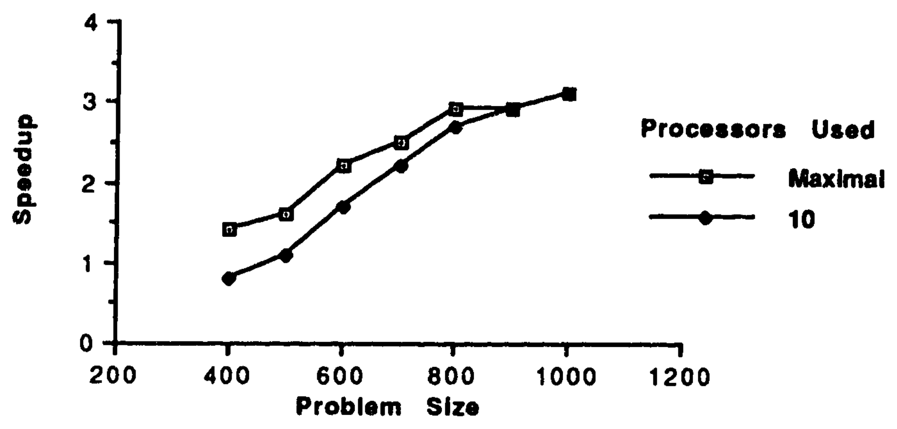


Figure 4.13: Speedup observed for transitive closure.

The solution to this would be to have the services receive the finish message after the hierarchal assets have **replied** to all finish messages. This modification is in the next version of the Enterprise executive.

The code did not need modification from the compiler.

4.3.2 Experiments

Matrices representing different sizes of graphs were tried to establish a minimal size that would yield a good balance between processing and message costs. The serial version gave user times of 0.5, 1.4, and 4.0 seconds for problem sizes of 100, 150, and 200 elements. Initially, a processing time of one or two seconds was thought to give a good balance. However, the cost of the busy-wait was not taken into account. Using a larger granularity resulted in a slowdown since the network would have a longer period of time to generate busy-wait messages. By taking a smaller processing granularity, the busy-wait messages are reduced, resulting in more productive work.

The size of blocks is set to 100 for the experiments reported here. Problem sizes of 400 to 1,000 with up to ten replicated processes were tested. Different levels of connectivity were tried with a value of 0.6 used for the presented results. Figure 4.13 shows the speedup possible using this algorithm with Enterprise. Appendix A Tables A.14 and A.15 list the experimental data.

4.3.3 Discussion

The message deallocation problems of ISIS did not cause this program to fail during the experimental runs. The only messages sent to replicated assets were small and posed no significant burden to the calling process (**StartUp**). The bulk of the messages were sent between the services (**RowVect** and **AdjMatrix**) and the individuals that belonged to the replicated asset **TransClos**. Communication services and individuals are direct, not forwarded (as is the case between individuals and any replicated asset). It appears that when using direct communication, ISIS deallocates messages properly.

The speedup graph, Figure 4.13, summarizes the results for different problem sizes and replication factors. The line representing the speedup for 10 processors demonstrates the effect of increasing size of blocks for a fixed problem size. The speedup gradually increased until it peaked with the maximum size of blocks.

The second line in the figure represents the maximum speedup observed for each problem size. This maximum occurs when the maximum block size for the problem size is used. Increasing the maximum size of blocks from 100 decreases the observed speedup.

4.3.4 Summary

Despite the implementation of a busy-wait, the program shows a speedup of 3.1 for a problem size of 1,000 using 10 processors. This is better than the block matrix multiplication solution of Chapter 4.2. Those results suggest the problem size of 1,000 is actually unsolvable.

The maximum speedup for each problem size corresponds to using the maximum size of blocks regardless of any extra processors to the minimum needed. This implies that filling the blocks and providing the maximum amount of work for the local CPU is more important than utilizing all the available processors.

The busy-wait implementation requires a smaller processor granularity (one-half second) than an expected minimum granularity of two seconds to reduce the number of wasted messages. In this case, busy-wait imposes an unnecessary extra load on network resources. Reducing the time spent in the busy-wait loop improves overall throughput even though the ratio between a single productive message and the local work produced is smaller.

A problem with the Enterprise shutdown procedure required that an extra statement be inserted for the parallel version to work correctly. This problem should disappear in the next version of the Enterprise executive.

This algorithm does not implement easily using Enterprise. The communication model used by Enterprise is not suited for peer-to-peer program communication. Perhaps another asset based on the round-table or board-room meeting analogy should be developed.

The pre-compiler output did not need further modification to run correctly.

4.4 Alpha-Beta Search

The following is a summary of the discussion found in Nilsson [22] and Akl [1] on alpha-beta search of combinatorial spaces.

A min/max tree can represent the state space of a game. The state of the game is represented by a node, while the arc connecting two nodes represents a legal move. That is, the possible combinations of legal moves that lead from a given state of a game to another can be traversed. However, the growth of the tree is exponential. This growth limits the size or depth of the trees that can be exhaustively searched within some fixed period of time. For example, searching the entire state space of chess is not possible.

Limiting the maximum depth searched results in a series of terminator or leaf nodes. These leaf nodes are be evaluated by a heuristic evaluator function, resulting in an estimated score. Traversing the tree results in all parent nodes visited acquiring a score based on either maximizing or minimizing the score of the child nodes. Propagating these scores up the tree, eventually, the root node can be assigned a final score which guides the selection of a move. The game moves to a new state and the process starts over.

Alpha-beta search is an algorithm which can prune or eliminate sub-trees from the search. By eliminating potential search paths, either deeper trees can be searched or solutions found sooner. At each level of search (ply), a provisional score is assigned while the children of that node are evaluated. A maximizing node can never have the provisional score (alpha) decrease. Conversely, a minimizing node can never have the score (beta) increase.

To stop searching below a minimizing node, the provisional score of one of its child nodes must be less than or equal to the earlier alpha scores. The parent node's provisional score is now its final score since the score needs only to be a bounded value. Similarly, the search is discontinued for a maximizing node if the beta score of the earlier scores is less than the beta score of one of its child nodes. The parent node has its final score set to the provisional score.

This algorithm must search a minimal subset of nodes in the tree to establish a score. This puts a minimum bound on the number of nodes visited. This is dependent on the branching factor and depth of the tree. For a given depth D and branching factor B the minimum number of leaf nodes visited, N_D, is given by [22]

$$N_D = \begin{cases} 2B^{D/2} - 1 & \text{even Depth} \\ B^{(D+1)/2} + B^{(D-1)/2} - 1 & \text{odd Depth.} \end{cases}$$

One way to parallelize this search is to search all the child nodes connected to a node in parallel [20]. This particular algorithm distributes work while traversing the leftmost branch of the tree. This leftmost branch is called the ***principal variation*** of the tree. All nodes connected to this branch must be examined while other branches of the tree have the potential for being pruned from the search space.

The potential speedup for random trees is limited to $O(\sqrt{\text{number of processors}})$ due to synchronization and load imbalances in the search [9]. (Idle processors waiting for work are expensive.) By careful ordering of the sibling nodes, good speedups can be achieved since the limiting bounds found early in the search will result in maximum pruning of future branches. Small branching factors imply only a few processors are

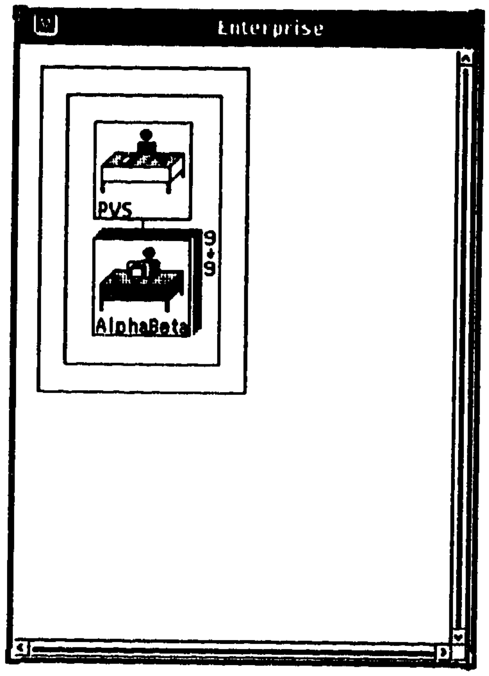


Figure 4.14: Enterprise graph for alpha-beta search – department or line.

necessary to achieve maximal potential speedup since there is only a limited amount of work to do in parallel.

4.4.1 Design and Implementation

The alpha-beta algorithm is implemented as either a line of two assets or a department with one subordinate asset (Figure 4.14). In either case, the implementation is the same. The second member of the line or the department replies back to the main program and new work is sent out. The functionality between a line and a department at this small scale is identical and the results were identical.

The alpha-beta search is divided into two assets, as seen in the Enterprise graph in Figure 4.14. The algorithm for the first asset is found in Figure 4.15. The algorithm for the second asset is any standard alpha-beta algorithm [1]. The first asset (**PVS**), the recursive call, traverses the principal variation of the tree. When the granularity of the recursion reaches the user's threshold, this routine calls one of the individuals that compose the second asset (**AlphaBeta**) to sequentially search a branch of the tree rooted by one of the child nodes. After all the work has been posted and received back, the waiting **PVS** asset sets the score (alpha or beta) for that node and the recursion unwinds. Unwinding the recursion increases the amount of work at each level being done by each **AlphaBeta** process. Each level further up the tree should have better information available to prune any unsearched branches if the ordering is favourable.

The program used in these experiments is a simulation tool for testing different

```
/* User parameters */
depth = Depth of tree search.
width = Branching factor at each node.
granularity = Switch from parallel to sequential searching at this depth.
root = root node of tree to be searched.

/* Start the recursive search */
score = PVS( root, depth, $-\infty$, $+\infty$, width, granularity);

/* Recursive principal variation search */
PVS ( treeRoot, depth, $\alpha$, $\beta$, granularity)
begin
/* Switch from parallel to sequential search */
    if (depth $\leq$ granularity) then
        score[1] = AlphaBeta( treeRoot, depth, $\alpha$, $\beta$, width);
        return score[1];
    endif;
/* Generate all possible moves from this node */
    treeRoot.son = GenerateMoves(treeRoot);
/*Move down the principal variation */
    a = $-\infty$;
    b = $\beta$;
    score[1] = PVS( treeRoot.son[1], depth-1, -b, -a, granularity);
    a = -score[1];
/* Search the rest of the child nodes in parallel */
    parallel i = 2 to treeRoot.moves do
        score[i] = AlphaBeta( treeRoot.son[i], depth-1, -b, -a, width);
    endparallel;
/* Find and return maximum score */
    score[1] = MAXIMUM(score);
    return score[1];
end.
```

Figure 4.15: Algorithm for parallel alpha-beta search.

search algorithms [21] on random trees with a constant branching factor. The program inputs are the search depth, branching factor, and the probability factors. Each branch is ordered by a probability factor which controls where the eventual answer will be found. Giving a large probability factor to one branch increases the likelihood of finding the correct answer within that particular branch. Changing the probability distribution changes the search pattern. The numbers used to represent the probability factor are integers and must sum to one hundred. Placing a high probability factor on the first branch allows the answer to be found in the left-most branches (a best ordering); a high probability for the last branch makes the right-most branch of the tree the most likely place to contain the answer (near worst-case ordering).

4.4.2 Experiments

The program compiled after some user intervention. The return values from other processes needed to be inserted into an array in order to have asynchronous operations. The pre-compiler cannot currently do this. After some user intervention, the modified source code was conventionally compiled without further problems.

The serial version was tested with several tree depths and branching factors to get a set of values that had a running time of greater than three hundred seconds. This size of problem should show some benefit of parallelization. Each experimental data point represents the best speedup after at least five runs.

Each experiment, for a given tree depth and branching factor, represents a change in granularity. The granularity level of parallelism of a problem is the point where the problem changes from being solved sequentially to parallel.

There were three groups of experiments. They investigate the effect of the number of processors, tree depth and branching factor, and the distribution of probability factors of the branches on speedup. However, the experiments showed that there is no difference in the speedup when reducing the number of processors available for parallel work. Only marginal differences were observed when changing the depth of a tree while keeping the branching factor constant. The data for these experiments is found in Appendix A, Tables A.18 and A.17. The problem lies with the failure to deallocate messages by ISIS.

The one set of experiments, which is presented, investigates the effect of changing granularity with different ordering of the probability factors. Varying the number of processors did not make any noticeable difference in the observed speedups.

The experiment (Figure 4.16) investigates the effect on the observed speedup when the probability factors are changed (Appendix A, Table A.18). The probability factors are varied from a perfect probability (best ordering) to a uniform distribution, and finally, to nearly the worst case.

The probability values for the nine branches were set to 100, 8(0); 12, 8(11); 8(0), 100 respectively. The term, 8(0), means the next eight probabilities were set to zero. The tree breadth was held constant while the depth was varied to generate sufficient work. The best ordering required a depth of thirteen; the uniform ordering required ten; the worst ordering required nine. The different depths were chosen to

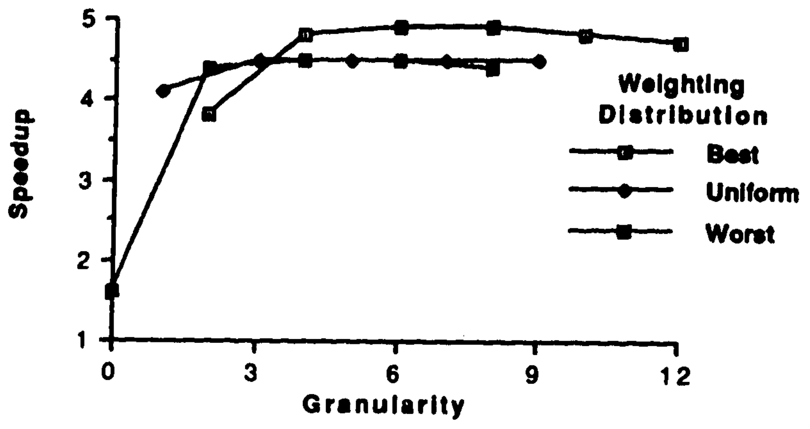


Figure 4.16: Speedup observed for different granularities with an alpha-beta search on a tree of breadth nine and depth thirteen (best), ten (uniform), and nine (worst) using nine replicated processes and different weighting of the branches.

give sufficient sequential time of greater than three hundred seconds. The choice of the breadth factor was to ensure all the work available would be done in parallel simultaneously with the maximum number of usable processors available. This is a physical limit set by the kernel (Chapter 3.4.3).

4.4.3 Discussion

The experiment shows the difference in speedup with trees of different branching probabilities (Figure 4.16). When the branching factor is optimally ordered, the speedup observed is maximized. If the answer is found in the leftmost branch, the other branches that are searched in parallel only have to search the leftmost branches of the subtree in order to cut off any more searching. This results in a greater observed speedup than with uniform or worst-case weighting. The best speedup was observed using a granularity of seven. At this granularity, the optimal balance was struck between parallel and sequential work.

The uniform and worst-case branch probabilities require searching of more of the subtrees by each processor. This extra work contributes to the lower speedups. The balance between parallel and sequential work was at a granularity of seven for uniform weighting and a granularity of five for worst case weighting.

4.4.4 Summary

A variety of configurations were tested using parallel alpha-beta search. The following conclusions were reached: the ordering or weighting of the branching affected the speedup; if the ordering leads to a perfectly ordered tree, the best speedup was observed; however, the uniform and worst-case ordering do benefit from parallel searches.

The communication server was found to affect the performance of this computationally intensive algorithm. Unlike a data intensive algorithm (block matrix multiplication), alpha-beta search has modest message size requirements. However, the number of messages did affect the time of delivery of reply messages. This delay of message delivery occurred after a threshold of accumulated messages was reached. A discussion of the effect of the communication server is found in the next chapter.

4.5 Evaluation Summary

Four algorithms were developed using Enterprise. Line matrix multiplication and transitive closure did not need any user changes in the output of the pre-compiler. Department matrix multiplication, alpha-beta search, chaotic Gauss-Seidel needed the user to modify the pre-compiler generated source code.

Decomposing the problems for the Enterprise model was simple for three of the four algorithms. The one problematic algorithm, transitive closure, needed substantial modifications before the Enterprise model worked.

All observed speedups are tainted by the major ISIS flaw of not deallocating messages after the messages are consumed. Tasks that generate minimal numbers of messages do better (chaotic Gauss-Seidel) than tasks that generate a lot of messages like matrix multiplication. A more detailed discussion of the ISIS failures are found in the next chapter.

In general, the observed performance was not impressive. Chaotic Gauss-Seidel should have near linear speedups since non-determinism should statistically give better performance. The matrix multiplication should have better results but memory constraints discouraged any reasonable speedups. Transitive closure, using a different algorithm than boolean matrix multiplication, does show better performance but at the cost of implementing a busy-wait loop in the consumer and producer tasks. Alpha beta search, while giving near theoretical speedups for smaller problems, suffered from message delivery delays if the number of accumulated messages rose above a certain threshold.

Using Enterprise to generate the code for these four algorithms results in productive time spent by the user thinking about the algorithm and its decomposition, and time not spent debugging low-level communication code. This is the most desirable feature of Enterprise. To generate working and debugged distributed parallel code with minimal effort on the user's part makes distributed parallel processing more of a science than an art. Users with little or no distributed programming experience can now test and develop parallel programs without worrying about low-level implementation details. The abstraction of these low-level details frees the user to concentrate on the important details of the science of distributed parallel processing.

Chapter 5

Performance Analysis

Previously, we have seen the performance of Enterprise examined from the point of view of the programming model and the implementation of four parallel algorithms. The performance of the programs was not outstanding. Which component of Enterprise contributed to the observed reduced problem speedups? Were the parallel algorithms at fault?

This chapter examines the cost of passing messages utilizing different Enterprise constructs using ISIS [12] and a comparable structure using the Network Multiprocessor Package (NMP) [6]. The efficiency of the asset manager is examined since a manager should be a minor overhead to the communication cost between two processors.

The efficiency of the programming model and the communication subsystem is a large factor in the observed speedup of a parallel algorithm. The performance of the chaotic Gauss-Seidel is used to evaluate the cost of the replicated-asset manager program. This algorithm is implemented using Enterprise and NMP.

5.1 Message Passing Costs

In coarse-grained distributed processing, the balancing of message costs and the anticipated CPU cost is an important factor in performance evaluation. If the number of messages overwhelms processors, memory management (page-swapping) will dominate the performance costs. If processors are starved for messages, processors will remain idle with little effective work being done. This also happens if the messages are large in relation to the amount of CPU processing or if very few messages are being generated.

The current version of the Enterprise executive is built upon the ISIS communication subsystem [12]. ISIS is, in turn, built upon the User Datagram Protocol (UDP) [16]. ISIS has built a large infrastructure to buffer the user from this unreliable communication protocol. In contrast, the Network Multiprocessor Package (NMP) [6] uses Transmission Control Protocol (TCP) [16]. This reliable message passing enables NMP to be quite small and act as a buffer between the user and raw sockets. NMP is a set of library routines built upon the sockets using UDP or TCP. The NMP version tested is based on TCP. To be fair, ISIS contains more capabilities than NMP, but Enterprise uses only a small subset of ISIS and, for the most part, NMP suffices.

This leads to an important question. What is the cost of sending a message of size x bytes from processor P_1 to processor P_2? Enterprise, using replication, created an asset manager that manages processes. Introducing this extra level of bureaucracy has an associated cost. The following experiments try to quantify this cost and isolate the reasons for the poor performance in Chapter 5.

- Determine the cost of sending 1,000 messages between processor P_1 and processor P_2 having the two processes communicate directly. This will contrast the cost of using a reliable versus an unreliable protocol.
- Repeat the above experiment but introduce a simple manager P_3 who only relays messages between P_1 and P_2. This will give an estimate of the cost of one level of indirection.
- The final experiment is to use the Enterprise asset manager and examine the cost of sending a message from P_1 to a pool of one P_2. That is, what is the cost of the current implementation of Enterprise?

All of these experiments issue a message and then block until the message reply is received. This provides an estimate for the round trip cost between P_1 and P_2. All runs were done on a quiescent subnet and were repeated at least 4 times on different days. The results varied, as expected. The results reported are based on the best elapsed times. It should be noted that system resource usage varied but, generally, not in direct proportion to the elapsed times. There does not seem to be a clearly visible trend.

5.2 Experiments

5.2.1 Line of Two Individuals

Tables 5.3 and 5.4 illustrate the cost of interprocess communication of small messages using either ISIS or NMP. The user, system, and elapsed times are much lower for NMP, indicating that there is an overhead processing messages in ISIS. There is an unexplained anomaly in the ISIS times. Some of the ISIS runs reported low user times (similar to NMP times) but the elapsed times are similar to the other ISIS timings. This is attributed to the internal workings of ISIS being distributed among several processors which are not being monitored. That is, the request went out quickly but ran into problems at the server. None of the system times demonstrated this phenomenon.

Message	Times (seconds)			Messages	
Size (bytes)	User	System	Elapsed	Sent	Received
40	0.04	0.94	2.90	1,000	2,000
400	0.21	1.18	4.10	1,000	2,000
800	0.37	1.11	4.98	1,000	2,000
1,600	0.58	1.56	7.83	1,000	2,000
2,800	0.55	1.85	10.20	1,000	2,002
3,200	0.98	3.59	11.62	1,000	2,880
4,000	1.35	4.27	13.38	1,000	3,082
8,000	1.91	8.66	28.56	1,000	5,125

Table 5.3: Best elapsed times for NMP – line of 2 individuals.

Message	Times (seconds)			Messages	
Size (bytes)	User	System	Elapsed	Sent	Received
40	2.81	2.77	11.71	1,000	1,002
400	3.46	2.46	12.94	1,000	1,002
800	3.78	2.73	14.91	1,000	1,002
1,600	4.37	3.28	17.53	1,002	1,004
2,800	4.74	4.16	22.73	1,002	1,004
3,200	5.04	4.18	23.71	1,002	1,004
4,000	5.59	4.72	27.66	1,002	1,006
8,000	12.13	10.66	48.06	3,002	3,011

Table 5.4: Best elapsed time for ISIS – line of 2 individuals.

The interesting point is the number of messages sent and received. TCP requires an acknowledgement for every message. This response doubles the number of messages received since a message must be received before the receiver returns the message to the sender. If there were exchanges of messages between the two processes, then these acknowledgements could be piggy-backed onto outgoing messages. UDP does not require this so the messages received correspond roughly to the messages sent.

In both cases, the number of received messages steadily increases with increasing message size. This phenomenon was observed to a lesser degree on a busier subnet. On this other subnet, there was a sharp increase upon moving from 4,000 byte messages to 8,000 bytes while prior to 4,000 byte messages the received message numbers stayed very close to 2,000.

The processes spawned by NMP are, in reality, owned by the **inetd** program (internet process daemon). The number of messages generated increases at this 4 kilobyte barrier. Examining the source code of **getrusage()** (a UNIX system function to get process statistics) revealed that messages sent or received had nothing to do with the number of packets generated per message. It is possible that the daemon is responsible for some of the received messages, particularly when a page boundary is exceeded. This is a supposition only, since source code is not available for the daemon

Message	Times (seconds)			Messages	
Size (bytes)	User	System	Elapsed	Sent	Received
40	0.14	0.94	4.92	1,000	2,000
400	0.12	1.15	7.90	1,000	2,000
800	0.60	1.28	9.82	1,000	2,000
1,600	0.51	1.33	15.45	1,000	2,000
2,800	0.35	2.94	20.37	1,000	2,000
3,200	[illegible]	3.27	23.08	1,000	2,792
4,000	[illegible]	4.35	26.42	1,000	2,827
8,000	[illegible]	4.92	57.21	1,000	5,077

Table 5.5: Best elapsed time for NMP - line of 3 individuals.

and the manuals are not explicit in this area.

Rerunning with larger message sizes (400 to 1,000 kilobytes) and fewer repetitions generated large numbers of messages being received to the number of messages being sent. The number of received messages did go up in relation to the message size. However, the number of messages received varied without significant changes observed in elapsed times. There were no page faults reported. This phenomenon is not critical in lightly loaded networks or machines but a heavily loaded machine or network will be a bottleneck in the performance of a distributed program.

5.2.2 Line of Three Individuals

Introducing a third process to simply relay messages between the two main processes should only double the cost of transmission. The cost of a manager should be comparable under either system.

Tables 5.5 and 5.6 summarize several statistics from the best elapsed times observed from at least four successful runs. The elapsed times show that the NMP version of the simple relayer has a cost of double the direct cost. However, the ISIS version did not manage to achieve this low overhead unless messages of less than 800 bytes were sent. The cost of implementing a simple relayer using ISIS adds 50 to 100 percent to the *expected* elapsed times.

Overall, NMP generated much better user and system times; this implies that the machine could do more useful computations rather than consuming resources sending and receiving messages. This does assume that the message sending process operates in a non-blocking mode to the program.

There is a definite additional cost to relaying messages in ISIS beyond the cost of simply forwarding the message. This leads to the evaluation of the asset manager implemented using the ISIS primitives for process groups.

Message	Times (seconds)			Messages	
Size (bytes)	User	System	Elapsed	Sent	Received
40	4.72	4.47	24.51	2,002	2,006
400	4.60	4.32	27.99	2,001	2,006
800	4.85	4.77	32.67	2,000	2,006
1,600	5.68	5.26	54.33	2,009	2,013
2,800	6.31	5.95	79.56	2,022	2,024
3,200	6.44	6.19	82.00	2,024	2,026
4,000	6.61	6.61	125.84	2,048	2,040
8,000	12.40	10.45	126.27	3,079	3,088

Table 5.6: Best elapsed time for ISIS – line of 3 individuals.

Message	Times (seconds)			Messages	
Size (bytes)	User	System	Elapsed	Sent	Received
40	5.74	2.89	1,042	2,066	5,270
400	5.56	3.00	1,043	2,064	5,270
800	0.70	7.31	1,855	2,117	5,575
1,600	5.76	8.78	1,828	2,109	5,536
2,800	10.72	6.81	2,001	2,118	5,623

Table 5.7: Best elapsed time for ISIS – pool of 1 individual.

5.2.3 Individual to a Pool of One

The results from Sections 5.2.1 and 5.2.2 lead one to expect that the cost of a manager would not be an excessive burden. ISIS provides a series of functions for management of process groups. An individual process can belong to several groups. In fact, Enterprise takes advantage of this to provide a stream of control and the allocation of replicated member resources.

The crucial difference between the previous experiments is that the asset manager is not used when individuals talk to one other. Given a pool of one, the cost of the asset manager should be comparable to the cost of a line of three individuals plus some overhead for maintaining some accounting for free processes in the pool.

Table 5.7 shows a shocking difference in the observed elapsed times! These times are not a mistake in typing – 1,042 seconds is quite different from 4.9 seconds for NMP (Table 5.5) and 24.5 seconds for ISIS (Table 5.6) for 1,000 40 byte messages.

The user and system times do not reflect this dramatic increase. Initially, it was thought that something that the asset manager is doing is consuming large amounts of time. Upon examining the source code for the asset manager, there is not much that it is doing that would account for such observed delays. The caller sends the message to the asset manager which queues the message. The manager, after dequeuing a message, forwards the message to the next available pool member (in this case, there is only the one pool member). The reply from the pool member is sent directly to the caller, bypassing the asset manager. The freed pool member then contacts the asset manager for more work.

Timing code inserted into the asset manager and machine manager code determined where the program spent its time waiting for messages. The results were surprising. Depending on the size of the messages, the asset manager spent the bulk of its time waiting for a message to arrive. For 2,800 and 3,200 byte messages the time was spent either waiting for the machine manager or forwarding messages to the worker. The idle time reported while waiting for a message dropped from 95 percent to less than 25 percent.

Another interesting statistic came from the ISIS server program, **cmd**. This program can force ISIS programs to dump a large amount of statistics about their current state. It states in the *ISIS User Reference Manual* [12], Section 19.11.1, that if a program is having difficulties in performance to check for either a large number (greater than 300) of unconsumed messages, or a large block of memory consumed (1,000 kilobytes). There were 271 unconsumed messages early on in one process while sending 8,000 byte messages 1,000 times.

The guilty process is the first asset in the line. This process is started with an **execve()** function call from a forked **run** process. This process, using the ISIS routine **bcast()**, sends a message to a process group and expects a reply to be placed in a user declared array. No message receiving is done except at this point. It appears that the **bcast()** command is not consuming the messages after they are returned or sent. Discussions with the ISIS developers indicate that they are aware of this problem and that it may be fixed in the next release of ISIS (Version 2.2.6).

The caller sees one message to the asset manager and one from the replicated asset worker. This is similar to the line of two or three individuals. There are several other messages generated, but they should not dramatically increase the elapsed times. The asset manager would be rapidly overwhelmed with messages if there were not enough idle workers, but in this case, the caller blocks until the message is received before generating a new message.

During the the alpha-beta experiments (Chapter 4.4), a definite threshold of the number of messages existed where a dramatic slow-down was observed. The message-passing program described above was modified to time the delay between sending a message and receiving the reply. Since the pool member does no processing, the cost of the ISIS overhead is clearly identified. It is assumed that the physical transport of the message between two processors is negligible.

Figures 5.17 and 5.18 summarize these results. Each line describes the effect of a particular message size. The sizes chosen reflect the message size of the alpha-beta algorithm except for the forty-byte messages. This size fits within one ISIS packet. In each case, there is a definite threshold where the turnaround time increases to a new relatively constant cost. The size of a message changes where the threshold occurs. Unfortunately, there is no clear trend where increasing message size shifts the threshold to fewer messages. The increasing noise at larger message sizes results from the increasing number of packets delivered per message.

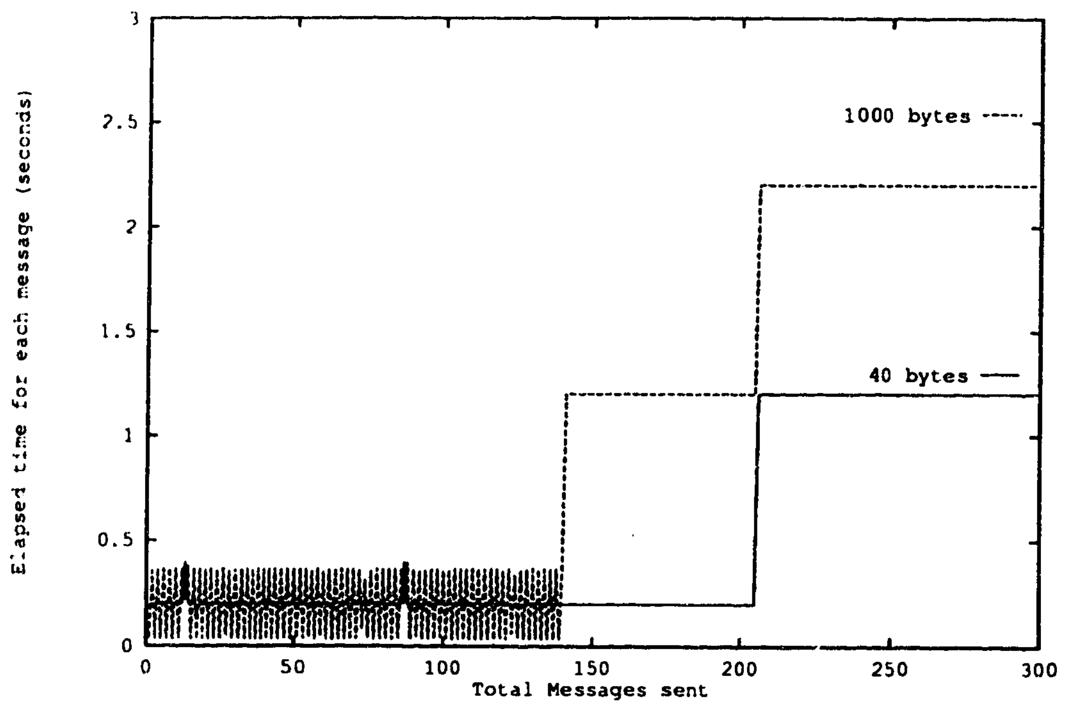


Figure 5.17: Time delay between sending 40 and 1,000 byte messages and receiving reply – ISIS pool of one.

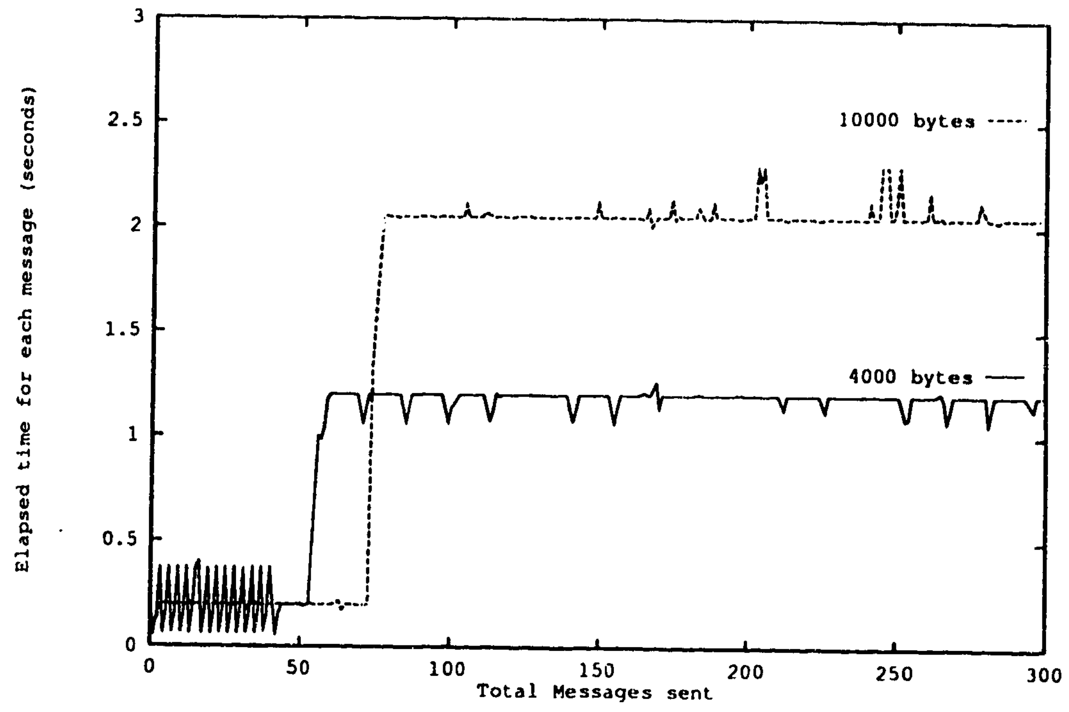


Figure 5.18: Time delay between sending 4,000 and 10,000 byte messages and receiving reply – ISIS pool of one.

Increasing the message size has two effects. The first effect is in the threshold where message delay time increases. Second, the magnitude of the delay varies. These effects are not linear with increasing message size. While the effects are clearly observable, an explanation is not easy since the ISIS source code is not available. Clearly, with small messages, memory consumption is not applicable but the cost of retaining these consumed messages by ISIS is considerable once the threshold has been reached. Adapting the user's code to avoid these implementation details is neither desirable nor, in most cases, possible.

This inability to properly deallocate messages is a major factor in the mediocre performance of the small test suite of programs used to evaluate the current state of Enterprise. When this problem is fixed, the inefficiencies in the asset manager can be properly evaluated.

Section 5.3 demonstrates the efficiency of the two communication systems by contrasting the difference between an NMP implementation of a chaotic Gauss-Seidel and the Enterprise version using ISIS. In both cases, the number of messages sent out from the main program is one for each pool member. More importantly, no reply is expected and no messages should accrue.

5.3 Comparison Between ISIS and NMP

One of the programs (chaotic Gauss-Seidel) was rewritten to work in a similar fashion under NMP. This is to evaluate the cost of converting between communication subsystems and the performance.

The details of the algorithm are discussed elsewhere (Chapter 4.1). The main difference between the two implementations is that the static structures imposed by the current version of Enterprise are not present in the NMP version. The dynamic structure size is identical to the static structure messages for 1,000 and 2,000 problem sizes. The Enterprise version uses a pool of n processors with the 0^{th} pool processor writing out the answer. The NMP version has the main process delegate the work to the other processes directly without using a manager.

Tables 5.9 and 5.8 summarize the two experiments. As before, the numbers represent the best observed elapsed times of at least four runs. The runs were performed on a quiescent net with homogeneous machines.

Both versions show the same problems when the processors are forced to swap memory pages due to the large problem size. The performance of the algorithm critically depends on avoiding page faulting. The critical size using the available machines is approximately four megabytes of storage space for the segment of the matrix A. If this size is exceeded (for example 1,500x750) then a 0.2 slowdown is observed. Conversely, using 10 processors and a problem size of 2,000 yields a speedup of 9.2 for the NMP version. The ISIS version has a 0.4 and 4.8 speedup factor for the same problem.

The problem occurring with a large number of undeallocated messages does not occur since the convergence of this particular matrix is fast.

Number of processors	500 Time	500 Speedup	1,000 Time	1,000 Speedup	1,500 Time	1,500 Speedup	2,000 Time	2,000 Speedup
Serial	9.89		29.62		66.99		120.1‡	
2	5.62	1.8	28.71	1.0	186.9	0.4		
4	3.63	2.7	10.97	2.7			164.1	0.7
5	3.06	3.2	9.71	3.1	19.85	3.4		
8			6.78	4.4			28.56	4.2
10			6.18	4.8	12.41	5.4	25.24	4.8

‡extrapolated value using cubic polynomial

Table 5.8: Elapsed times in seconds and speedup for distributed Gauss-Seidel using ISIS

Number of processors	500 Time	500 Speedup	1,000 Time	1,000 Speedup	1,500 Time	1,500 Speedup	2,000 Time	2,000 Speedup
Serial	9.89		29.62		66.99		120.1‡	
2	3.22	3.1	11.77	2.5	441.2	0.2		
4	2.16	4.6	7.14	4.1	15.87	4.2	285.9	0.4
5	2.04	4.8	6.00	4.9	12.83	5.2	164.1	0.7
8			6.96	4.3			14.43	8.3
10	2.46	4.0	6.22	4.8	13.11	5.1	13.06	9.2

‡extrapolated value using cubic polynomial

Table 5.9: Elapsed times in seconds and speedup for distributed Gauss-Seidel using NMP.

The difference in speedups is directly attributable to the overhead ISIS contributes and the cost of a pool manager. Chaotic Gauss-Seidel will demonstrate a super-linear speedup due to the non-determinism of the overall answer.

Creating the NMP version was tiresome and took a long time to debug. This exercise demonstrates the need for Enterprise (or something similar) to aid the programmer in the development of distributed software. Regardless of the underlying communication protocol/system, the programmer should concentrate on the problem rather than on the implementation details. That is not to say the implementation details are not important, but time spent debugging something that is demonstratably capable, generated automatically by the pre-compiler, is time wasted.

5.4 Summary

The mediocre performance of Enterprise required the identification of the bottleneck. Several experiments were done attempting to identify the bottleneck. The performance of simple message passing between NMP and ISIS processes is investigated.

The cost of passing messages between two processes using ISIS is about double that of NMP. Introducing a simple forwarder increases the difference to about four and five times between ISIS and NMP. NMP costs between individuals and the simple forwarder were about double – as expected. The ISIS versions of message passing between individuals and the simple forwarder ranged between two and four times. This increase is not related to an increase in message size. The cost is somewhere in the ISIS subsystem software.

However, introducing the intelligent asset manager increases the message cost to about ten times the cost of the simple forwarder. The problem lies not with the asset manager as first thought, but with the failure of ISIS to properly deallocate consumed messages. The affected processes page-faulted themselves to drastic slowdowns trying to manage the extra and unneeded messages.

The identification of a quantum leap in turnaround times between messages is surprising. The expected response was a steady degradation of performance with an increasing number of messages. ISIS does degrade in quantum leaps and the threshold number of messages where this occurs is not proportional to the message size.

Clearly, the ISIS communication subsystem must be discarded for a more suitable communication model. These known problems with ISIS have persisted for a considerable length of time with little prospect of resolution in the near future.

The asset manager does have a cost that is reflected in the speedup differences between the ISIS and NMP versions of the chaotic Gauss-Seidel algorithm. The differences in speedup were 5.4 for ISIS versus 9.2 for NMP. The NMP version was not easy to implement and debug. Whether or not to use Enterprise must be balanced against the ease of implementation and the relatively poor performance of the resultant code.

Chapter 6

Evaluation and Recommendations

The tested version of Enterprise has been evaluated by implementing four parallel algorithms. The conversion from sequential to distributed processing went well in theory, but not in practice. Problems with the pre-compiler output having to be modified by the user prior to conventional compiling interfered with the production of the parallel programs. The quality of the communication subsystem, ISIS, degraded the performance of the tested parallel programs.

This version of Enterprise has several problems associated with it. However, given a choice between the communication subsystem ISIS along with the automatic insertion of the necessary code for distributed processing and using the NMP package, Enterprise is the obvious choice. Though Enterprise is not perfect, the problems encountered while producing the programs were trivial compared to the frustration generated by the NMP programming.

The performance of these parallel programs is an important evaluation criterium. The performance of the algorithms running under ISIS is poor, even when the errors introduced by ISIS were avoided by reducing either message size or frequency. Message passing between processes suffers from the failure of message deallocation. ISIS is aware of this problem and intends to resolve it in the next release. The next release of ISIS is already several months behind schedule. This and other more minor problems with this software product have still not been resolved.

6.1 Recommendations

There are a number of recommendations for the next version of Enterprise. First, here are the major recommendations. These are important from either the performance perspective or the user interface. They should have first priority for implementation.

- Replace ISIS with either a custom communication library or some other commercial product. This decision should be made before any more work is done upgrading the model or executive implementations. It was necessary to have some communication system to prototype Enterprise. Now that the requirements of

Enterprise have been identified, any communication package (eg. NMP, sockets) could and should be used.

- All future testing should be done using the graphical user interface (GUI). This will create some problems until the pre-compiler is upgraded so that, at the very least, the four algorithms implemented in this thesis can be compiled without any further modification by the user.
- The Enterprise executive should be freed from the constraint of using one communication package. This can be done by changing the code inserted by the pre-compiler so that communication implementation independent code is inserted. This abstraction is necessary since Enterprise is supposed to be implementable on any communication system. This applies to both the modified user code and the Enterprise executive processes.
- Provide a mechanism whereby the user can specify which processor the executive tools are to be run on. Currently, this is not done. The executive processes are run without any user control. For example, the matrix multiplication algorithm (Chapter 4.2) would benefit from having the asset manager located on a machine with lots of main memory.

The following are small details that, while not strictly necessary for the correct operation of Enterprise, would make the programmer's life easier.

- Modify the graph file to allow dependencies of include files and libraries to be automatically dealt with when compiling to the distributed level and later compiling to the machine level. Currently, this is the user's responsibility.
- Change the names of the pre-compiler tools. **ccl, gcc** are used for something completely different than what the pre-compiler intends.
- Change the search path from having to search the local directory first. This is not acceptable from a security point of view.
- Having a directory where all the Enterprise binaries are stored is highly desirable. Then, the users only need to add that directory to their path.
- Allow the user to insert compiler directives that are machine independent. Optimizing code and debugging are two obvious examples. The keywords are implemented, recognized and ignored by the latest version of the pre-compiler.
- When running a distributed program, have an orderly shutdown of all assets in the graph (not just the first asset) before shutting down any service assets. This problem was first seen in the transitive closure algorithm.

6.2 Future Work and Research Areas

Here are several areas that have research potential or are larger bodies of work. These are intended to enhance Enterprise.

- An obvious omission of the simple algorithms tested is a parallel sort. One such distributed sorting algorithm is parallel sorting using regular sampling (PSRS) [18] which uses peer-to-peer communication and as such is not easily implementable using the current version of Enterprise. One way would be to implement all sorting tasks as *services* with one master process calling these *services*. This is not an effective way to utilize Enterprise since there must be one process acting as the master while only the slave (*service*) processes do the sorting. The code is not easily modified when increasing or decreasing the number of processors, since each *service* asset must have its user-supplied source file modified to reflect change in the communication connections. The implementation and model should be modified to permit peer-to-peer communication. This new asset type is not intended to be recursive but rather it permits a round-table or board-meeting type of communication.

- Test the Enterprise model using a heterogeneous mix of processors. This, while important in an university environment, is not important to the industrial sponsors or users of this work.

- Develop and test larger programs other that these four toy programs using Enterprise. Larger programs will have different dynamics during the course of the run. This will stress Enterprise in new ways and demonstrate practicality for real-world problems.

- The current implementation of Enterprise is a centralized control (the machine manager) with some distributed parts (asset manager). This implementation puts a heavy demand on the machine manager processor resources while lightly loading the other processors. This is neither fair nor reliable. The implementation of a more distributed control should be examined. The new daemon server of Enterprise addresses this problem.

- An important question in the design of Enterprise was whether or not to dynamically launch the assets as needed or to statically launch all processes before starting up the Enterprise program. The dynamic launching argument stresses the sequential flow of the user's program. The beginning assets could be finished before later assets are needed. This results in better processor usage and may reduce the number of necessary processors. The main drawback is how to implement this. A call to a non-existent process will not trigger a program error but, rather, signals the Enterprise executive to launch the missing process. However, when a process exits, one constraint is that there can be no internal state of that process required for the next use of the process.

Statically launched processes require the maximum number of processors available at run time (excluding the variable replication). If there are not enough processors, multiple processes are placed on a single processor. This is not generally conducive to good speedups. Static launching is easier to implement in the executive. The current version of the executive statically launches the assets for simplicity.

- The process group concept found in ISIS should be implemented in the new communication subsystem used by Enterprise since it decouples processor location from the logical identification of the asset both from the single or replicated point-of-view.

- Create some mechanism that respects the ownership of processors. The purpose of having a low-cost supercomputer based on networked workstations is to adsorb the idle CPU cycles. If a general user or the owner of a processor wishes to use that processor, Enterprise should be able to either safely checkpoint the offending Enterprise process and migrate it to another idle processor or wait until the processor is idle and then restart computations. All of this should be transparent to the user.

6.3 Appraisal of Enterprise

Enterprise is intended to free the user from creating and maintaining the code necessary to set up and ensure communication between different processors. The two phases – the model and implementation – are given separate appraisals.

Enterprise, the model, is in good shape. Many coarse-grained distributed processes are amenable to easy implementation using Enterprise. The one exception is peer-to-peer communication. Enterprise is built with hierarchal communication in mind. Peer-to-peer is possible by using a blackboard mechanism (*services*) but it is awkward and potentially costly.

Enterprise, the implementation, is in relatively poor shape. This is based on two metrics: ease of creation and performance. Ease of creation of distributed programs is a subjective metric while the performance of the algorithm is an objective metric for assessing the effectiveness of this system.

The pre-compiler (used for insertion of the distributed code) is not up to the necessary standard of avoiding routine user intervention. However, when the pre-compiler does do its work, conversion from sequential to distributed processing takes virtually no time at all compared to the coding necessary without Enterprise. If the user must intervene with the pre-compiler output, usually a few minutes are all that are needed to insert the few necessary code modifications.

The performance of the distributed program is not very good using Enterprise. It is possible to get a 5.4 speedup using 10 processors for the Gauss-Seidel algorithm with essentially minimal effort. For some considerable effort, a speedup of 9.2 was obtained using handcrafted coding. The cost of handcrafting must be weighed against the benefits of usir , programs like Enterprise. If the Enterprise program compiles correctly, no further effort other than developing the serial code is required.

The speedup achieved is only part of the benefit of using distributed processing. Much larger problems are now solvable by taking advantage of distributed local and virtual memory, without moving to a different (more costly) machine. Having a group of low-cost machines is more likely to occur rather than a single larger machine.

The run-time executive of Enterprise has problems. It fails to return control of the program to the user in a consistent manner for three of the four tested algorithms. The fourth algorithm has control returned too soon resulting in the program failing to compute the correct answer.

The blending of the three models, Enterprise, communication, and user's is still not complete. The tested version of Enterprise is not viable for general public use but the promise of Enterprise is seen in its potential capabilities. Despite the serious implementation flaws, it is possible to appreciate the advantages of Enterprise to create distributed parallel programs with minimal effort and to achieve speedup. Solving these implementation flaws will make Enterprise a desirable product both for research in and production of parallel algorithms on a distributed network of workstations.

Bibliography

[1] Selim G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, NJ., 1989.

[2] Howard Anton and Chris Rorres. *Elementary Linear Algebra with Applications.* John Wiley and Sons, Toronto, 1987.

[3] Özalp Babaoğlu, Lorenzo Alvisi, Alessandro Amoroso, and Renzo Davoli. Paralex: An environment for parallel programming in distributed systems. Technical Report UP-LCS-91-01, University of Bologna, Italy, Feb. 1991.

[4] G.M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors.* PhD thesis, Department of Computing Sciences, Carnegie-Mellon University, 1978.

[5] Adam Beguelin, Jack J. Dongarra, G.A. Geist, Robert Manchek, and V.S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing 91*, pages 435–444, 1991.

[6] T. Breitkreutz, S. Sutphen, and T.A. Marsland. Developing NMP Applications. Technical Report TR 89-11, University of Alberta, March 1989.

[7] Enoch Chan. The Enterprise Code Librarian. Master's thesis, Department of Computing Sciences, University of Alberta, 1992.

[8] Enoch Chan, Paul Lu, Jimmy Mohsin, Jonathan Schaeffer, Carol Smith, Duane Szafron, and Pok Sze Wong. ENTERPRISE: An Interactive Graphical Programming Environment for Distributed Software Development. Technical Report TR 91-17, University of Alberta, September 1991.

[9] J. Fishburn. *Analysis of speedup in distributed algorithms.* PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, 1981.

[10] Curtis F. Gerald. *Applied Numerical Analysis.* Addison-Wesley, Don Mills, Ontario, 1970.

[11] Gene H. Golub and Charles F. Van Loan. *Matrix Computations: Second Edition.* John Hopkins University Press, Baltimore, Maryland, 1989.

[12] Isis Distributed Systems, Inc., Ithica, N.Y. *The Isis Dstributed Toolkit: Version 3.0, User Reference Manual*, 1992.

[13] A. Jones and P. Schwartz. Experience using multiprocessor system - a status report. *Computing Surveys*, 12(3):121–166, June 1980.

[14] Erwin Kreyszig. *Advanced Engineering Mathematic: Third Edition.* John Wiley and Sons, Toronto, 1972.

[15] Thomas J. LeBlanc and Evangelos P. Markatos. Shared Memory vs. Message Passing in Shared-Memory Multiprocessors. In *Proc. 4th IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, December 1992. (to appear).

[16] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Don Mill, Ontario, 1990.

[17] Ted Lehr, Zary Segall, Dalibor F. Vrsalovic, Eddie Caplan, Alan L. Chung, and Charoes E. Fineman. Visualization performance debugging. *IEEE Computer*, 22(10):38–51, 1989.

[18] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, and Hanmao Shi. On the Versatility of Parallel Sorting by Regular Sampling. Technical Report TR 91-06, University of Alberta, March 1992.

[19] Greg Lobe. The Enterprise Interface. Master's thesis, Department of Computing Sciences, University of Alberta, In progress for 1993.

[20] T.A. Marsland and M.S. Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14:533–551, 1982.

[21] T.A. Marsland, Alexander Reinefeld, and Jonathan Schaeffer. Low overhead alternatives to SSS*. *Artificial Intelligence*, 31(1):185–199, 1987.

[22] Nils J. Nilsson. *Problem-solving methods in Artificial Intelligence.* McGraw-Hill, Toronto, 1971.

[23] Cherri M. Pancake and Sue Utter. Models for visualization in parallel debuggers. In *Proceedings Supercomputing '89*, pages 627–636, Nov. 1989.

[24] Terrence W. Pratt. The PISCES 2 Parallel Environment. In *Proceedings of the International Conference on Parallel Computing*, pages 439–445, 1987.

[25] Zary Segall and Larry Rudolph. PIE: A Programming and Instrumentaion Environment for Parallel Processing. *IEEE Software*, pages 22–37, Nov. 1985.

[26] Ajit Singh. *A Template-Based Model for Parallel Computing.* PhD thesis, Department of Computing Sciences, University of Alberta, 1992.

[27] P.B. Soares. On remote procedure call. In *Proceedings of the 1992 CAS Conference*, pages 215–267, Toronto, Canada, Nov. 9-12 1992.

[28] Pok Sze Wong. The Enterprise Executive. Master's thesis, Department of Computing Sciences, University of Alberta, 1992.

Appendix A

Experimental Data

Size	User Time (seconds)				
	Dense	Tridiagonal 10	Tridiagonal 25	Banded 10	Dense (no SOR)
50	0.46	0.47	0.50	1.62	0.10
100	1.77	3.58	1.99	12.53	0.35
200	7.14	25.40	18.68	99.41	1.17
300	15.81	66.63	85.25	332.57	2.48
400	28.48	128.14	236.70	792.06	4.33
500	44.87	210.15	498.03	1,548.60	9.89
600	65.03	310.62	883.96	2,675.39	14.04
1,000					29.62
1,200					43.03
1,500					66.99

Table A.10: User times for serial Gauss-Seidel using successive over-relaxation (SOR).

Replication factor	Matrix size							
	500		1,000		1,500		2,000	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
2	5.62	1.8	28.71	1.0	186.9	0.4		
3					98.12	0.7		
4	3.63	2.7	10.97	2.7			164.1	0.8
5	3.06	3.2	9.71	3.0	19.85	3.4		
8			6.78	4.4			28.56	4.5
10			6.18	4.8	12.41	5.4	25.24	5.1

Table A.11: Elapsed times in seconds and speedup for distributed Gauss-Seidel using a line.

Matrix	Serial	Replication 4		Replication 8		Replication 10	
Size	Times	Elapsed	Speedup	Elapsed	Speedup	Elapsed	Speedup
100	1.80	3.53	0.5	3.45	0.5	4.09	0.4
200	15.10	9.99	1.5	10.37	1.5	10.78	1.4
300	50.81	28.19	1.8	28.37	1.8	31.43	1.6
400	120.94	62.86	1.9	64.82	1.9	72.56	1.7
500	237.29	123.23	1.9	125.60	1.9	142.33	1.7
600	413.54	211.29	2.0	217.72	1.9	258.31	1.6

Table A.12: Serial user times and distributed elapsed times and speedup for block multiplication using a line.

Matrix Size	Serial User time	Elapsed time	Speedup
100	1.80	3.76	0.5
200	15.10	18.00	0.8
300	50.81	60.59	0.8
400	120.94	175.51	0.7

Table A.13: Serial user times and distributed elapsed times and speedup for block multiplication using a department. Replication factor is set to 8.

Problem	Serial	Replication Factor						
Size	Time	4	5	6	7	8	9	10
400	44.13	31.33	36.31			35.71		54.49
500	86.16		52.75					77.47
600	149.28			67.17		61.84		87.44
700	236.76				93.77			106.86
800	354.30					122.11		133.03
900	505.09						174.46	175.98
1,000	693.22							225.84

Table A.14: Serial user times and distributed elapsed times for transitive closure using a department.

Problem	Replication Factor						
Size	4	5	6	7	8	9	10
400	1.4	1.2			1.2		0.8
500		1.6					1.1
600			2.2		2.4		1.7
700				2.5			2.2
800					2.9		2.7
900						2.9	2.9
1,000							3.1

Table A.15: Speedup for transitive closure using a department.

	Best weighting: 100 0 0 0 0 0 0 0 0	
Level of Parallelism	Elapsed time (seconds)	Speedup
Serial‡	393.0	
12	78.64	4.7
10	77.07	4.8
8	76.39	4.9
6	75.36	4.9
4	76.[illegible]5	4.8
2	[illegible]	3.8

	[illegible]ghting: 12 11 11 11 11 11 11 11 11	
Level of Parallelism	[illegible] (seconds)	Speedup
[illegible]rial‡	600.5	
9	134.3	4.5
7	134.7	4.5
5	132.9	4.5
3	132.7	4.5
1	144.8	4.1

	Worst weighting: 0 0 0 0 0 0 0 0 100	
Level of Parallelism	Elapsed time (seconds)	Speedup
Serial‡	503.2	
8	114.1	4.4
6	111.0	[illegible]
4	112.7	4.5
2	115.0	4.4
0	309.9	1.6

‡ User time (seconds)

Table A.16: Effect of different weightings on performance of alpha-beta search. Replication factor is set to 9.

Granularity	Depth 10 Elapsed Time (seconds)	Depth 10 Speedup	Depth 11 Elapsed (seconds)	Depth 11 Speedup
Serial‡	227.1		872.9	
4	59.32	3.8	222.7	3.9
5	59.83	3.8	222.8	3.9
6	59.42	3.8	222.4	3.9
7	59.41	3.8	223.2	3.9
8	62.07	3.7	224.9	3.9
9	62.13	3.6	233.7	3.7

Weighting: 40 10 10 10 10 5 5 5 5
‡ User time (seconds)

Table A.17: Distributed elapsed times, and speedup for alpha-beta search on a tree of width 9 using replication factor of 9.

Granularity	Serial times	6 processors Elapsed	6 processors Speedup	9 processors Elapsed	9 processors Speedup
1	4,710	1,367	3.4	1,363	3.5
2	4,706	1,339	3.5	1,346	3.5
3	4,709	1,316	3.[illegible]	1,315	3.6
4	4,701	1,294	3.6	1,296	3.6
5	4,691	1,287	3.6	1,290	3.6
6	4,418	1,365	3.2	1,367	3.2
7	1,364	1,361	1.0	1,364	1.0

Weighting: 60 15 10 5 10(1) 26(0)

Table A.18: Serial user times, distributed elapsed times and speedup for alpha-beta search using replication factor of 6 and 9 with a tree of width 40 and depth 7.

Appendix B

ISIS Installation

B.1 Shell Modifications

The shell modifications are primarily directed to the environment variables. However, the locations of the statements are critical. All these statements must occur before the **.cshrc** file exits if the shell started is not interactive.

For this implementation, there are four variables that must be created while two others can be created but are defaulted to preset values if not created. Table B.19 lists where and what the commands are.

The four that must be created are as follows:

- ISISHOST – the name or address of the machine that is running an ISIS server. Separating multiple servers by colons is mentioned in the ISIS manual but it does not work. Possibly, the run-time executive is not handling this multiple entry properly.

- ISISREMOTE – the port number by which ISIS processes communicate between one another. This could be ignored if ISIS has a **/etc/services** file entry. This only happens if ISIS has been installed by the system manager. With the implementation tested, this installation was not done by the system manager.

```
setenv ISISHOST so-heart-rv.cs.ualberta.ca
setenv ISISREMOTE 1613
setenv ISISPORT 1612
setenv ISIS_INCLUDE /usr/maligne-lk/Enterprise/isis2.2/SUN4
#setenv mach_file /usr/maligne-lk/grad/ian/.machine-file
#setenv ent_dir /usr/maligne-lk/grad/ian/SRC
set path = (. $path)
#skip remaining setup if not an interactive shell
if ($?USER == 0 || $?prompt == 0) exit
```

Table B.19: The shell startup (.cshrc) modifications for Enterprise.

- ISISPORT - the port number by which ISIS processes listen for messages. Again this can be ignored if ISIS is installed by the system manager.
- ISIS_INCLUDE - the location of the ISIS include files. In the next version of Enterprise this has been replaced with the ISISPATH and ARCH environment variables which are intended for heterogeneity and binary locations.

The two optional environment variables are as follows:

- mach_file - the file containing the list of available machines. It defaults to mach_file in the current user directory.
- ent_dir - the directory where the Enterprise program is found. It defaults to the current working directory of the user.

The other major modification to the shell is the **PATH** environment variable. The PATH environment variable must be set to look in the current working directory first. Both the run-time executive and the pre-compiler have to find files in the local directory. The simple solution is co use symbolic links for the files. This does not consume much space; it does clutter the directory.

This modification is only intended as a temporary measure until Enterprise becomes more robust and stable. When this happens, the various programs that compose Enterprise can be located in a separate directory and the path added to the user's list. Having to search first for executables in the current directory is neither a secure nor desirable feature.

B.2 The ISIS Server

Enterprise is currently implemented using the ISIS communication package. The ISIS server [12] provides all the necessary services for communication between distributed processes. Other communication subsystems could be used as long as they provide the minimal subset of functionality required by Enterprise.

The notion of process groups is an attractive feature of ISIS. A process group is an unique logical address where processes rendezvous to deposit or collect messages. Enterprise uses this to control replication and interprocess communications. Debugging is as simple as connecting to the monitoring process group. Starting and stopping of the various processes is also handled via process groups.

Each ISIS server processor requires six process slots for the various processes that make up ISIS system. The binary files for ISIS are listed in Table B.20 under the heading ISIS. ISIS requires three communication ports globally free over the entire network. The selection of the port numbers is done in the **sites** file which lists all sites for possible ISIS servers. It is possible to use different port numbers but the servers using this different set of port numbers must not know about any other set of ports. This results in multiple **site** files and complicates the maintenance of the ISIS subsystem.

Compiling	Running	ISIS
compile	run	isis
gcc	asset_mgr	protos
cc1	machine_mgr	xmgr
cpp	monitor	rexec
cppp		rmgr
		news

Table B.20: List of binary files used by Enterprise.

Port connections are made by the values contained in the environment variables ISISREMOTE and ISISPORT to the local ISIS server defined by ISISHOST as defined earlier. If Enterprise is a system installation then ISIS ports would be identified in the **/usr/etc/services** file and the two port environment variables would not be needed.

ISIS permits interprocess communication through unique logical addresses that are global in scope. A process asks its local server for a particular address; the local server will forward the message to the appropriate remote server for delivery. If there are many servers, then the cost of maintaining this global address list has an effect on network performance.

There is a current limitation on Enterprise caused by the process groups. There can only be one Enterprise process running on any ISIS network. The **run** process which connects to all Enterprise processes is a unique process group name. Any other Enterprise program that starts up will also use that process group. The results are non-deterministic, sometimes amusing, but inevitably fatal to both programs. Creating process group addresses that are unique to the processor that starts the Enterprise program is trivial (launching host name and process identifier) and is in the version 0.6 of Enterprise.