University of Alberta

Library Release Form

Name of Author: David R. Woloschuk

Title of Thesis: Analysis and Display of Parallel Program Performance Information within Enterprise

Degree: Masters of Science

Year this Degree Granted: 1998

. . . .

David R. Woloschuk
9312-75 St
Edmonton, Alberta
Canada, T6C 4H4

Date: . . . .

University of Alberta

Analysis and Display of Parallel Program Performance Information within Enterprise

by

David R. Woloschuk

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

Department of Computing Science

Edmonton, Alberta
Spring 1998

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 **Analysis and Display of Parallel Program Performance Information within Enterprise** submitted by David R. Woloschuk in partial fulfillment of the requirements for the degree of **Masters of Science**.

. . . .
Jonathan Schaeffer

. . . .
Duane Szafron

. . . .
John Buchanan

. . . .
Edo Nyland

Date: . . . .

To my parents.

Abstract

Enterprise is a parallel program development environment which is used to facilitate the creation of new parallel applications and to assist a user to quickly convert existing sequential programs into a parallel counterpart. Enterprise contains tools which allow the user to design, code, compile, execute, and debug a parallel program. However, one capability Enterprise lacks is the ability to tell the user how well his or her program performs.

Quantifying how well a program performs is a very useful feature to a parallel programmer. Execution time of a program, the most obvious choice of a judgement criteria, is not always an adequate measure of the "goodness" or "badness" of a program. What is required is a means of quantifying other aspects of a program such as resource utilization. This can be accomplished through the use of performance analysis tools.

The goal of the work contained in this document is to demonstrate that the addition of real-time performance analysis tools tailored to the Enterprise system can provide valuable and useful insights into a program. This includes a discussion of the issues associated with performance analysis and a description of the analysis tools added to Enterprise, the insights they provide and the problems faced with integrating the tools into an existing environment.

Acknowledgements

I'd like to take this opportunity thank my supervisors, Dr. Jonathan Schaeffer and Dr. Duane Szafron for their support, encouragement and their patience. I'd also like to thank the members of the Distributed and Parallel Computing laboratory for their help and support (special kudos to Diego Novillo, Ian Parsons and Steve MacDonald for their help in the preparation of this document).

A special "thank you" goes out to my family and friends for the support they provided at the times I needed it the most.

Contents

List of Figures

Chapter 1

Introduction

It is an inescapable fact of life that no matter how many cultural or technical boundaries we cross, or how enlightened as a species we become, we will always find in our midst the archetypical "snake-oil salesman." The best of the confidence artists elevate the act of fooling the naive public to an art form and are always ready, willing and able to sell us our own follies (at a discount price, of course). However, the ability to cloud the minds of an unsuspecting audience is not exclusive to street-smart vendors, television info-mercial hosts or career politicians; the field of computing science has its share of hustlers as well.

Consider for a moment the goal of parallel computing: a computationally expensive problem is distributed over several processors with the hope that a parallel solution will be evaluated in less time than a sequential solution would. Based on this premise, it is a simple matter to determine whether or not a given parallel implementation of a program is "better" than its sequential counterpart: compare the execution times of both programs and discover which version is the first to run to completion. However, the more interesting question of whether one parallel implementation is better than another is far more difficult to answer. As an extreme example, consider two parallel implementations of a program called "Groo." One version of the program written by Joe Programmer makes use of 10 processors. Another version, written by Fred Programmer, makes use of 2 processors. Joe runs his program and finds that it runs 3 times as fast as the sequential version. On the other hand, Fred runs his program and finds that it runs only 1.5 times faster than the sequential version. Joe could claim his program is the better of the two because it runs to completion faster than Fred's. Fred could counter this statement by noting that his program gets nearly the same speed up factor as Joe's but uses 8 fewer processors. Fred's rationale is that because his program makes better use of the available resources it is limited *only* by the fact it runs on 2 processors. Fred and Joe could both offer convincing arguments and, with judicious hand-waving, each could convince an observer that his was the better program. The heart of the paradox is that both Joe and Fred, in performing the comparison of their programs, are basing their statements on different criteria. Fred bases his judgement on resource usage whereas Joe bases his on program speed. The judging process lacking in convincing evidence thus degrades into a magician's act of smoke and mirrors where the audience sees only what the illusionist (or parallel programmer) wants them to see.

Rash judgements about a program's "goodness" or "badness" are symptomatic of a short-sighted approach to program analysis where a more thoughtful approach is required. However, in the quest for ultimate program performance, several important judging criteria are often ignored:

- How long did it take to develop the program?
- How easy is the program to maintain?
- How easy is the program to use?
- Is the program scalable?
- Is the program robust?

- Is the program hardware dependent?

It must be noted that users are sometimes willing to make small sacrifices in performance for a system that is easy to use and maintain. These users would gladly choose Joe's program over Fred's if it meant they spent less time maintaining the code. That being the case, it becomes an even more daunting task to evaluate a program. Fortunately, the task is only difficult, not impossible.

Expanding the scope of analysis to include more than just speed[1], yet limiting the examination to quantifiable metrics, makes the task of analyzing a program easier. Speed may provide a reasonable first estimate of a program's performance, but as a lone judging criteria it falls short. Resource utilization may also provide a valid means of gauging a program, but used in isolation it also fails to provide a complete picture. The solution is to consider all *relevant* criteria in a consistent, structured manner in order to make an informed opinion. However, knowing the solution and enacting the solution are two entirely different issues. It is not an easy task to quantify and measure a parallel program's performance signature.

As tools and techniques for writing a parallel program evolve, the task of generating a working parallel program is simplified. Unfortunately, generating a *good* parallel program still requires finesse (and perhaps a *little* black magic). The example of Joe and Fred, simplistic as it may seem, is a plausible scenario. Neither speed nor resource utilization *alone* is a sufficient metric to gauge a program's "goodness" or "badness". Anyone claiming that one implementation of a program is better than another solely because it runs faster is as guilty of misleading the public as the proverbial snake-oil salesman. What is the answer? How can we determine whether or not one parallel program is better than another?

1.1 Thesis Motivation

The Enterprise Parallel Programming system is an easy to use, versatile and interactive environment that helps a programmer build a parallel application to run on a network of heterogeneous Unix workstations. Not surprisingly, Enterprise "delivers the goods." Enterprise does provide an impressive graphical interface which is easy to use yet performs a variety complex operations. Enterprise does allow a programmer to quickly develop a working parallel application by providing an intuitive environment in which a user writes a parallel application. Enterprise does allow an application to run on a network of heterogeneous workstations. However, if Enterprise were perfect or complete, this would be a very brief document.

Despite its promise, there is still room for improvement in Enterprise. The Enterprise programming model has known deficiencies such as the absence of true shared memory and the lack of parallel input and output facilities. Fortunately, Enterprise is like any large software application and is continually being improved. Each new release of Enterprise sees corrections that either minimize or eliminate the known deficiencies. However, there was always one area of Enterprise that remained untouched: performance analysis. Until now, there was no mechanism in place to help users determine how well their programs perform. Users were left to fend for themselves.

Creating an Enterprise application (program) to solve a problem is like any other problem solving strategy. As there are both good solutions and bad solutions to a problem, there are both good Enterprise programs and bad Enterprise programs. However, "good" and "bad" are relative, and subjective, terms that imply ordering without measure. A programmer can write different versions of a parallel program using Enterprise and perform a series of tests to determine which program version runs faster. However, to determine where a given program is lacking is a far more difficult proposition. The work in this document shows that tools for informed, real-time performance monitoring and analysis can be seamlessly integrated into the Enterprise environment. Such tools are invaluable assets used to evaluate and fine-tune a parallel programs in Enterprise.

[1] For the purposes of this document, the terms *program speed* and *execution time* refer to the amount of time required for a program to run to completion.

1.2 Scope and Organization of this Thesis

This thesis is divided into six chapters. The remainder of the first chapter introduces issues pertaining to parallel program performance analysis. The second chapter introduces the Enterprise parallel programming system, its programming model, its metaprogramming model and discusses the performance issues specific to Enterprise. The third chapter provides the first look at the Enterprise visualization tools and discusses how an Enterprise programmer would use the tools to analyze and improve the performance of a parallel application. The fourth chapter provides further insight into the form and function of the performance visualization tools. The fifth chapter provides implementation information about the performance analysis mechanisms in Enterprise. The sixth chapter offers summaries and conclusions of the work contained in this document.

1.3 Performance Analysis Issues

For the purposes of this document, *performance analysis* is defined as the process of examining a program's run-time behavior in order to characterize resource usage or algorithm effectiveness. The behaviors being scrutinized are the result of an accumulation of *events*, run-time occurrences that encapsulate key aspects of the program's execution. Events of note may include memory address accesses, register shifts, message transmissions or even the number of times a specific counter is reset to zero. Examining events in the order which they were generated will reproduce exactly the program execution from where the events were drawn.

There are two methods by which data (events) may be processed in order to obtain performance information. The first, *real-time analysis*, involves gathering and analyzing data at a rate comparable to the speed at which the data is generated. Real-time systems are attractive because they provide an instantaneous view of a program's execution, a desirable feature if the execution time is measured in hours, days or even longer periods of time. If a flawed program takes 2 months to run to completion, and the programmer is forced to wait the full 2 months before the problem can be found, then that was 2 months of computing resources wasted. However, if the programmer could discover the flaw during the course of the execution, he or she could halt the program, correct the problem and restart the computations and thus waste fewer computing cycles. Discovering that there is something amiss with a program as soon as possible is always in the best interest of the user.

Real-time approaches also allow for the possibility of user interaction during the program's execution. If such a facility is available, the user can modify execution parameters "on the fly", witness the results of the changes, and react accordingly. The user can optimize execution parameters as the program is running.

The alternative to real-time analysis is to use a *post-mortem* strategy. Like real-time systems, post-mortem approaches also require data to be gathered during the program's execution. Unlike real-time analysis, event processing is delayed until after the program has run to completion. Post-mortem systems take the data obtained during a sample run of a program from a *trace file* in which raw events were recorded as the program was executing. The raw events can then be post-processed at any time to reproduce the run-time behavior of the program. A second important difference between real-time and post-mortem systems is that a user has greater control over how the passage of time is handled during a post-mortem analysis. Because a trace file is used, full knowledge of the events, the times they were generated, and their effects is available. Since there are no longer any random factors present, the responsibility of processing events can rest solely with the analysis system. Events may then be processed as quickly or as slowly as the post-mortem analysis system permits.

Attractive as real-time strategies appear, most parallel program analysis systems adopt the post-mortem approach because of the inherent drawbacks of real-time systems. First, real-time analysis strategies are limited by their immediacy. Trends that emerge over time cannot readily be identified during real-time examinations. Such systems have incomplete knowledge of the program's execution and any attempt to correlate disparate events require backwards analysis occurring simultaneously with new event analysis, a computationally expensive procedure. The second problem with real-time strategies is that the time differential between events can be so small that they progress faster than the human eye can follow. If an event of particular interest happens during a period of high activity, it may be completely overlooked by the user.

Although the obvious problems with real-time strategies are troublesome, they are by no means insurmountable. However, there is a more serious problem. The greatest difficulty that must be overcome by real-time systems is not due to the analysis software, but to the limitations of computer hardware. Multiple processors working in parallel can generate thousands of events per second. For true analysis to occur, each of these events require individual scrutiny by the analysis system. Periods of high activity could flood the analysis mechanism with events which must then be placed in a queue to wait for their turn to be examined. Every moment spent in the queue increases the lag between when the event was generated and when the analysis occurs, thus distancing the analysis data from the executing program; what the user sees would *not* be what is actually occuring. Limiting the breadth and complexity of analysis lessens the problem, but also limits the amount of information that can be obtained. The hardware problems are lessened with each new generation of faster computers, but it never goes away [2]

Regardless of the strategy used, the analysis process can be broken down into three operations. The first, *acquisition*, is the gathering of data characterizing a program's execution. The second, *analysis*, is the filtering and processing of the gathered raw data into useful information. The third, *presentation*, is to relay the performance information to the user in a clear, concise and unambiguous manner. Despite the apparent separation of the analysis phases, the division of functionality between the operations is for convenience. There is nothing to prevent filtering or analysis from occuring in either the acquisition or presentation phases. Likewise, there is no steadfast ordering which the steps must occur. In fact, there is no strict requirement that all three operations must be performed. At the very least, data must somehow be acquired. The analysis and presentation phases may be wholly independent of acquisition (as would be required by post-mortem strategies). Alternatively, the three operations can be interleaved to create a highly interdependent system (as would be required by real-time processing strategies). Ultimately, the ordering and interaction of the three operations depends both upon the requirements of the user and the capabilities of the system.

1.3.1 Data Acquisition

Data acquisition is the process of gathering raw data used to characterize a program's execution. Acquisition can be accomplished by one of two methods: *simulation* or *capture*. Simulation strategies require the use of a program simulator to generate a data set of theoretical values. This method is not commonly used as the primary means of analysis because simulated data could mask any trends that might naturally occur in true data sets. However, simulation is useful for predicting performance trends by extrapolating performance data from true data sets. For example, a simulation approach could be used to scale up or scale down the number of processors in order to determine a theoretical upper bound to attainable speed-ups. The preferred method of data acquisition is to capture performance information directly obtained from an actual execution run of the program.

There are a variety of means, some more elegant than others, used to gather event data. For example, some computer systems include hardware designed to record low-level events such as cache misses, megaflops (millions of floating point operations per second) and mips (millions of instructions per second). This approach has the virtue that the user's code need not be altered in any way; the computer hardware manages the gathering of data. Although this is a minimally intrusive solution that requires little work on the part of the user, it is not a robust method. The acquisition process has a prerequisite hardware requirement that cannot always be met and is therefore not portable across most architectures. A second and perhaps greater failing of hardware based approaches is that the information gathered may be too far removed from the application to provide the insight required. A user wishing to know the number of times a particular array element is accessed will find little value in knowing the number of times a register shift occurs. Finally, values such as megaflops may not be an adequate measure of how well a program performs. A program performing few floating point operations in favor of jumps or other operations will show fewer megaflops. Therefore, any

[2] Faster computers can not only process more events per second, they can also generate more events per second!

system that uses megaflops as a performance metric may not accurately reflect what is happening as the program runs.

The second approach to gathering data, one that is largely independent of the user's program, is to use a separate *monitor program* that periodically queries the executing program for its status information. This approach requires each of the processor to periodically stop and send a "snapshot" of its state information to the monitor. The pitfall of this method is the heavy intrusion into the program's execution as each processor stops and "takes stock" of itself. This method was used in the first version of TOPSYS (TOols for Parallel SYStems [BB93]) but was abandoned in later versions because of the intrusion it caused.

The third alternative is to insert *instrumentation code* into the user's program itself. This code registers high-level events, conditions or occurrences that are of interest to the programmer. The process could be as simple as inserting print statements into the program or as complex as linking special software libraries into the user's code. PICL (Portable Instrumentation Communication Library [KS93]) is one example of a library based approach. PICL is composed of a set of library routines which automatically instruments the user's code and generates trace files as the program is executing. The trace files can then be consumed for post-mortem analysis at any time by data presentation tools that can manage the PICL file format. The benefit of this approach is that it is robust and flexible. The pitfall to this approach, aside from the additional effort on the part of the user to call the libraries or insert the instrumentation code, is the intrusion upon the execution of the program. This perturbance of a program's execution by foreign code is known as the *probe-effect*, the consequences of inserting a "probe" into a running program. SIEVE (Spreadsheet based Interactive Event Visualization Environment [SG93]) uses perturbation analysis to calculate the disturbance a program experiences due to the instrumentation code and attempts to compensate for the probe effect.

1.3.2 Data Analysis

The goal of the data analysis phase is to take the raw data acquired during acquisition, process the events and distill from it information that is useful to the user. This operation, or group of operations, may be as simple as echoing the data to a file or as complex as extrapolating a predictive curve based on the data at hand. A common application of the analysis phase is to generate statistics based on the execution run: the average number of tasks executed, the average CPU utilization, the maximum speed-up attained, etc. A secondary function of an analyzer is to post-process data into a form usable in the data presentation phase.

Unlike sequential programs where there is a single processor and an easily traceable execution thread, parallel programs run on several processors and create a multi-threaded execution trace that must be resolved. The order of events occurring concurrently often affect the program's outcome and so special care must be made to ensure the order of events, and thus causality, is maintained. Thus, an important function of an analyzer is to reconcile events and ensure that *tachyons*, events where the effect precedes the cause, never occur. The problem of misordered events is exacerbated by situations outside the user's control such as the use of multiple processors each with its own internal methods of measuring time, and communication delays between the processors and the analysis tool.

1.3.3 Data Presentation

Data presentation is the most visible of the three phases and is responsible for relaying performance information to the user in a meaningful and unambiguous manner. The presentation tools are usually visual in nature and often employ clever, colorful displays to convey information to the user. Some tools make use of animation techniques to present information to the user, an ideal means to capture and abstract the dynamic nature of programs. Graphical displays often rely on familiar visual cues and symbolism to exploit a human's innate ability to process visual information. This use of such graphical displays and animations is known as *program visualization.*

ParaGraph [KS93] is perhaps the most fully developed parallel program visualization tool with the widest assortment of graphical displays offered. It is a strictly post-mortem tool which consumes

PICL trace files, performs an analysis of the data and offers several different visualizations of a parallel program execution. A sampling of ParaGraph visualization displays can be found in Figure 1.1.

Despite the power of visualization, there are several limitations to visually oriented presentation systems. First, if too much information is displayed at once, or if the information is displayed too quickly, the users will overlook the details for which they are searching. Second, there are the practical constraints imposed by the hardware used. Graphical operations are computationally expensive and visualization systems are handicapped unless the computer is either very fast or has special provisions to handle graphical operations. The size of the display, in both physical dimensions and display resolution, also places a limit on the amount of information that can be displayed at any given time. The solution to this particular problem, display scaling (selectively displaying events on the computer screen), requires additional "book-keeping" operations and thus incurs additional overhead costs. These physical limitations are particularly aggravating in systems where processors number in the thousands and the ability to scale a display is vital. However, visual cues are not the only means of conveying performance information.

There has been work done in the area of auralization [FJ93], using sound to characterize a parallel program's behavior. Sound is a four dimensional medium consisting of *pitch, tone, tempo* and *duration* which used in various combinations can convey a greater range of information than the conventional two-dimensional computer display. Each of a sound stream's attributes can be manipulated to produce a wide variety of harmonies, textures and melodies which can be mapped to parallel programming events. For example, a busy processor can be mapped to the sound of a flying insect. A group of busy processors becomes a swarm of insects which the user can "follow" by listening to where the swarm is flying and thus determining processor load at various times during execution. The approach is novel and provides an interesting method to analyze programs, but it has several drawbacks. First, not all people have a "musical ear" and would have difficulty in distinguishing useful information from noise. Second, auralization suffers from exotic hardware and software requirements needed to manage audio information. Finally, the mapping of sonic properties to a parallel programming events is a non-trivial task.

Regardless of the means or medium, the goal of the presentation phase is to inform the user of some aspect of the parallel program. Whether it is a visual mapping of events to a graphical display, or an audio mapping of events to sounds, or even an olfactory mapping of events to scents, the user must obtain the desired information.

1.4 Chapter Summary

The ability to evaluate how well a program performs is not needed to write a parallel program, but it does make the task of writing a good program easier. By allowing a user to determine whether or not a resource is being used properly, or whether or not a program has achieved its fullest potential, the user is given the power to write a better program. Though there are a variety of means to do so, gathering, analyzing and presenting performance data are tasks that must be undertaken by a tool that purports to analyze a program.

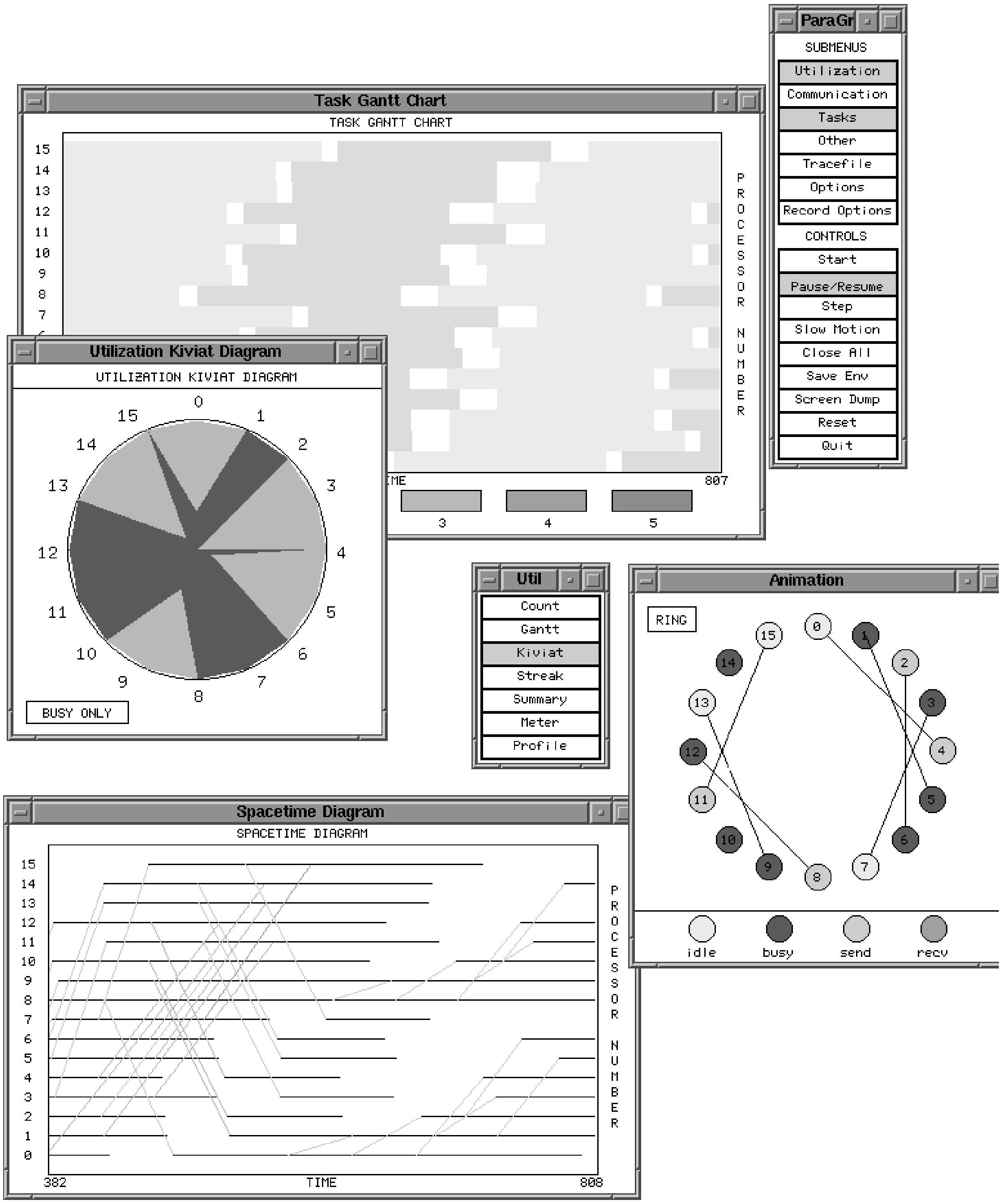

Figure 1.1: Examples of ParaGraph visualization displays

Chapter 2

The Enterprise Programming Environment

2.1 Overview

The Enterprise Parallel Programming System [SSLI93] is one member of a family of similar systems whose purpose is to facilitate the development of parallel programs. Like other members of this family, Enterprise is more than just a collection of tools; it is a fully integrated programming environment which assists a programmer to write, execute, analyze and debug a parallel application. This chapter serves as an introduction to the Enterprise programming system and begins with a description of the subsystems that compose Enterprise. This is followed by a description of the Enterprise programming model and metaprogramming model with the remainder of the chapter left to catalogue program performance issues that arise in Enterprise.

2.2 The Components of Enterprise

Enterprise consists of four interconnected subsystems that perform the specific tasks needed to run a parallel application. The *pre-compiler*, the *run-time executor*, and the *communication manager* are all coordinated by the fourth subsystem, the *graphical user interface.*

2.2.1 The Enterprise Graphical User Interface

The graphical user interface is the means by which the user accesses the facilities found in Enterprise and is the only component of Enterprise with which the user interacts directly. Through the interface, the user launches editors for writing code, opens dialogs used in compiling and executing code, and launches debugger sessions used to debug code. The interface also includes special facilities used to perform post-mortem trace file animation and replay.

The operations which are available to the user are determined by which one of the three *views* of Enterprise is currently active (a view roughly correspond to an operating mode). Through the *design view*, the user defines the *program graph*, a visual representation of the application's parallelism, and enters user source code. The *animation view* is used to animate trace files generated by the run-time executor during a program's execution. The *replay view* not only animates a trace file, but also forces a controlled re-execution of code exactly as it occured during the originating run.

Though the preceding paragraph lists the capabilities of the interface, a more detailed description is not required by this document. Further information can be found in [SSLI93] and [IMM+95] with further insight provided by [Igl94].

2.2.2 The Enterprise Pre-Compiler

The pre-compiler takes both the user source code and the program graph and automatically inserts the Enterprise code needed to manage the synchronization and communication needs of the program. This pre-processing phase requires the pre-compiler to parse the source code and program graph in order to perform a series of checks that ensures that both are mutually consistent. The end product is the creation of intermediate "Enterprise" code which is then compiled by a standard ANSI C compiler to produce the program binaries. The Enterprise pre-compiler is discussed further in [SSLI93].

2.2.3 The Run-Time Executor

The executor is responsible for scheduling, binding, launching and terminating Enterprise processes spawned over a network of workstations. The first Enterprise process launched, the *root process*, fulfills these functions by acting as an executive manager that governs the other Enterprise processes during a program's execution. The root process, in conjunction with the communication manager, is responsible for establishing worker-to-worker communication channels as well as providing a two-way communication link between an executing Enterprise program and the user interface.

2.2.4 The Communication Manager

The communication manager is responsible for the reliable transmission of messages between processors and processes over the network. Unlike the other Enterprise sub-systems, the current communication manager, PVM (Parallel Virtual Machine [Sun90]) was developed outside the University of Alberta and was meant to be used in a wide variety of applications. PVM was incorporated into Enterprise because of its promised functionality and its wide acceptance in the parallel programming community. Earlier versions of Enterprise used other communication managers such as NMP (Network Multi-Processor [MBS91]) and ISIS [Inc92].

2.3 The Enterprise Metaprogramming Model

The Enterprise metaprogramming model is the means whereby a programmer expresses the parallelism of an application graphically in the form of a directed, acyclic *program graph* (an acyclic graph prevents the deadlock problems that can occur in cyclic systems). Vertices of the program graph correspond to Unix processes that perform the user's work with edges between the vertices denoting the channels of communication between these processes. The graph is built and modified via the interface and is an abstraction mechanism. All information about the parallelism of a given application is contained in the graphical representation given by the program graph. Because the desired parallelism is obtained directly from the graph, the user does not need to supply any code to parallelize the program. Instead, it is the pre-compiler's responsibility to supply the parallelization code.

An example of an Enterprise program (both the program graph and the user source code) can be found in Figure 2.1.

The philosophy of Enterprise is that a program developer need not be a parallel programming expert. The use of the program graph furthers this goal by taking the tedious and error prone task of coding the parallelism out of the developer's hands. This is carried one step further by defining parallel programming constructs in terms of a familiar metaphor: a business enterprise. Exploiting the metaphor provides even the most naive *parallel* programmers with an accessible model of a parallel programming system without introducing some of the complexities inherent in parallel programming.

To illustrate the power of the metaphor, consider the structure of a business organization. A business holds assets, resources that are used in its day-to-day operations. These resources include personnel (individuals, receptionists, representatives and managers), sub-organizations within the business (departments, assembly lines, and divisions) and communal services (photo-copiers, pencil

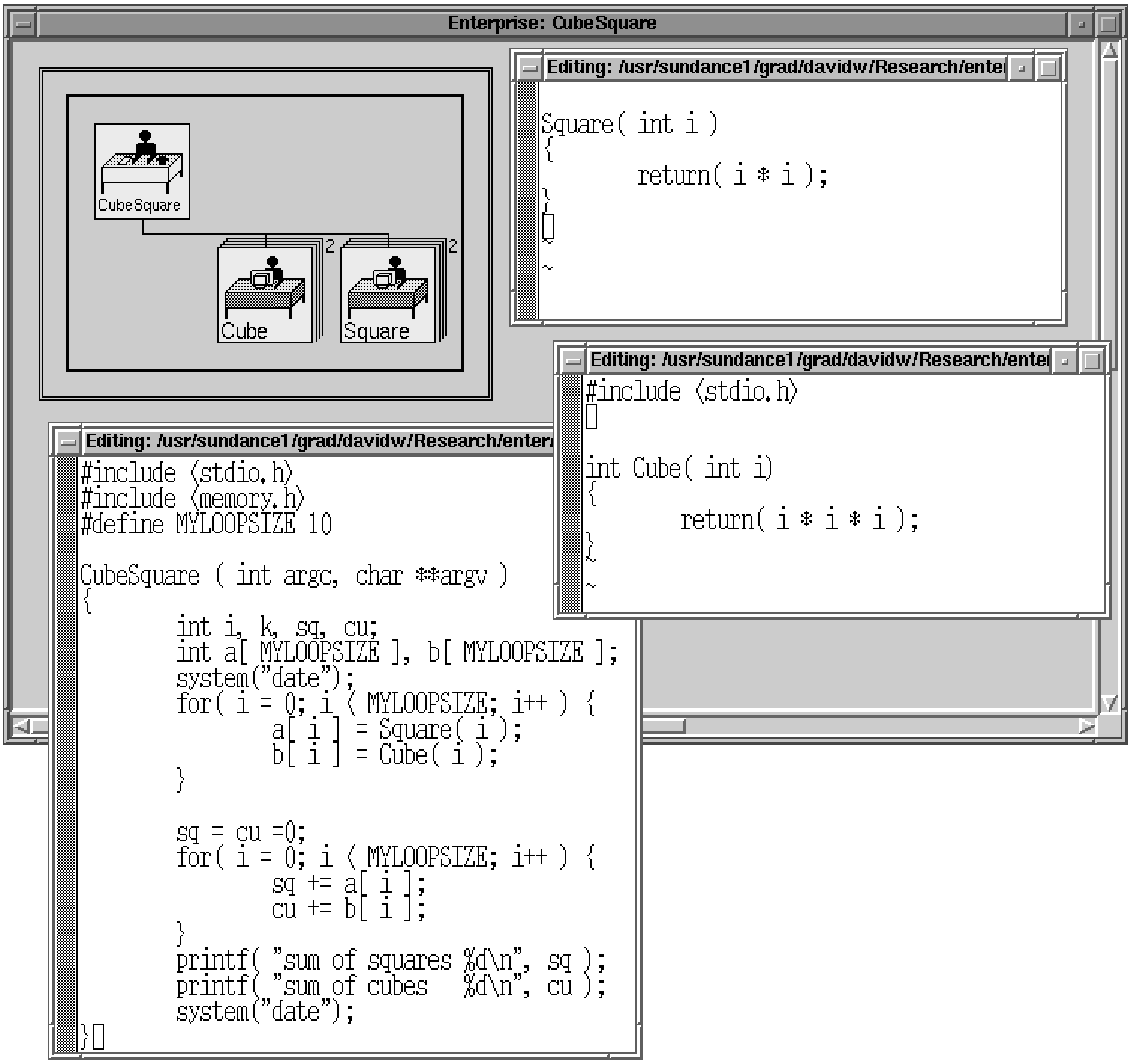

Figure 2.1: The program graph and source code for "CubeSquare".

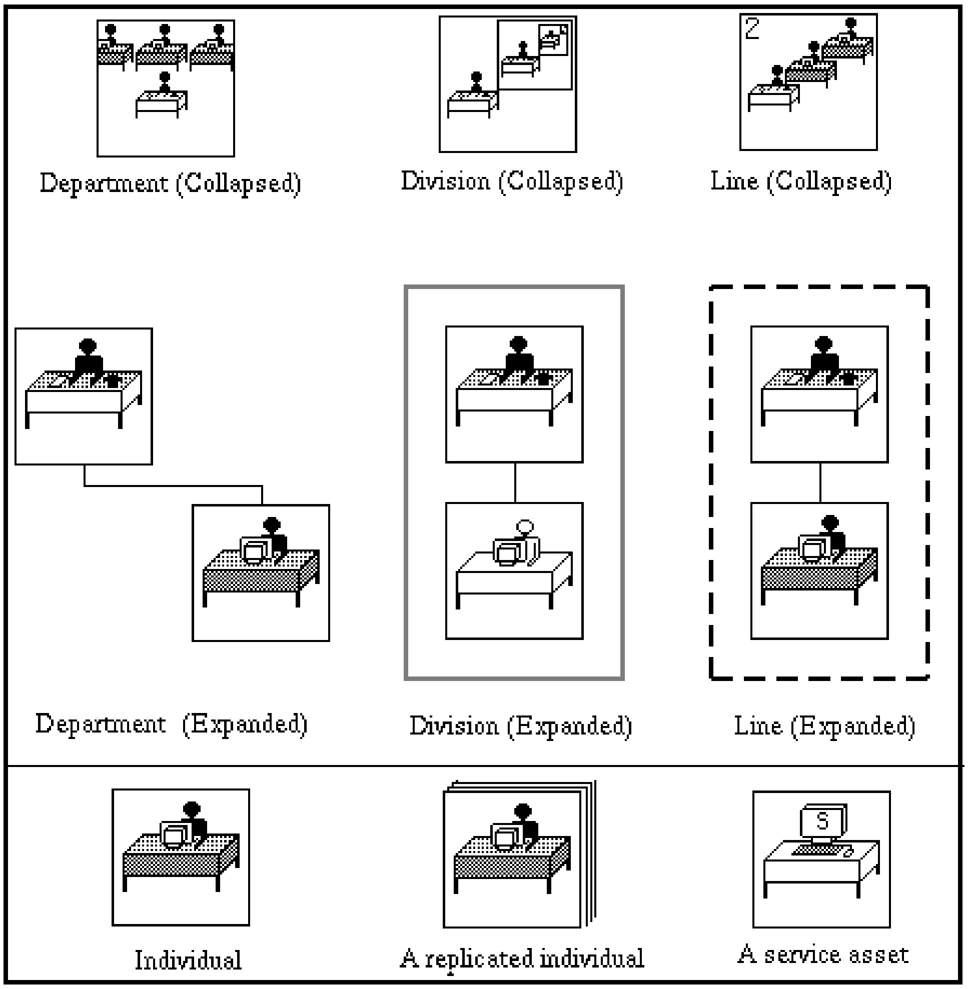

Figure 2.2: The asset types of Enterprise.

sharpeners and time-clocks). An examination of the Enterprise metaprogramming model reveals similar structures with their corresponding parallel behaviors. In Enterprise, a worker who spends time processing work is a resource and is thus an *asset*. *Departments*, *divisions* and *lines* are all special sub-organizational structures also found in Enterprise. Each structure represents a parallel organizational scheme and serves as a template for a particular parallel behavior. Even interprocess communication can be viewed in terms of a business metaphor when one realizes that communication is a vital component to any successful business venture. Under these terms, one Enterprise asset sending a message to another becomes the equivalent of one worker of an organization sending a written note, or e-mail, to another.

The following sections provide a more detailed description of the metaprogramming model's structures and their use. The assets as they appear in the program graph are shown in figure 2.2.

2.3.1 The Individual Asset

The individual asset is the basic building block of the Enterprise model. Whenever a new asset is created in the program graph, it starts as an individual. If a more interesting organizational structure is desired, an individual asset can be transformed into a *composite asset* (discussed in the next section). Otherwise, an individual's purpose is to accept a task, perform the work required from it and send back a reply (if one is requested). As such, an individual is a *codable asset*, one in which a programmer enters C source code to be executed. The source code of a codable asset is always executed sequentially. Parallelism in Enterprise is achieved by launching several assets on several processors with each asset launched executing concurrently with the others.

2.3.2 Composite Assets

Composite assets are compound entities which are made up of several assets and have a pre-defined structure and parallel behavior. Assets that make up the composite asset may either be individuals doing a user's work or can themselves be transformed (coerced) into composite assets to create organizations of greater complexity. The exception to the rule is the *receptionist*, the first asset found in any composite structure. The receptionist is a special form of an individual which serves as the entry point to the composite and can never be coerced. It is responsible for distributing work which it receives (or generates) to the other members of the composite asset. Because it is a specialized individual, the receptionist is a codable asset.

Currently, there are three composite asset types: *lines*, *departments* and *divisions*. The *line* asset is akin to an assembly line found in many factories. This asset is composed of the receptionist and one or more assets linked in sequence to form a line. Output of the first asset is sent as input to the second asset in the line whose output is sent as input to the third asset in line an so on. Work passes from one asset to the next sequentially. However, since each asset is executing concurrently the net throughput is increased.

Like the line, the *department* is composed of a receptionist and one or more assets which can either be individuals or composite assets. Unlike the line, there is no implicit ordering of the assets other than the receptionist being the first asset of the structure. Each asset of the department executes concurrently and is independent of the others. Further parallelism is achieved by allowing individuals to be *replicated* (discussed in a later section).

The *division* differs from the line and department in several respects. First, a division is the only recursive asset of the Enterprise model and is used to implement parallel recursion for divide-and-conquer algorithms. Second, although a division does employ a receptionist, the only other assets permitted in a division are either a *representative*, a codable asset which holds the recursive code, or another division. Finally, unlike other composite assets, the receptionist and the representative must share the same user code.

2.3.3 Service Assets

A service asset is comparable to a public resource such as a photocopier, a library or a wall clock and is used to implement access control for a shared resource (such as shared files). Services are unique in Enterprise in that they are the only assets that can be called by any other non-service asset of the program. A service is a codable asset.

2.3.4 Replication and Managers

In many situations, it is possible for an Enterprise asset to become overworked as tasks arrive more quickly than they can be processed. Under this scenario, outstanding tasks are queued until the asset finishes its current job at which point the wait queue will be drained one task at a time. If message queuing is a persistent problem, program throughput suffers as tasks sit in the queue. In keeping with the business metaphor, the response of a business manager in such a bind would be to hire more help (provided that the resources are available). In Enterprise, a similar strategy called *replication* is used. A programmer "hires" more assets through the "replication" process, creating identical copies of an individual[1]. These copies, or *replicas*, take the form of additional worker processes that are launched at run-time. The work directed to the replicated asset is spread over all the replicas ensuring that the bottleneck effect is lessened. Replication, however, does not come without cost.

The hidden cost to the replication procedure is the creation of *managers*, special processes that are automatically launched by Enterprise to manage the distribution of work to replicas. As with any

[1] A more accurate term would be "cloning".

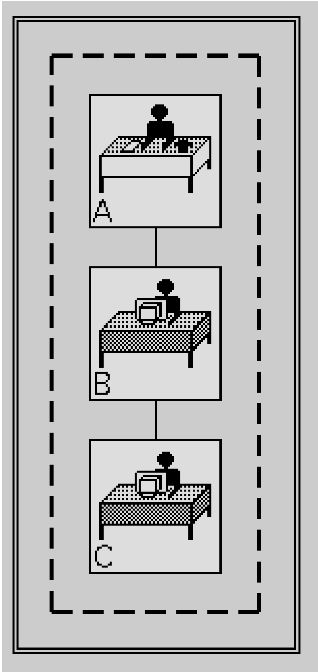

Figure 2.3: A line asset of Enterprise.

other Enterprise asset, a manager is a Unix process that is spawned on a computer's processor and requires use of the same resources as any other asset. Unlike other Enterprise assets, the manager process is outside the programmer's control.

2.4 The Programming Model

The programming language for Enterprise is standard ANSI C with a few special qualifications that arise due to the inherent differences between parallel and sequential computing. At the highest level, an Enterprise program appears to be a collection of programmable modules (codable assets) organized in a hierarchy as specified by the program graph. These assets are launched on the machines over the network by the executor and execute concurrently.

Each module contains a functional header listing the number and the types of its calling parameters and serves as the asset's interface protocol. Communication between assets is accomplished by "calling" the asset with the appropriate parameters, a process that is similar to invoking a function or a procedure. In reality, the calling parameters are packaged into a message and shunted to the recipient asset under the care of the communication manager. Communication channels between assets are governed by the program graph which imposes strict restrictions as to which assets can communicate with each other. As an example of this control mechanism, consider a depth three line asset shown in Figure 2.3. Asset **A** can send a message only to **B**, the second asset in the line. There is no link from **A** to **C** thus **A** can never call asset **C** directly.

2.4.1 p-Calls, f-Calls, Futures and Lazy Synchronization

There are two types of asset calls in Enterprise: *p-calls* and *f-calls*. A p-call is similar to a procedure call where there are no side effects to parameters and no return values are expected by the caller. An asset making a p-call can continue to execute concurrently with the recipient asset after the call is made because there is no dependency on returned values or side effects thus avoiding the need to synchronize behaviors (asynchronous parallelism). In contrast, an f-call is comparable to a function call where side effects and return values are expected. An asset making a f-call wants to make use of a return value from the called asset at some point. The returned value or side effect is called a *future*. If the calling asset tries to make use of a future before it has been evaluated, the asset must *block* itself from executing any further until the value is known. As an optimization step, the calling asset will avoid blocking until it requires use of a future (lazy synchronization).

2.4.2 Parameter Passing Macros

The comparison of an asset call to the invocation of a procedure or function call is not a chance occurrence. From the onset, Enterprise was meant to be a tool accessible by programmers unfamiliar with parallel programming intricacies. A deliberate effort was made to preserve the syntax and semantics of C, a widely used language that was to be the basis of an Enterprise program. Thus an Enterprise user need not learn a new language or even a new set of commands. However, one compromise had to be made.

Passing scalar data types (integer, float, double) as parameters in C is understood to be a "pass by value" mechanism, a semantic *mostly* preserved by Enterprise. The exception is due to a particularly troublesome problem that is most noticeable when vector data types, such as arrays, are passed as parameters. Arrays as a parameter of a function or procedure are usually handled in C by using a pass-by-reference approach. However, passing a reference is meaningless if the reference is to a location in the memory space of a different processor as is the case in distributed computing. Shared memory between processors would alleviate this problem, however that option is not currently available in Enterprise. In order to achieve the proper behavior, Enterprise provides three macros that emulate the pass by reference approach for arrays. Instead of sending a memory location reference as a parameter, Enterprise packages the elements of the array into a message that is sent from one asset to another. To reduce the size of the messages sent in this manner, each macro includes a single parameter which specifies the number of array elements to be sent in the message.

- **IN_PARAM()** specifies that the array gets packed into a message by the caller and sent to the called asset, but no values are returned,
- **OUT_PARAM()** specifies that no initial values are sent to the called asset, but that array values are packed into a message to be sent to the caller by the called asset when it returns,
- **INOUT_PARAM()** specifies that the array is passed as a message in both directions.

2.5 Program Execution In Enterprise

Running an Enterprise program triggers a sequence of events that are normally invisible to the user. First, Enterprise performs a quick series of checks to verify that the executable code and the source code are both current and consistent. If the source code and executable code are inconsistent with each other, the operation is aborted and the user is notified of the error. Second, if the source code and program graph are consistent, the root process is launched. Finally, the root process itself launches the remainder of the Enterprise processes and signals the true start of the program.

The root process uses the program graph in conjunction with a list of available workstations to map assets to the available processors. The default process-to-processor binding step attempts to create a one-to-one binding. Failing that, surplus assets are launched on one of the already populated machines. For replicated assets, the replication factor is checked and the replicas, along with their managers, are mapped to processors and launched.

At any time during a program's execution, an asset is in one of four execution states:

- **idle** where the asset is alive and waiting to do the user's work,
- **busy** where the asset is actively processing the user's work,
- **waiting** where the asset has suspended its execution pending the arrival of an outstanding message or,

- **dead** where the asset has finished all its work and has been shut down.

When an asset is first launched it is idle and awaiting work. Once an asset receives work it enters into a busy state. If an asset issues an f-call and tries to use a future before it is available it enters the blocked state. Once the future is available, the asset once again enters a busy state. When an asset finishes its assigned task it returns to an idle state and the cycle begins anew. An asset enters the dead state once all work is completed.

2.6 Performance Issues in Enterprise

Given a well designed application and ideal execution conditions, an Enterprise program will produce an execution run with appreciable speedups. However, conditions are not always ideal and many factors, some beyond the user's control, will impact on how well a program performs. The sections that follow discuss some of the pertinent factors that may hinder a program's performance.

2.6.1 Application Design

While it is a relatively simple task to parallelize some sequential programs using Enterprise, a simple translation will not always produce the best results. The conceptual mapping of an asset to a function (or procedure) provides a simple guide to define a program graph. However, a message passing system requires that the problem granularity be coarse enough that the time a message spends in transit does not overshadow the time spent being processed. If the translation results in assets that do fine-grain parallelism, the program performance will suffer.

Similarly, using a specific parallelization strategy unwisely will also result in unfavorable program performance. For example, a poor choice of a replication factor can hinder a program. If an asset is issuing f-calls to an under-replicated asset, the replicas may become overworked resulting in an accumulation of unprocessed messages at the manager with a corresponding decrease in throughput. The flip side to this problem occurs when an overly replicated asset is underworked and spends a disproportionate amount of its life idle. In this case, a processor that could be doing other work is wasted.

2.6.2 Processor Load

An Enterprise asset running on a loaded processor will perform poorly. All the processes running on a workstation are in constant contention for the finite resources of the host. For example, an Enterprise asset requiring extensive use of the CPU may be "starved out" as other processes take their share.

2.6.3 The Network

The communication characteristics of the network connecting the hosts has a direct bearing on how well a given Enterprise program performs. For example, a program run on a busy network will have significantly different performance characteristics than one run on a quiet network. Likewise, the topology, the distance between hosts and the bridges between different networks all have an impact on how well a program performs. The Enterprise system distributes work to assets via messages. If these messages are throttled on the network, the delay they experience translates into wasted computing cycles and is thus a performance penalty.

2.6.4 Managers

Managers have two important consequences with respect to a program's performance. First, a manager is a Unix process like any other Enterprise asset and thus consumes both processor cycles and memory. The implication is that an individual replicated seven times is actually *eight* processes: the seven replicas *and the manager*. Should the manager be launched on a machine already hosting an Enterprise asset, both processes must vie for the machine's resources. Fortunately, the manager acts only as a distributor and does not require extensive use of the workstation's processor.

The second consequence of managers is that they form a link in the communication chain. For a manager to forward a message to a replica, it must first receive the message in its entirety before it can relay the message to the replica (store-and-forward). This makes the task of sending a message from a caller to an individual a "two jump" process which is costly if the messages prove to be large or if the network is congested. In order to lessen the effects of this problem, an attempt is made to launch the manager on the same processor as the caller asset. Instead of sending the message to the manager via the external network, the caller asset utilizes the processor's local bus to accelerate the first jump to the manager.

The second consequence suggests a potentially greater problem: *manager overloading*. During program execution, it is possible for a manager to be swamped by messages. The manager is the liaison between a sending asset and the replicas, and is continually relaying messages between them. Each message must be handled sequentially and in the order it was received. Thus incoming messages that cannot be processed until their turn has come must join a queue. During periods of high message passing activity, the manager will form a bottleneck as messages waiting to be relayed sit inactive in the ever-growing queue.

2.6.5 Gathering of Logging Information

Paradoxically, the method used to help determine how well a program performs can itself influence the program's overall performance. Gathering events requires the assets to perform a self examination and log the information as the program is running. An asset who is being monitored is therefore spending some of its precious processor cycles to do a chore that is not relevant to the task for which it was created. Normally, the effect of executing the performance logging code is so small that it is negligible to the overall performance of the program. However, if the perturbations influence the execution of the code, as it would during execution of chaotic systems of computations, then performance signatures surely change as well.

Chapter 3

Using the Visualization Tools of Enterprise

3.1 Overview

The ultimate measure of any tool's worth is gauged by how well it fulfills the needs of its user. However, this is not to claim that if a tool is useful, it will be used. A chef would not use a blowtorch to open a can of soup nor would a welder use a can-opener to cut sheets of metal (though it is possible to do both). Blowtorches and can-openers are both useful tools, but they are designed to fulfill different needs. A person seeks the use of a tool to make a job easier, not more difficult. Thus, for a tool to be used it must satisfy two criteria:

1. the tool should be suited to the user's task and,
2. the benefits gained from using the tool must offset the effort required to use it.

The goal of this chapter is twofold. First, it provides a preliminary look at the performance visualization views of Enterprise. Second, it demonstrates that these views satisfy the two criteria outlined above. This is accomplished by examining three commonly occuring performance tuning scenarios, using three different applications, from a program developer's perspective. The next chapter discusses the tools introduced here and the issues involved in greater depth and detail.

3.2 The Three Scenarios

The best way to demonstrate the usefulness of the Enterprise performance analysis and visualization tools is to show how each tool would be used in practice. In this case, the goal is to assist a programmer to find and correct flaws that degrade a parallel application's overall performance. Thus in keeping with a "visualization over verbosity" philosophy, three example scenarios will be examined in lieu of lengthy descriptions. Each scenario presented here represents a different facet of how a program's performance can falter and serves to demonstrate how the visualization tools can help the user rapidly identify and correct the problem.

Because the performance issues of Enterprise programs are the primary focus of the chapter, full program and source code descriptions are not needed. Instead, a general description of common performance debugging scenarios, their causes, and how the visualization tools would help correct the problem are all that is required. For simplicity, the "programs" examined here are considered syntactically and semantically correct.

An important point to consider while reviewing the following sections is that all three scenarios share a common trait: from a user's perspective, nothing appears to be amiss with the program. In all three situations, the program will always run to completion (barring any unusual circumstances) and produce the correct result(s). Use of the replay and animation views to visualize the execution of the program would reveal only that the program appears to be running as expected. Based on the evidence at hand, only a programmer who is unusually suspicious, or looking for better performance, would even bother to look at the assets in enough depth to find any specific performance problems.

3.2.1 Scenario 1: One or more Replicated Assets are Under-utilized or Over-utilized

A programmer has written a program using the Enterprise environment and has determined that there is enough work sent to an asset to justify the use of replicas to increase the parallelism of the application. The programmer replicates the asset and tests the program but is unaware that a poor replication factor has been chosen. If the replication factor is too small, the replicas will be overworked, messages will be queued and the additional expenses associated with the manager may become more pronounced. If the replication factor is set too high, then the some replicas will not receive enough work to justify their existence and will sit idle. If the the replication factor is set abnormally high, then the added replicas will not only be underworked, but would likely have been launched on machines already hosting Enterprise assets. The wasted replicas "steal" the workstation's resources (memory, processor cycles) from the other assets who may be doing useful work and thus overall performance is degraded.

3.2.2 Scenario 2: An Asset of a Line becomes a Bottleneck

Given a line of two or more assets, a problem will occur when the throughput of one asset is significantly less than that of the other assets in the line. The flow of work through the pipe is reduced to a trickle at the offending asset and all assets in the line that follow must sit idle while waiting for work. Because one asset in the line is experiencing difficulties, all subsequent assets suffer and thus the overall performance will suffer as well.

3.2.3 Scenario 3: Performance Tuning a Working Program

Of the three scenarios being considered, this is perhaps the most interesting. A programmer has created a working application that produces noticeable but unimpressive speed-ups. The next question that the programmer would ask is "can I do better?" Although any decrease in computation time can be viewed as a victory, most serious programmers are looking for the best possible performance gain that can be achieved given the available resources. However, the question as to how a programmer can determine whether or not a resource is being used to its fullest potential remains unanswered.

3.3 Exploring the Three Scenarios

3.3.1 Scenario 1 and the Asset Utilization View

The Asset Utilization View, shown in Figure 3.1, is used to display a visual summary of how much time each Enterprise asset spends in the three execution states (*busy*, *idle*, and *blocked*).

Each asset's state information is represented by a color-coded pie-chart graph where each wedge of the graph represents the proportional amount of time (with respect to the elapsed execution time of the program) spent in a given state. A wedge's color identifies the state it represents and corresponds to the color-state mapping used in the animation view: green for busy, red for blocked, yellow for idle. If a non-color display is used, it is still be possible to identify the different summarized values. Starting at "12:00" on the chart and proceeding clockwise, the order of the wedges is always busy, idle, and blocked. If the wedges are too small to identify, or if the contrast of dithered colors is insufficient to differentiate the wedges, the numerical statistics on the right side of the chart can be used (the numbers of the chart are explained in the next chapter). For expediency, a color display is assumed.

Figure 3.1: The Asset Utilization View of underworked replicas.

Figure 3.1 shows the asset utilization of a simple program called **Example1** which has one asset, named **Worker**. The asset **Worker** is replicated three times and thus consists of a manager, **Worker.1.1** and four replicas: **Worker.1.1.1**, **Worker.1.1.2**, **Worker.1.1.3** and **Worker.1.1.4**. Disregarding the numbers associated with each pie-chart for the moment, it should be clear from these charts alone that the replicas **Worker.1.1.2** and **Worker.1.1.4** have different utilization histories than the first and third replicas, **Worker.1.1.1** and **Worker.1.1.3**. Comparing the pie-chart display of the second and fourth replicated assets with those of the first and third replicas clearly shows that the busy time of the **Worker.1.1.2** and **Worker.1.1.4** are significantly less than that of the others. A user viewing this information should begin to suspect that something is amiss and that perhaps the replication bears further scrutiny.

To verify that a problem exists with the replication factor of **Worker**, the program should be re-run in order to determine that the results are consistent. If the results of disparate runs match (one replica always being underworked) the user must then look into correcting the problem of an underutilized replica. In this case, changing the replication factor from four to three and re-executing the program should produce the desired result: the three remaining replicas should show an increase in their busy times but the program should run to completion nearly as quickly as it had when it made use of four replicas. The program uses fewer system resources for the same results and is therefore a "win" for the programmer.

The flip-side to the previous example, the problem of overworked assets, is slightly more difficult to identify. Using a variation of the program **Example1**, consider the Asset Utilization view shown in Figure 3.2. Unlike the first example, no asset of Figure 3.2 stands out from the others (with the exception of the manager, **Worker.1.1**, discussed later). Both the first and second replicas of **Worker** show nearly uniform busy times for most of the program's execution. This by itself is not a cause for alarm considering that a good program keeps assets as busy as possible without overburdening them. However, if it is found that the two replicas of **Worker** are continually busy, then there is the possibility that the replicas are *overworked.* That is, each replica is given so many tasks to process that they cannot keep up with the demand placed upon them. Experimenting with the replication factor and re-running the program would reveal whether or not additional replicas can ease the burden placed on the assets while doing enough work to justify their existence.

The managers shown in Figures 3.2 bear further discussion. By their appearance on the display, the manager **Worker.1.1** has all the earmarks of an overworked asset. However, this is not truly the case. A manager is responsible for relaying messages, not doing a user's work. A manager who

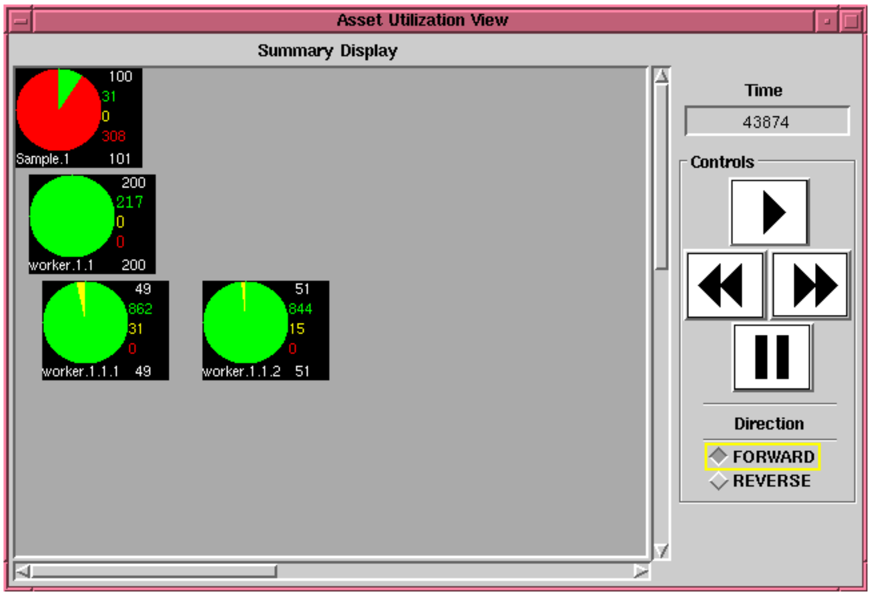

Figure 3.2: A view of two potentially overworked replicas.

is constantly busy is therefore one who is fulfilling its duties. A manager who is idle or blocked for any significant amount of time is thus a cause for worry.

3.3.2 Scenario 2 and the Transaction Summary and Transaction Time-line Views

The Asset Utilization View discussed in the previous section shows the cumulative effects of each message arriving at an Enterprise asset and is a means of examining the working efficiency of an asset. What the Asset Utilization View fails to capture is that some messages being exchanged may have a greater impact on an asset's workload than others. However, the Transaction Summary and Transaction Time-line Views do not overlook this fact.

The Transaction Time-line View summarizes the message passing history of a program by graphically plotting the path of a message as it moves from asset to asset against an execution time-line. Each Enterprise asset is represented by a horizontal *asset line* that crosses the "Y" coordinate of a Time-line graph at a specific point. A message sent from one asset to another is recorded as a colored *message line* connecting two asset lines. The time a message was sent by one asset is the "X" coordinate of the lines starting point. The time the message is received by another asset serves as the "X" coordinate of the endpoint of the line. The full message trace of a message is thus the connection of all message lines for a given message.

The Transaction Summary View provides an indexed list of each message sent during the execution of an Enterprise program. The list references all pertinent transaction information about a message excluding the actual contents of the message (a message's contents are irrelevant from a performance analysis perspective). With this view, it is possible to quickly identify the message's originator, to determine the time it was created, and to trace the path it took as it moved from asset to asset. The graphical charts of this view reflects both the time a message spends in transit between assets and the time it spends at an asset being processed. The *Global Summary* portion of this view displays the cumulative statistical information about messages including the minimum, maximum and average time these messages spent being processed and in transit.

Returning to the problem suggested in Scenario 2, the line asset bottleneck, consider how a message based perspective could help solve this problem. A simple program named **Example2** shown in Figure 3.3 is used as an example. For the purposes of this discussion, asset **Processing** is the bottleneck of the line. For reasons unknown to the programmer, **Processing** is taking more

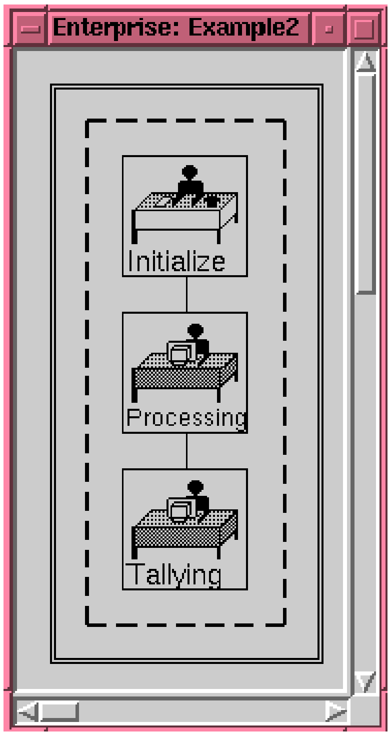

Figure 3.3: A program graph of a three asset line asset.

time than expected to process the messages it receives. This results in a large backlog of work at **Processing** leaving **Tallying** starved for work.

Using the Asset Utilization View, a user should be able to quickly identify the delinquent asset. However, knowing who the offender is and knowing why the asset is an offender are two different problems. The cause of the degradation may no longer be as simple as a replication factor error, but could instead be any one of a multitude of problems. Perhaps the asset is being run on a heavily loaded machine. Perhaps the amount of time needed by the asset to process a message has been underestimated by the programmer. Of the possibilities listed, the most problematic situation would be for the processing time of a message to be dependent on the contents of the message itself. In this case, sporadic messages which are difficult to trace would cause program performance to suffer for only a brief period of time, yet still manage to cause the program's performance to suffer. Regardless of why the asset may be bottlenecked, there is a limit as to the amount of information that can be obtained from the Asset Utilization View.

Because the problem with the program is attributed to a congested line, the first course of action would be to consult with the Transaction Time-line View in order to determine what is happening when messages are being sent. Figure 3.4 shows the Transaction Time-line View of the program **Example2**. Presented in this view are the three asset lines for the assets (**Initialize**, **Processing** and **Tallying**) and the message lines showing the communication between the assets as the program was executing. From the time-line view of **Example2**, there are several important details that should be noted:

1. **Initialize** sends all the work it wishes to send to **Processing** in a small amount of time as evidenced by the common transmission send time (upper left corner of Figure 3.4).

2. The messages sent from **Initialize** to **Processing** are represented on the display by message lines whose slopes are gradually decreasing. This implies that either the travel time between the two assets is steadily increasing or that the messages are spending increasingly longer amounts of time in **Processing's** input queue.

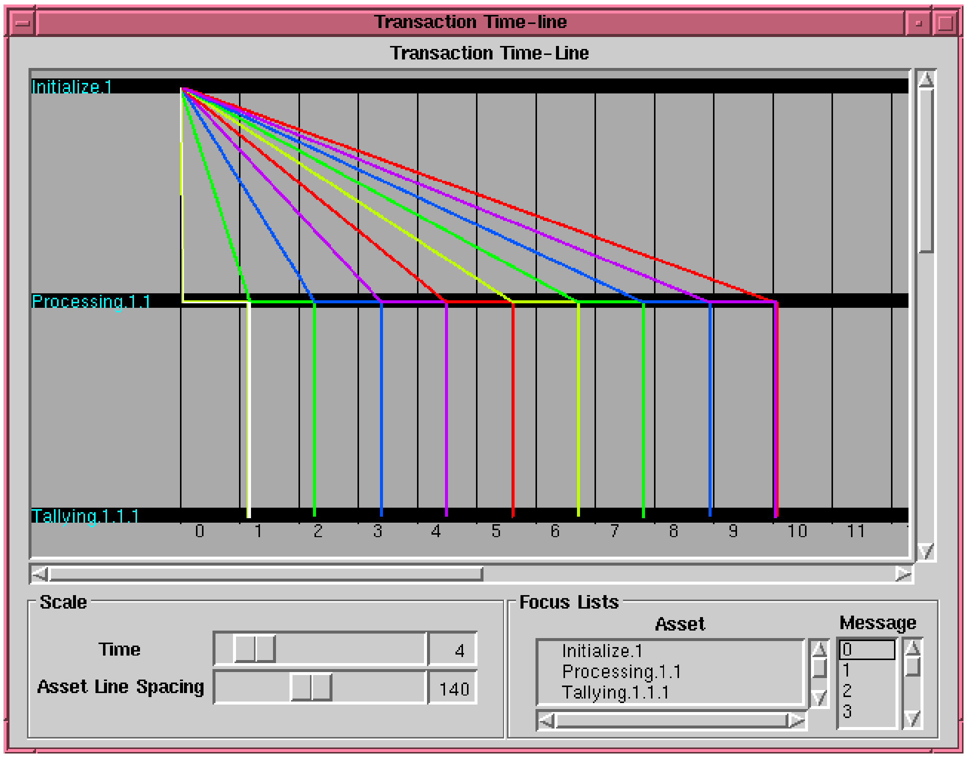

Figure 3.4: The Transaction Time-line View.

3. The horizontal message lines found at **Processing** indicates that the messages are spending a large portion of their existence being processed at **Processing**.

4. The amount of time each message requires to be processed at **Processing** appears to be constant.

5. Messages sent from **Processing** to **Tallying** appear as nearly vertical message lines on the display indicating that the travel time between these two assets is small and that **Tallying** consumes the messages quickly.

The key to solving the problem is to notice that although all the work is sent from **Initialize** to **Processing** in a very short span of time (Point 1), it takes an increasingly longer amounts of time for messages to be consumed (Point 2). Because it appears that **Processing** is constantly busy, and is taking nearly the same amount of time to process each message (Point 3), it is reasonable to assume that messages that arrive at **Processing** are being processed as quickly as **Processing** can handle. Therefore a logical conclusion is that all incoming messages at **Processing** are being placed in its input queue and must wait a long time before they are processed, a symptom that **Processing** is overworked.

The Transaction Summary view (Figure 3.5) offers a less abstract view of the data contained in the Transaction Time-line view and provides a compact summary of a message's transaction data. This view offers the "hard numbers" lacking in the Time-line view.

In the case of the program **Example2**, this view confirms many of the observations made (or assumed) after viewing the Transaction Time-line view. The *Global Summary* area of this view presents the statistical summaries of all messages sent as the program was executing. Looking at the range of *Send and Consume* times verifies that there is a wide variation of transmission and consumptions times as evidenced by the large gap between the maximum and minimum **Send and Consume** times. Because there are no return values of this program (no asset ever sends a reply to an asset who gave it work), there are no **Reply and Consume** statistics, hence the zero entries.

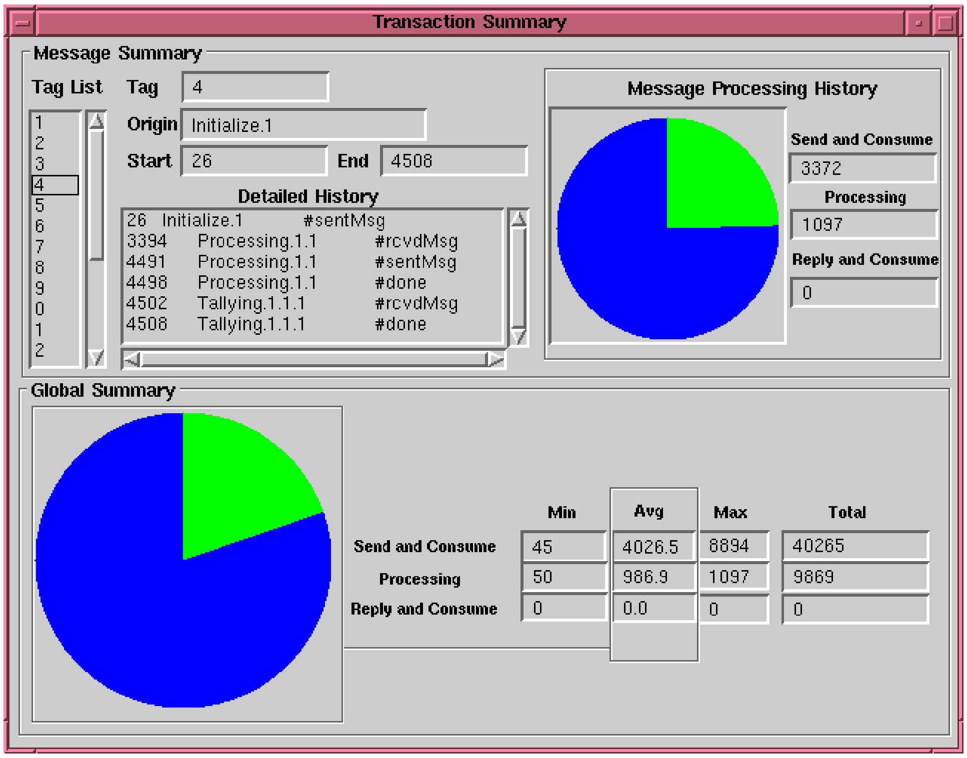

Figure 3.5: The Transaction Summary View.

For an example as simple as **Example2**, this view may seem superfluous. However, in more complex systems where there are more messages and more assets, a means of singling out one message from thousands is a welcome ability.

3.3.3 Scenario 3 and the Performance Annotation View

The third scenario, fine-tuning an existing application, poses an interesting problem. In a sense, the goal of writing a parallel application has been accomplished: a program exists, it produces the correct results, and all obvious flaws have been corrected. Now, the user must re-evaluate the program and its performance in an attempt to improve the program. The new objective requires the user to match the actual program gains against the expected or desired gains. This proposition is never easy when dealing with a parallel programming system. A user can always vary the number of processors used, change the parallelism of the system (the graph) or modify their source code. Enterprise users can even alter the method by which assets are mapped to processors. Unfortunately, changing parameters may not always produce the expected results. External factors such as processor load and network usage are unpredictable and can influence how a program behaves. To compound the matter, changing one parameter can affect how other parameters effect the execution of the program.

Suppose an Enterprise programmer is attempting to fine-tune a program, but no performance monitoring tools are present. The procedure he or she might follow would most likely resemble the following procedure:

1. Change the program graph and clock the execution time.

2. Re-run the program several times to study the effects of the changes.

3. Draw a conclusion about the changed aspect.

4. Change another aspect and do similar experiments and timings.

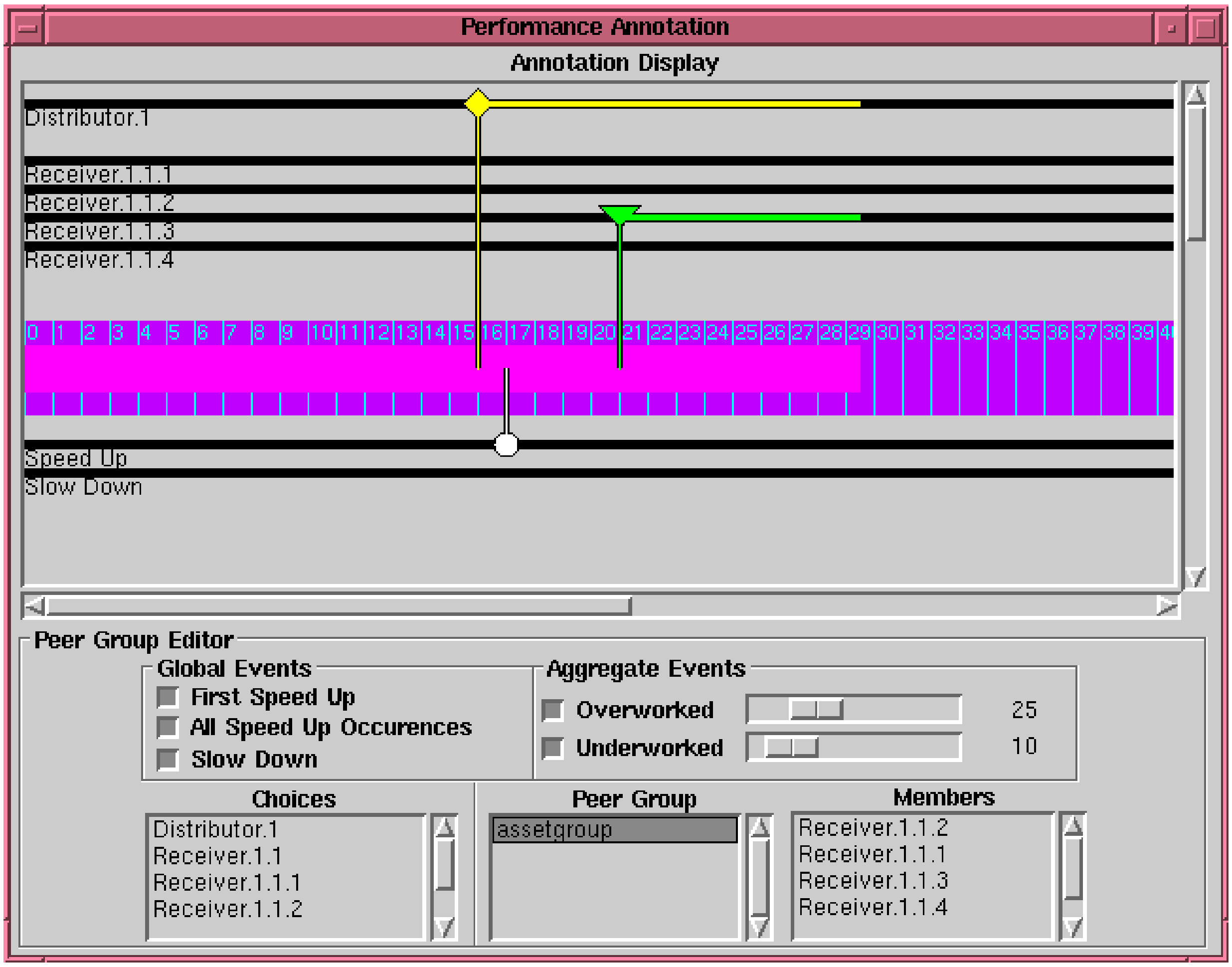

Figure 3.6: The Performance Annotation View.

5. Stop when the program is "good enough" or when time, money or patience runs out.

This exhaustive testing methodology may sometimes work, but its "hit and miss" aspects are both wasteful and prone to errors in interpretation. The programmer only sees the net effect of the changes by how execution time is affected, never the other consequences of the change. Undiscovered possibilities resulting from various combinations of changes to the program may be lost unless the programmer is fortunate enough to stumble upon the proper combination of actions. A methodical approach to predicting effects of changes may achieve the desired results, but the strategy still requires a substantial amount time and effort on the part of the programmer. The Performance Annotation view is a tool to address this problem.

The Performance Annotation View allows a user to group assets and have their behaviors automatically checked as the program is running. If a group of assets experiences some unusual condition or occurrence, the event is recorded and identified as an *annotated event* on the Annotation Display.

Consider the Performance Annotation view (Figure 3.6) for a program **Example3** which consists of the receptionist, **Distributor.1**, and a replicated asset **Receiver** consisting of the manager **Receiver.1.1** (not shown in Figure 3.6) and its four replicas (**Receiver.1.1.1**, **Receiver.1.1.2**, **Receiver.1.1.3** and **Receiver.1.1.4**. Before the program was executed and the data shown on the Annotation Display was obtained, the user had to decide what type of events were to be detected and which assets were to be examined for such events. The user created a group of assets named *assetgroup* consisting of the assets **Receiver.1.1.1**, **Receiver.1.1.2**, **Distributor.1** and **Receiver.1.1.3** with the **Peer Group Editor**.

Of the assets belonging to *assetgroup*, the programmer is interested to know which assets do 25% more of the work than is done by the other assets of the group. That is, which assets are *overworked* when compared with the other members. Similarly, the programmer wishes to know which assets are doing less than 10% of the work done by the members of assetgroup. That is, the user wants to know which assets are *underworked*. These events that are derived from an examination of a grouping of assets are known as *aggregate events*. The Peer Group Editor of Figure 3.6 shows the Aggregate Event checklist with the event types and their corresponding comparison percentages selected.

Finally, the programmer wishes to know when the program experiences its first speed-up, whether or or not the program experiences a slow-down, and whether or not the program ever recovers from a slow-down. These quantities that encompass *all* Enterprise assets are know as *Global Events*. Like the aggregate event, these items are also entered via the Peer Group Editor.

Once the asset group is defined and all the events of interest are selected, the program is executed. As the program is executing, several things occur:

1. The elapsed execution time is displayed on the *execution time-line* which is divided into one second intervals. Figure 3.6 shows the execution of the program **Example3** to have ended at just after 28 seconds.

2. The asset groups are checked to see if any aggregate events occur. In Figure 3.6 two such events were found and displayed:

   (a) Asset **Distributor.1** was found to be *underworked* as indicated by the diamond icon found on the *Distributor.1* line at the 16 second mark. An "underworked" annotation appears as a yellow diamond icon and yellow and lines on a color display. The horizontal line leaving the icon is indicative of the duration of the event. In this case, **Distributor.1** remained underworked until the end of the program.

   (b) Asset **Receiver.1.1.3** was found to be *overworked* at the 21 second mark. This annotation appears on the display as an inverted triangle on the *Receiver.1.1.3* line of the Annotation Display. The horizontal line also indicates that the condition that triggered the "overworked" event lasts until the end of the program. On a color display, an "overworked" annotation appears green.

3. All program assets are examined as a whole in order to find out if the program experiences any speed-ups or slow-downs. In Figure 3.6, a speed-up is reported at the 17 second mark (shown as the white circle annotation found on the Speed Up line). The speed up occurs well after the program run began.

With respect to **Example3**, the Performance Annotation View shown in Figure 3.6 to indicates that the program does not experience a speed-up until roughly two-third's of the program's execution has passed. Just before that point, the asset **Distributor.1** becomes underworked. Though no *definite* conclusions may be drawn, the view could be used to make an informed "guess" as to why the phenomena occurs. A potential reason that the program might not experience a speed-up earlier is that **Distributor.1** is performing some initialization actions and distributing work to the replicas for processing. Only once the replicas have all their tasks may a speed-up occur. This supposition is consistent with the display, even to the inclusion of the overworked event found at **Receiver.1.1.3** (the fact that a replica becomes overworked indicates that all the replicas are busy). Thus a possible course of action would be to concentrate on optimizing **Distributor.1** so that optimizing or initialization actions can proceed more quickly. If a speed-up can be triggered sooner, then the program may run to completion more quickly.

3.4 Chapter Summary

Does an application programmer using Enterprise need to use performance analysis tools? *No.* A programmer does not need to spend time tuning a program's performance. However, the demanding programmer *does* want to tune the program and would most assuredly want to have a tool that helps them achieve this goal. The *Asset Utilization View*, the *Transaction Summary View*, the *Transaction Time-line View* and the *Performance Annotation View* are all available to help a programmer enhance a program's performance. Each view fulfills a simple basic function, but one of their greatest strengths is that they complement each other and can be used in conjunction to help the programmer. The greatest benefit reaped is that an Enterprise user no longer needs to manually insert instrumentation code or analyze complex trace data in order to draw reasonable, informed conclusions.

Chapter 4

The Performance Visualization Views of Enterprise

4.1 Overview

The preceding chapter discussed how an application developer would use the performance visualization tools of Enterprise to fine-tune an Enterprise program. An examination of three typical scenarios served to emphasize that performance tuning is an achievable goal by demonstrating how each of the Enterprise performance views would be used in the scenarios. This chapter expands upon the topics introduced in the last chapter by providing greater insight into the form and function of the performance visualization views themselves.

Fine-tuning a parallel program is an iterative procedure. There are a multitude of factors, some beyond the user's control, that contribute to how well or how poorly a program performs. Stumbling through various combinations of these factors is a frustrating and time consuming chore. The visualization tools of Enterprise are a means to lessen the guesswork.

There are currently four performance analysis and visualization views available in Enterprise: the *Asset Utilization View*, the *Transaction Time-line View*, the *Transaction Summary View* and the *Performance Annotation View*. These views are discussed in the sections that follow.

4.2 The Asset Utilization View

The Asset Utilization View, shown in Figure 4.1, presents a graphical summary of each asset's utilization history and is used to determine whether or not the workload of each asset is appropriate.

The Asset Utilization View is divided into two sections: the *Summary Display* and its *Controls*.

4.2.1 The Summary Display and Asset Summary Diagrams

The Summary Display represents the utilization statistics of each asset as a collection of *asset summary diagrams* arranged to correspond to the program graph. There is one summary diagram for each asset of the program (an example of an asset summary diagram is shown in Figure 4.2). However, unlike the Enterprise design view, managers and all replicas are included in the display.

The primary component of an asset summary diagram is the pie-chart display. Starting at "high-noon" and proceeding clockwise, the wedges of the chart display the proportionate amount of busy time (green), idle time (yellow) and blocked time (red) that the asset experiences. With respect to the total execution time of the program, the more time an asset spends in a given state, the larger the wedge will appear on the display.

To provide a better understanding of how messages affect the workload of the asset, five additional *numeric* utilization statistics are provided. Shown on the right side of the asset summary diagram, these numbers in order from top to bottom represent:

Figure 4.1: The Asset Utilization View.

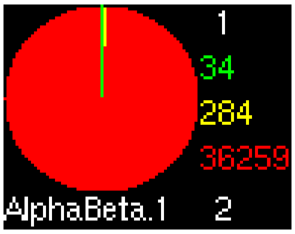

Figure 4.2: An Asset Summary diagram.

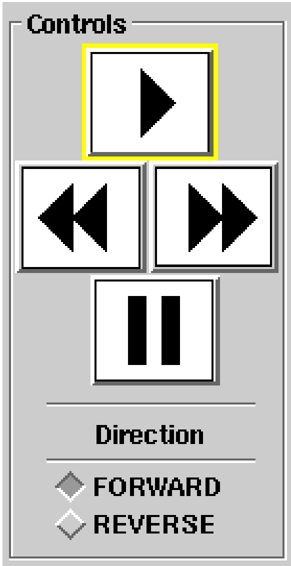

Figure 4.3: The control panel of the Asset Utilization View.

number of messages received the number of messages the asset receives,

busy time per message milli-seconds per message busy time (on average),

idle time per message milli-seconds per message idle time (on average),

blocked time per message milli-seconds per message blocked time (on average) and

number of messages sent the number of messages the asset sends.

The time values are color coordinated with the asset state color mapping of green-busy, yellow-idle and red-blocked.

4.2.2 The Controls of the Asset Utilization View

During the execution of a program, there are often times when the asset utilization statistics may change dramatically. Because the Summary Display is constantly being updated in real-time, the asset utilization changes may be supplanted before the user has had a chance to view them. In order to circumvent this problem, each update is "recorded" for future review. The controls provide a mechanism to examine the recorded updates.

The controls, show in Figure 4.3 are similar in form and function to those of most modern video or audio tape recorders:

The top button is used to start or resume the update replay,

the middle left button "rewinds" the update event recording,

the middle right button "fast-forwards" through the update event recording and

the bottom button pauses the replay of events.

The **Forward** and **Reverse** radio button dictate the direction time flows and thus which direction the replay follows. These controls are enabled once program execution is complete in order to avoid conflicts of displaying new updates in conjunction with recorded updates.

4.2.3 The Importance of Load Balance and the Asset Utilization View

Watching the asset utilization characteristics of an Enterprise program is an important step in the performance tuning process. Correction of nearly all performance problems is a result of changing the work distribution of the program. For example, replicating an asset divides the workload amongst several workers. Creating a line, department or division also divides the workload but imposes a structure defining how the work is divided. Thus the act of tuning a program's performance is the act of modifying the asset utilization characteristics of the program. There are several factors that contribute to a load imbalance, the most notable are:

Poor choice of replication factor. The idea of replicating a worker to increase concurrency is a temptation that is sometimes too great to resist. This leads to a common mistake of creating too many replicas for the amount of work to be done. Like its metaphorical namesake, Enterprise can fall victim to *the law of diminishing returns* where there comes a point when the addition of a worker costs the program more than the worker helps the program. When such a program is run, only a subset of the replicas will ever do any useful work. The surplus replicas will sit idle for most of the program eating up system resources but producing nothing. On the other hand, too small a replication factor will result in some replicas becoming over-worked leading to the reverse problem of queued messages and falling throughput.

Network transmission delays. This problem arises when the transmission time of a message exceeds its processing time. This situation could be attributed to either the granularity of the work performed by the asset or with the propagation of the messages over the network itself. Regardless of its size, a message that spends more time moving between assets than being processed by them is a performance liability. Similarly, workstations who host asset processes but are physically or logically located at a greater distance from the others on the network imply a longer travel time and are also a liability. During extended transmission times, the remote processor sits idle awaiting work. Local clusters of processors have a shorter wait and can therefore acquire and process new work more quickly for higher throughput.

Machine load. As common sense would indicate, Enterprise assets running on a loaded machine will have a lower throughput. Workstations on a network are seldom idle; there are always processes that require both the memory and CPU on a workstation. Enterprise assets must also make use of these same resources thus there is an ongoing competition for CPU and memory. An Enterprise asset running on a "loaded" machine will take longer to perform its task than one running on a "quiet" machine due to the competition for resources.

Managerial bottlenecks. Managers as a bottleneck happens less frequently than the previously discussed situations, but it is a particularly frustrating problem. First, if a manager is launched on a processor with a high load then it will suffers the same performance consequences as any other asset. Second, the "store-and-forward" situation of the manager relaying messages to and from assets is a potential problem. If the manager is flooded with messages and cannot forward them quickly enough it must queue the outstanding messages creating a long wait for messages at the end of the line. Assets waiting for a queued message are then wasted as they sit either blocked or idle until work arrives.

The Asset Utilization View is the first and best indicator that there are performance problems with a given program. As described in scenarios one and two of the last chapter, assets that are over-utilized or under-utilized are readily noticeable using the graphical display of the Asset Utilization View, the situation changes.

4.3 The Transaction Time-line View

The Transaction Time-line View, shown in Figure 4.4, is complementary to the Message Transaction View and is used to track message transmission patterns. This view is a graphical abstraction of the Detailed History table found on the Transaction Summary view.

In this view, each Enterprise asset is given a time-line representation on the Transaction Time-line display. The horizontal portion of these time-lines, from left to right, corresponds to the elapsed program execution time (the horizontal bars divide the time-lines into one second intervals). Each

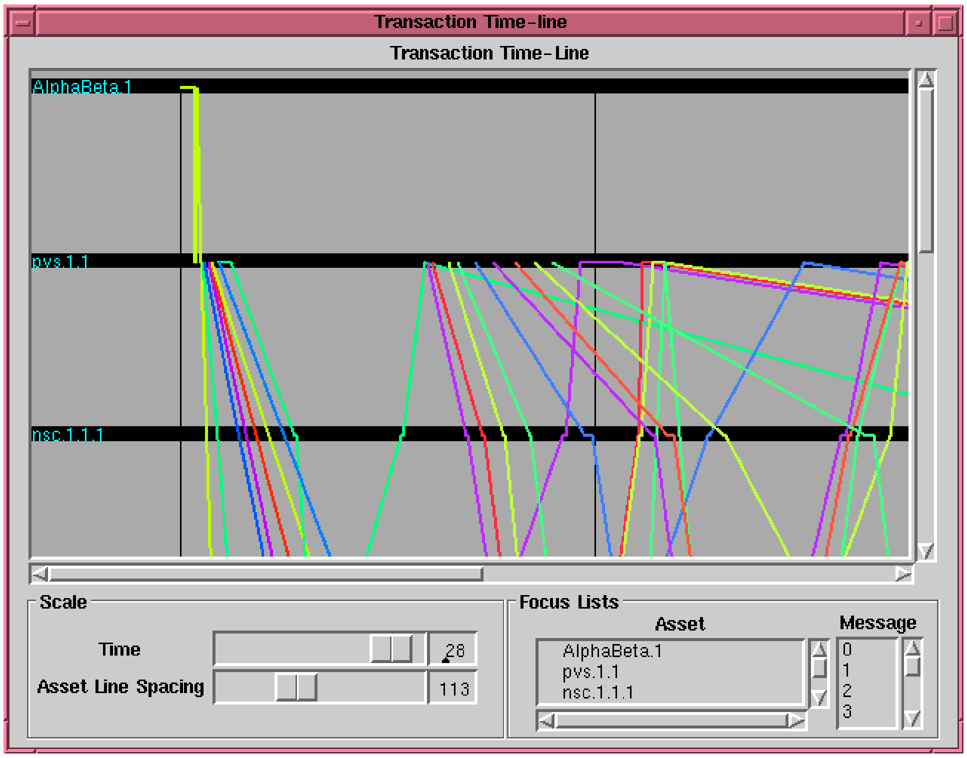

Figure 4.4: The Transaction Time-Line View.

Enterprise message transmitted during the execution of a program is represented by a series of continuous *message lines* that connect the various asset time-lines to form a "transmission trail" that can be followed.

Whenever one asset sends a message to another, a line from the asset sending the message to the asset receiving the message is drawn. The endpoints of the line identify when the message leaves or arrives at an asset thus the slope of this line is indicative of the time required to transmit and consume the message. For example, a steep slope with respect to the slopes of other transmission lines indicates that the message arrived at its destination and was consumed relatively quickly. A shallow slope indicates that the message may have take a longer time to arrive at its destination or that it spent a great deal of time waiting in the recipient asset's input queue. The horizontal portions of a message line centered on the asset time-lines are indicative of the amount of time the asset requires to process the message.

4.3.1 Special Features of the Transaction Time-line View

Because there are potentially thousands of messages being sent during the execution of a program, the display could easily become overpopulated with the message lines. To combat this problem, there are several provisions built into this view:

Color Coding of Messages

An attempt is made to represent each message line with a unique color. This is not always possible if there are more messages than colors available (due to limitations in the computer display and human perceptions). If the display exhausts the colors available, the color allocation cycle will begin again and colors will be reused. Under normal circumstances, messages lines that share a color will have been separated by enough time as to remove any ambiguities that may occur. Although a color display works best, the colors assigned to a message are cycled so that there is enough contrast for messages to be traced on both grayscale and monochrome monitors. If the colored lines prove to be a problem, one of the *asset focus* features may be required.

Focus Lists

The *Message* focus holds a listing of all messages sent during the execution of the program and is a hilighting mechanism that allows one message to be singled out from a larger cluster of messages. Selecting a message tag from the focus list instantly highlights the message on the display for easier recognition, regardless of the type of display being used.

The *Asset* focus list contains a list of all assets that are represented on the Transaction Summary view. Selecting an asset, or group of assets, from the list *excludes* all messages that are sent from the selection. This mechanism is in place to reduce the number of message lines displayed, a potential problem when the user is faced with large numbers of assets or messages.

Axis Scaling

With the **Time** and **Asset Line Spacing** sliders it is possible to change the scale of the horizontal (time) axis and vertical (asset spacing) axis. Should events happen too rapidly and not appear as distinct occurrences on the display, the time axis can be scaled thus "stretching" the display. If the scope of the display is too small and not all assets are showing, the asset spacing scale can be adjusted to narrow the gap between the asset lines thereby allowing more assets to be displayed at one time. Scaling both the time and asset spacing axis compresses the entire display into a small area thus allowing entire an run (or as much that will fit in the window) to be represented on the graphical display.

4.4 The Transaction Summary View

The Transaction Summary view, shown in Figure 4.5, is used to trace Enterprise messages sent and received by assets during a program's execution. It is possible to find and follow the trail of an individual message in order to determine whether or not the message is transmitted without problem and processed without difficulty.

This Transaction Summary View is divided into two components: the *Message Summary* region which presents the history of a *single* message, and the *Global Summary* region which characterizes the history of *all* messages sent during a program's execution.

4.4.1 The Message Summary region

The constituent components of the Message Summary are:

Tag List a selectable index of all messages transmitted,

Tag the message currently selected from the tag list,

Origin the name of the asset who originated the message,

Start the time when the message was originated,

End the time when the message was removed from circulation and

Detailed History a table of data that traces the complete transmission path of the message.

Whenever a message is selected from the tag list, the fields of this region are updated to reflect the values of the selected message.

The *Message Processing History* area represents the transmission and processing times experienced by a message using a graphical format similar in form to the Asset Summary diagrams found in the Asset Utilization View. The time a message is subject to a given operation is shown graphically as wedges of a pie-chart graph. This chart is divided into three sections:

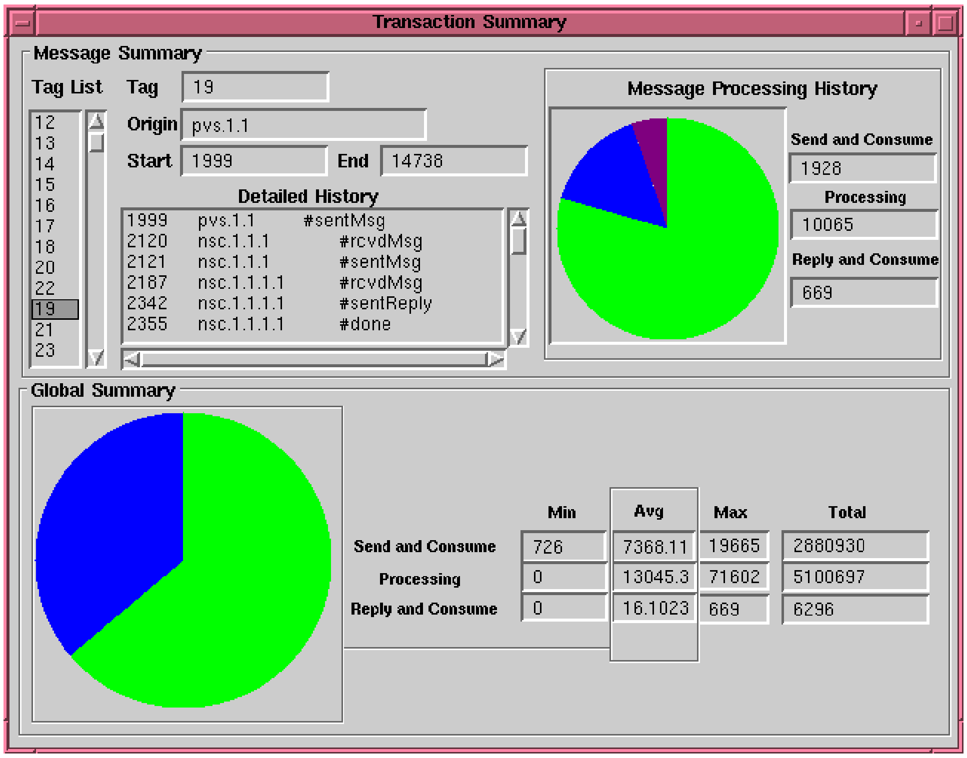

Figure 4.5: The Transaction Summary view.

Send and Consume the time a message spends and waiting in an input queue,

Processing the time a message spends being processed at an asset and

Reply and Consume the time a message spends in transit after it is issued as a reply to an earlier asset call and the time waiting in a reply queue.

On colored displays, the Send and Consume wedge appears blue, the Processing wedge appears blue, and the Reply and Consume wedge appears magenta. The transit times include the amount of time the message spends waiting to be processed in the recipient asset's input queue (the consume time). Thus the message processing time begins when a message is removed from an asset's input queue and ends when a message is either forwarded to another asset or is sent in reply to a work request.

4.4.2 Global Summary

The Global Summary region consists of a graphical display and data table of *global* message statistics ("global" refers to the fact that *all* messages are accounted for when generating the statistical data contained in the table). The table of values records the minimum, maximum and average transmission and processing times of all messages. As with the Message Processing History of the Message Summary region, consumption times are absorbed into the send and reply times of the message. The graphical display represents the statistical average values recorded in the data table of this region. The color mapping of the wedges is the same as that of the Message Processing area of the Message Summary region.

4.5 The Performance Annotation View

The Performance Annotation View (shown in Figure 4.6) is a high-level, interactive tool which is used to automatically detect special conditions or events that may occur during the execution of an

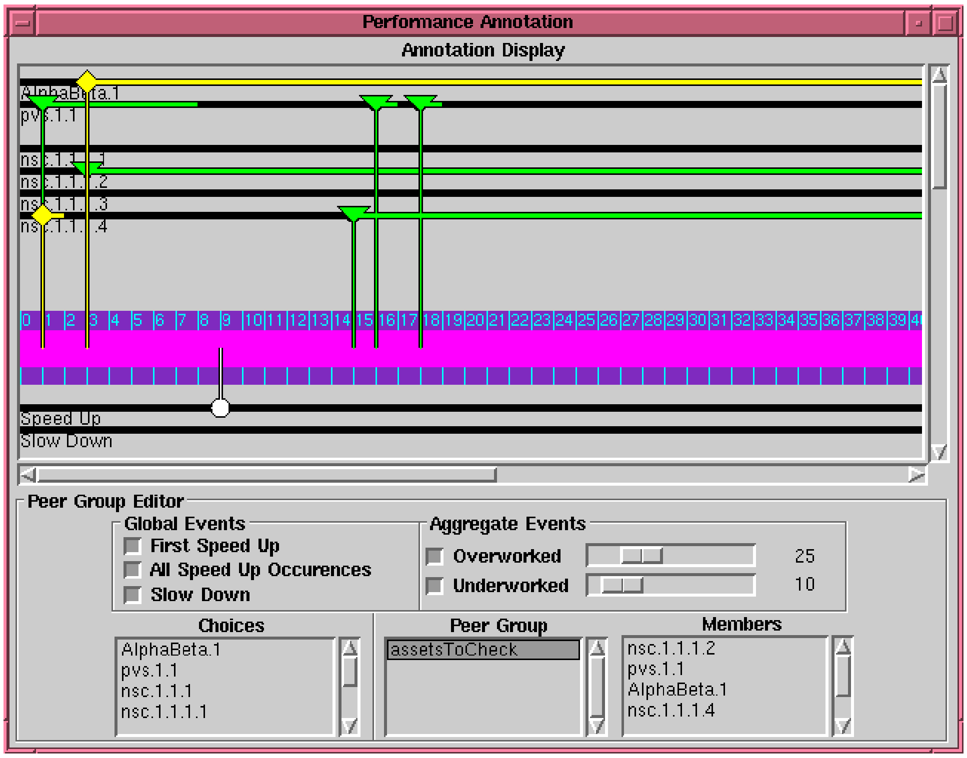

Figure 4.6: The Performance Annotation View.

Enterprise program. This view is divided into two regions: the *Annotation Display* and the *Peer Group Editor*.

4.5.1 The Peer Group Editor

The Peer Group Editor is used to define and configure subsets of assets, known as *peer groups*, to be examined as the program is executing. The editor consists of three membership identification and modification lists and an event type checklist. The three membership lists are:

Choices a list of all assets (including managers and replicas),

Peer Group a list of all defined peer groups, and

Members a list of all the members of a selected peer group.

Peer groups are defined, modified and saved through use of context sensitive menus attached to each of the three lists.

The event checklist offers two categories of events to choose from: *Global Events* and *Aggregate Events*. The Global event types are:

first speed-up when the program first experiences a speed-up,

slow-down when a program goes from a speed-up to a slow-down,

all speed-up to indicate all times a program experiences a speed-up after a slow-down.

Global events encompass *all* Enterprise assets with all assets partaking in the analysis procedure. The user does not need to create a peer group if only global events are sought.

Aggregate events type are found by examining the members of a peer group as defined by an Enterprise user. If more than one group is defined, each peer group of assets is examined in turn to determine whether or not the aggregate event types are:

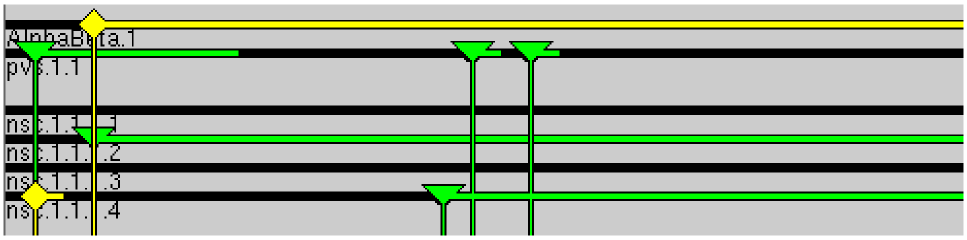

Figure 4.7: The aggregate lines of the example Performance Annotation View.

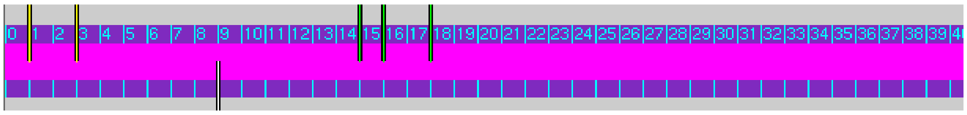

Figure 4.8: The main time-line of the example Performance Annotation View.

asset overworked when asset(s) of a peer group are doing too much work and

asset underworked when asset(s) of a peer group are not doing enough work.

Aggregate events are of special interest because they are based upon the comparison of a peer group member's usage statistics against the *aggregate* value of its peer's usage statistics. This allows the Performance Annotation View's internal analysis mechanism to determine whether or not a single asset's utilization characteristics falls outside the norm of the group. If the asset's statistics are found to differ significantly from its peers, a problem potentially exists and the user is notified in the form of a graphical annotation.

For such a scheme to work, there must be a method by which a single asset's data can be compared against its group's aggregate data. This is accomplished by using the peer group's average usage statistics (as calculated by the Performance Annotation View's analysis mechanism) and a *variation* value supplied by the user. This variance value signifies to the analysis mechanism how far outside the group average (norm) an asset's utilization value must fall before an event condition is satisfied. For example, suppose a variance of 25% is supplied to an overworked event type for a peer group named **Alpha**. This signals to the Performance Annotation View that should any member of **Alpha's** busy time be 25% (or more) greater than **Alpha's** aggregate average, the member is to be considered overworked and must be annotated on the display.

Any number of peer groups is allowed and any Enterprise asset can belong to one or more peer groups. The only stipulation to the creation of a peer group is that its name must be unique.

4.5.2 The Annotation Display

The Annotation Display is subdivided into three regions. The first region contains the *aggregate event time-lines* used to display the annotated events. There is one aggregate line for each asset of the program being examined and it appears as a named time-line in the upper portion of the display. The middle region of the display is made up of the *time-line gauge* (divided into one second intervals), and the *execution time-line* (showing elapsed program execution time). The lower region displays *global event time-lines*. Figures 4.7 through 4.9 show these three regions for the Annotation Display of Figure 4.6.

Figure 4.9: The global lines of the example Performance Annotation View.

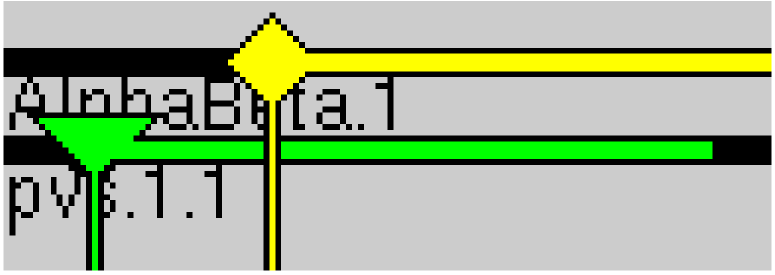

Figure 4.10: An enlargement of two annotated events.

To prevent the display from becoming overly crowded, only the aggregate lines of assets belonging to the currently selected peer group are shown on the display. Depending on the assignment of assets to a peer group, large gaps between the aggregate lines may sometimes form. An example of this "gap" can be found in Figure 4.7 between assets **pvs.1.1** and **nsc.1.1.1.1**. These gaps correspond to the aggregate lines of assets which are not members of the peer group. This tactic prevents abrupt shifts in the display layout when switching between peer groups and provides a degree of consistency in the display.

An annotated, aggregate event appears as a shaped icon on the event line of the asset at which the event was found to occur. Likewise, an annotated global event appears as a dot on the applicable global event time-line of the display. In either case, a line from the icon to a point on the execution time line is drawn to help identify when the event occured. If the annotated event is one that persists over time, a second *duration line* is drawn horizontally from the icon stopping at the point in time when the event condition ceases to be true. An overworked asset annotation icon is shown as an inverted triangle and appears green on colored displays; an underworked asset annotation is shown as a diamond and appears yellow on colored displays. Global annotation events appear as white circles. An enlargement of an annotated event (the dot, the drop line and the duration line) for asset **pvs.1.1** is shown in figure 4.10.

4.5.3 Using the Performance Annotation View

The steps that must be followed in order to use the Performance Annotation View are:

1. Define asset groups which are to be examined as the program runs.
2. Select events of interest from the list of event types,
3. Save the parameters for the asset group.
4. Run the program.

As the program is running, the Performance Annotation View monitors each of the asset groups for occurrences of the interesting events. If such an occurrence is found, the event is recorded and the Annotation Display is updated.

Chapter 5

Performance Data Acquisition and Analysis in Enterprise

5.1 Overview

The expansion of Enterprise to include performance monitoring tools is not a task carried out in isolation. It is the next step in the evolution of a system that already boasts an impressive data acquisition mechanism and state-of-the-art presentation system. This chapter addresses the issues involved in the design, implementation and integration of the performance analysis tool-kit into the Enterprise system and discusses some ways in which the tool-kit distinguishes itself from other visualization tools. Included in this discussion is a description of how Enterprise carries out the first two stages of performance analysis, acquisition and analysis, and serves to delineate where "the old Enterprise" and "the new Enterprise" meet.

5.2 The Legacy: Foundations of Performance Analysis in Enterprise

Before the addition of the performance monitoring facilities to Enterprise, there were two existing post-mortem processing subsystems present: the *animation view* and the *replay view*. The animation view allows a user to visualize the execution of their parallel program by using an animated display to visually represent the messages exchanged between assets during the execution (the information displayed is derived from a trace file generated during the program's execution). However, the animation view is more than just a picture window of an executing program. It features an interactive component which provides the user with a great deal of control over the progression of the animation. If desired, the user can halt the animation, proceed in a stepwise fashion through the review, and even alter the rate at which the animation progresses. The second subsystem, the replay view, is similar in form and function to the animation view but it differs in one important respect: the replay view forces a controlled re-execution of the parallel program. When program replay is selected, the interface re-launches the Enterprise assets and forces the re-execution to match events found in the trace file. By re-running the program in a controlled manner, the user is able to open debugger tools to help correct program errors. Like the animation view, the user is given the same control over how the animation and replay progresses.

One drawback to both the animation and replay views is that they are strictly "one-way" post-mortem systems. The user cannot selectively animate or replay portions of a program trace. To witness the last few seconds of a program's execution, the user must sit through the trace for the entire program. However, a greater criticism of the animation and replay systems of Enterprise is that they fail to preserve the relative timings of events. Logged events are extracted from the trace file and processed at a constant rate. Thus the time taken to process and animate events that occurred one milli-second apart *is exactly the same as it would be for events that occurred one hour apart.* Alone, the post-mortem views offer a misleading view of the program's performance. This situation is remedied by the performance monitors added to Enterprise.

5.2.1 Enterprise Events

The performance data relayed to the interface consists of special report messages called *Enterprise events*, or simply *events*, which are generated by each asset as the program is running. These events are simple, plain text messages which contain summary information about the action an asset performed. These events are sent to the root process for further handling. If post-mortem analysis is to be performed, the root process writes the event to the program trace log file. If real-time performance analysis is to occur, the root process forwards the event to the interface for further processing and analysis. Real-time performance analysis occurs concurrently with program execution.

An Enterprise event structure consists of the following fields:

type the type of event which occurred and is being reported,

primary asset indicating at which asset the event occurred and who is doing the reporting,

secondary asset indicating who sent the message to the primary asset or who is the intended recipient of a message sent by the primary asset,

tag the unique number that indexes the message and,

time indicating when the report message was generated.

The **time** field contains the timestamp applied by each asset which is based on an offset from the execution start-time of the program. Before execution begins, the run-time executor sends a synchronization pulse to each of the workstations telling the processor what to consider *time zero.* All subsequent asset time references are based upon this start time. Time resolution in Enterprise is measured in milli-seconds.

Events are generated each time an asset undergoes a state change, an occurrence that happens whenever an asset has sent or received a message. The type of event signifies the operation that has occurred and the state transition an asset undergoes. Valid event types are:

sentMsg the primary asset has sent a message,

rcvdMsg the primary has received a message,

sentReply the primary has sent a reply to a message,

rcvdReply the primary has received a reply to a message,

doneMsg the primary indicates that it has finished a task,

block the primary signals that it is awaiting a reply to a particular message or

die the primary asset has finished all of its work and is now dead

The types and their interaction are shown in Figure 5.1 [SG93]. As an example of how the state transition scheme works, consider an asset named **SampleAsset** that is initially **idle**. On receiving a message (work), **SampleAsset** goes from the **idle** to **busy** state and issues a **rcvdMsg** typed message to the root process. On completion of its task, **SampleAsset** will send a reply to its calling asset and issue a **sentReply** typed message to the root process as it returns to an **idle** state. The event message and its format are a legacy of the work done by others [SSLI93].

This particular method of generating events is interesting because it is an elegant method of ensuring that an analysis system is able to derive useful information about the execution of a program. By knowing both the exact instant a state change occurs and the type of the state

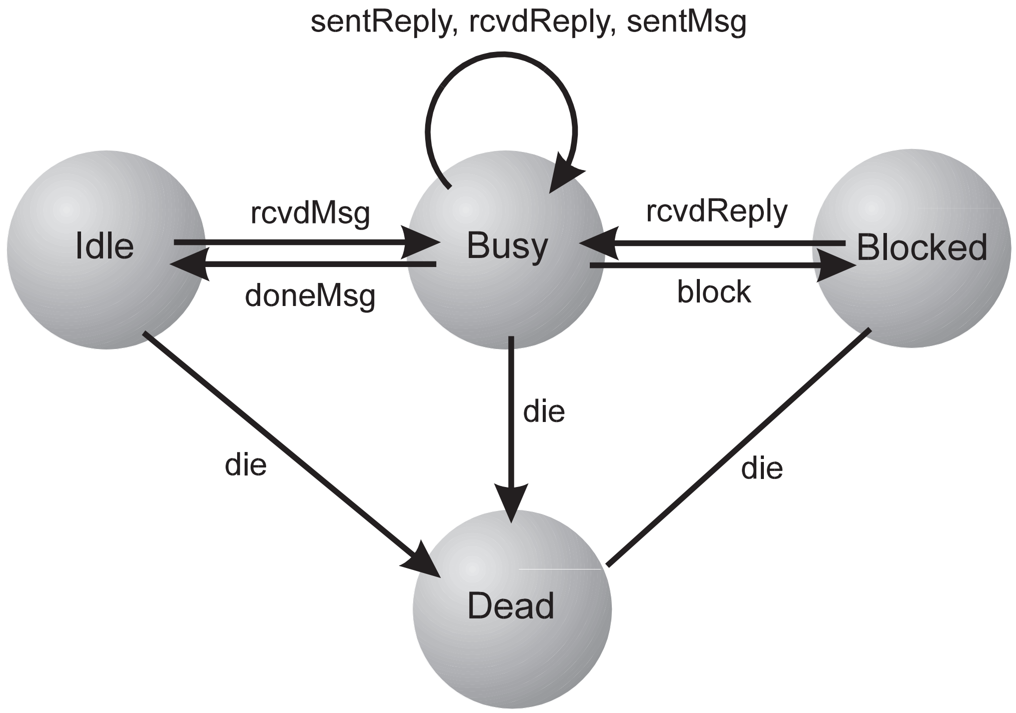

Figure 5.1: The state transition diagram of Enterprise.

change, an asset's usage characteristics can be extracted. By realizing that both the send time and receive time of a message is logged, it is possible to derive in-transit times for Enterprise messages. The importance of these quantities cannot be ignored. In fact, the key to performance analysis in Enterprise is found in these quantities extracted from the trace file.

5.3 The History of Performance Analysis in Enterprise

The addition of performance analysis tools into Enterprise occured in two phases. The first phase required performance analysis capabilities be added only to the post-mortem animation subsystem. Events, as they were extracted from the trace file for animation, were processed by the interface to determine their effects on the assets and the information was recorded. This phase also saw the creation of several visualization displays which would later evolve into the Asset Utilization Utilization View and the Performance Annotation view (the Transaction and Summaries views were created during the second developmental phase). The visualization displays were updated after each event processed. The second phase was to extend performance analysis operations to occur in real-time and saw fundamental changes in the performance system.

Real-time analysis requires that the processing rate of events match their arrival rate at the interface making the time required to process events a critical factor. The original analysis system used a decentralized approach whereby each visualization display was responsible for calculating the performance statistics which it displayed. This strategy was acceptable for post-mortem analysis, but has two serious problems when trying to extend it to include real-time analysis. First, there was only a single execution thread in the interface. The procedure of accepting an event, calculating its effects, and updating the display must be carried out sequentially. This introduces much overhead into the interface making it unable to process events in a timely manner thus rendering real-time analysis ineffective. Second, the decentralized approach sees the visualization tools repeating some of the work required for the analysis process and thus wasting limited computation time. The new performance analysis system uses a more centralized approach and employs a multi-threaded execution stream. Work that was common to the visualization tools was abstracted out into a *performance manager* system which is now responsible for carrying out analysis functions and storing the results. Because the performance system has divorced itself from the interface, it can be treated as a separate entity. This makes it possible to be launched as a separate process. The interface accepting events and the event manager processing events run parallel to each other thereby increasing the rate of event processing.

The new performance analysis system, and the issues introduced here, are discussed in more detail later in this chapter.

5.4 Performance Analysis in Enterprise

There are three steps involved in the performance analysis of an Enterprise program:

1. Event preprocessing
2. Data Analysis
3. Data Presentation

These three operations correspond roughly to the three phases of performance analysis discussed in Chapter 1. The following sections discuss the implementation issues associated with each.

5.4.1 Raw Event Preprocessing

During execution, the root process feeds the interface events it is gathering from each of the Enterprise assets. However, before an event can be analyzed, it must first be preprocessed into a form palatable to the interface. The incoming event arriving off the communication pipe is a simple byte stream which, in its current form, is unusable by the performance analysis system. The stream must first be parsed in order to obtain a *raw event*, the interface's internal representation of an Enterprise event. These raw events are then stored into *two* data storage structures within the interface. The first structure stores events into an array indexed by the message tag field, the second stores events into an intermediate structure sorted by the message timestamp.

Although the use of dual storage structures introduces redundancy into the system it is a necessary optimization. There are two disjoint points of view from which to examine a running Enterprise program: the *message's* viewpoint and the *asset's* viewpoint. From the asset's perspective, quantities such as the number of messages processed, the total amount of time spent in each of the execution states, or how much work it does compared to other assets are the quantities of interest. In this case, the cumulative *effect* a message has on assets is more important than the message itself. The alternative approach is to view a program's execution from a *message's* perspective as it travels between assets. A message sent from one asset to another is considered to be on a voyage. From this point of view, notable quantities would include the time spent in transit, the time spent waiting in queues and the asset path it takes. The *journey* a message takes is more important than it's consequences.

Both the asset and message perspectives of an executing program offer useful insights and although one perspective could be derived from the other, it is not economical to do so. To reproduce the program trace for a message from an asset's perspective requires that each asset be searched for all encounters with a given message. To reproduce the asset's perspective from a message's point of view would require the re-processing of each message by retracing its path and deriving the effect it has on the assets it encounters. The repetition of work is a computationally wasteful operation, especially if it is to be undertaken in real-time where processing cycles are scarce. After all is said and done, it is simply more cost effective to maintain two distinct lists.

Processing Lag and Chaotic Processing

Under ideal conditions, a message sent over a network experiences delays that are proportional only to the propagation delay of the network. Unfortunately, ideal conditions seldom occur. Message collisions, transmission errors and other such problems will often result in messages arriving later than expected. It is not uncommon for an event message to arrive at the interface in an unordered fashion producing what appears to be a tachyon. From a post-mortem standpoint, this is not a serious problem as events can be sorted once execution has been completed. However, from a real-time standpoint, unordered messages pose a serious problem. Unordered messages pulled from the communication pipe and processed directly make tachyons possible.

To compensate for unordered events, the performance monitoring system introduces an artificial *processing lag* into the analysis process. As noted earlier, incoming events are placed into an intermediate structure which sorts events by their timestamp. The "lag" introduced is a directive issued to the performance monitor to delay the processing of events by the amount of time specified by the lag factor. That is, events will be buffered for a time no less than the lag factor before they are processed. Thus events are given a "window of opportunity" to arrive late. If an event is misordered and the difference between when the asset *does* arrive and when it *should* have arrived is less than the lag factor, the event can be put into its proper place in the event buffer and processed normally. A misordered event that arrives outside the window may have missed its chance to take its proper place in the buffer because its peers may have have already been processed. In this case, the event will still be processed but because it is out-of-order it will lead to erroneous results. If such an occurrence should happen during real-time analysis, the performance monitor will issue a warning to the user but will continue to process the events as though all were well.

It should be noted that the lag is purely an operational feature to enhance analysis and not part of the analysis process itself. It is used to ensure that analysis is performed at an optimal rate given the environment in which to program is executing. The task of determining the best lag factor is still more of an art than a science and relies mainly on the intuition and judgement of the user.

There has been much said of "lag times" and "sample times" but little said about what "time" actually means in Enterprise. Time intervals are based on the time passage of an internal "virtual clock", not on the system clock or other such "wall clock" times. Virtual clock time always passes at a constant rate however, unlike the conventionally accepted notice of a clock, the *rate* at which virtual time passes is *variable*. This may seem unusual at first, but one must take into account that the rate at which raw events arrive is erratic and that the rate which buffered events can be processed is dependent on any number of external factors, all of which make using a system clock unreliable.

Chaos processing, a special option available to the user, directs the performance monitor to process events as they arrive without using the "window buffering" strategy outlined earlier. By selecting this option, the user indicates to Enterprise that a reliable network is present and that the ordering of events is not in question. The purpose of such an option is to eliminate the overhead associated with maintaining buffers and is meant to speed up the event processing should event speed become an issue.

5.4.2 Data Analysis

Data analysis in Enterprise is carried out by a collective entity known as the *performance manager* with the *event handler* being its primary component. The other components of the performance manager are the control panel, the performance data storage structures, the visualization tools described in Chapters 3 and 4, and the event buffer. Of all the components contained in the collective, the user interacts only directly with the control panel and the visualization tools. The event handler remains alive and active so long as the program is executing or there are events waiting to be processed. Once the program ends and the event buffer is drained, the event handler is starved and finally dies leaving the other components for post-mortem examination.

The event handler is a separate processing thread launched at run-time by the interface and is the "workhorse" of the performance manager. It is responsible for sampling the event buffer, processing events, deriving statistical information, and updating the visualization displays as the program is running. It works in conjunction with the user interface which is responsible for accepting incoming events and placing them into the event buffer. The two duties, greeting incoming events and processing events, are divided into two processes for optimization reasons. While buffered events are being processed by the event handler, new events are being preprocessed in parallel by the interface. Because the incoming events do not have to wait for the event handler's attention, event processing throughput is high. The cost of this operation is the added complexities that arise when two systems access the same data structure simultaneously.

An interesting feature of the event handler is that it is a dynamic entity. It examines the event buffers and determines how event handling is to be done. If the event buffer becomes overly full, the event handler increases the event processing rate. If the buffer is under filled, the event handler will slow the processing rate giving the buffer a chance to be replenished. This "slowing down" and "speeding up" is accomplished by changing the event handler's internal definition of "one second." Under normal circumstances, the performance manager defines one second to be equivalent to one "wall-clock" second. However, if processing is proceeding too quickly, the event handler may change its definition of one second to be equivalent to two "wall-clock" seconds thus the processing rate is halved. Alternatively, consider when the "internal" second is defined to be three "wall-clock" seconds and the event buffer is overflowing. In this case the event handler could restore the internal second to be equal to a wall-clock second thereby tripling the event processing rate. The only rule that must be followed is that the the passage of time is constant.

Event Sampling and Performance Snapshots

At regular intervals during the execution of the program, the performance manager gathers the state information from each of the assets as part of the sampling process. At the start of each new sample interval, all events that fall into that interval are moved to a temporary buffer before they are processed. This partitioning is necessary because the event buffer is being accessed by two entities, the event handler and the interface. Partitioning of events ensures that only the events found in the current sample interval are processed to determine their effects on system performance information. The remaining events found in the event buffer wait for their sample interval to occur. Because the events are already sorted by their timestamp, a sequential search beginning at the start of the list is sufficient to determine which events should be moved to the temporary buffer.

Once the event partitioning is complete, the event handler inspects each event contained in the temporary buffer and identifies the primary asset, the state changes that occur and proceeds to update the asset's state information (each asset's information is kept in an *asset performance* data structure which is indexed by the name of the asset). After the temporary buffer is drained, each asset's performance statistics (busy time, idle time, blocked time) are updated. The performance manager then builds the overall performance statistics for the interval and creates a "snapshot" of the system for the sample interval. The snapshot is then placed into an array which can be searched by each of the visualization tools at a later time. Maintaining the snapshots in the array allows the performance history to be examined at the user's leisure. The visualization views interpret the snapshots in their own way and present the data in a form that is dictated by their function.

5.4.3 Data Presentation and the Visualization Views

Each visualization view, from an implementation point of view, is a distinct and separate module that knows how to manage itself. That is, each visualization tool module contains the code needed to display its data, manage update requests, and process user supplied directives. For example, the Asset Utilization view, after receiving an update notice, will obtain the latest asset utilization statistics from the performance manager and update its graphical display accordingly. Each visualization tool is completely independent of the others meaning that changes, updates and other modifications to one tool will have no effect on the functionality of the other tools. This also means that the user can use each tool in isolation or combine the views in various manners to provide different insights into the program.

5.5 Annotation Directed Analysis and the Performance Annotation View

In addition to the statistical analysis outlined in the previous section, a more sophisticated analysis can be requested by the user through the *Performance Annotation View* mentioned in the previous chapters. The Performance Annotation View makes use of the same data and snapshots outlined earlier, but it does additional checks to determine whether or not special conditions that are of particular interest have occurred. If one of these interesting conditions is met, the annotation View creates a visual annotation on a program time-line that indicates when the condition occurred, the assets involved and the type of condition that was met.

Conditions that encompass the entire program and all its assets are known as *universal events.* Such conditions include:

- time of the first true speed up,
- time(s) when a program experiences a slow down, and
- time(s) when a program experiences any speed-ups

A second category, *Aggregate Events* are those which are discovered, or derived, from the examination of a subset of Enterprise assets known as a *peer group.* At every sample interval, each peer group is examined to derive the performance characteristics for the group. Each asset of the group is compared against the group's norm. Should some aspect of the asset's performance fall too far outside the norm, a warning is generated at the time of the occurrence and the event is displayed on the Performance Annotation View's display. Multiple peer groups are allowed, each with its own event specifications possible. This means the user can look for different conditions in different peer groups. Aggregate events of note include:

- assets who become overworked,
- assets who become underworked,
- assets who spend too much time in a blocked state,
- assets who process too many messages, and
- assets who process too few messages.

The interesting feature of aggregate events is that they are derived from the ability to single out an asset that meets or fails a given criteria from a set of assets. The key to this procedure is the use of *thresholds and variations* when doing the analysis. Prior to run-time, the user supplies a threshold for each aggregate event type selected. When a peer group's statistics are generated, the average over all assets is found. The statistics of each member of the peer group is then compared against the group average. If the differences between the member matched against the group norm differs by more than the threshold value, the asset is flagged and annotated. To prevent an already flagged asset from being singled out for the same problem repeatedly, the asset is placed on a list where it remains until the condition is no longer met. This appears as the "drag" line on the display that starts from the time the event occurred.

5.5.1 Performance Analysis in Enterprise versus Other Analysis Systems

The most prominent difference between the Performance Analysis system of Enterprise and other such systems, most notably ParaGraph[1], is that it is intimately tied to the Enterprise model and its analogy of worker "assets." ParaGraph is a general purpose visualization tool which requires that trace logs be in the PICL format. This generality makes ParaGraph suitable for a wide variety

[1] ParaGraph is perhaps the most advanced tool and is used to gauge other such tools in this document.

of applications. Conversely, Enterprise uses its own format for trace data[2] and utilizes an internal analysis system for processing and presenting information. This is not to say that the Enterprise performance analysis modules could not be adapted or generalized to suit other systems. However, the time and expense of separating the performance analysis system from the Enterprise model and adapting it to some unnamed system would be a non-trivial, time-consuming task.

Because the Enterprise analysis system is so closely associated with the Enterprise model, it is possible to do the analysis in real-time, a feat not possible in ParaGraph, a post-mortem analysis and visualization system. Unlike Enterprise, ParaGraph is for visualization and analysis only; data acquisition is beyond its responsibility. Enterprise gathers events, processes data and displays the information at a speed comparable to match the program execution. If the Enterprise analysis system cannot pace the program execution, it will start to trail the program's execution but still makes the attempt to process data in real-time so that the difference between event generation and event display is minimized. ParaGraph takes trace data and processes it as quickly as possible, but because the trace data is post-processed, the problem of unordered events can be easily handled.

Finally, the Performance Annotation tool introduced here is unique to visualization and analysis tools. It is a very high-level approach to monitoring a program's execution in real-time by annotating specific events of interest on a graphical display. The automated process identifies specific problems and when they occurred in an attempt to assist the user *interpret* the information. This method does not take the onus off the user who makes the judgement call, but serves as a pointer to things that fall outside the desired norm.

[2] The "legacy" of Enterprise described earlier.

Chapter 6

Summary

6.1 Thesis Summary

To understand the need for parallel performance analysis and debugging, one must look back to the reason that a parallel solution was first sought. A person has a problem to solve and requires the use of a computer to do so. Some problems are so large, or so complex, that even the fastest computers take days, weeks, months, or even years to solve the problem. Unfortunately, the person can't wait that long (however long that really is) and so they consider a parallel scheme hoping that the problem can be solved in a shorter time. So the person writes a parallel program to solve the problem and determines that it can be solved in three-fifths the time a sequential version would take. Overjoyed with such rapid success, the programmer may now be confident enough to try to do better and writes a new version of the program and finds it does no better than the first. So they try again, and again, and again. What the user does not know is that the first version made the best possible use of the available computing power and thus any further modifications would produce no better results. This, in a nutshell, demonstrates why performance analysis is needed.

A parallel programmer seeking the best possible solution to a problem requires the means and methodology to determine what "best" actually means. A naive approach would be to limit the judging process to the issue of program speed, but that ignores many other contributing factors. Is the program easy to write and maintain? Is it easy to use? How long did the program take to develop? And perhaps most importantly, is the program using its resources wisely? It is possible to write a parallel program that runs on ten processors but only experiences a 2 fold speed-up. Based on speed alone, this is a victory. Based on common sense, something appears to be amiss. However, common sense is not evidence. Only by analyzing how the program performs will enough quantifiable evidence be gained to pass judgement. Only then will the user have the means to determine when the "best" program is found.

Although performance analysis of a parallel program is a desired goal, it is not a simple task. The programmer must be able to gather performance data without adversely affecting how well their program performs. This data then has to be transformed into meaningful quantities and presented in a clear and understandable manner. There are software tools and hardware systems that performs such operations, but each has its own limitations. Some solutions are too individualized to be useful. Some suffer from exotic hardware requirements. Some tools can do the analysis only after the program has run to completion.

One particular system, the Enterprise Parallel Programming Environment attempts to integrate a suite of tools into a comprehensive, easy-to-use environment. In addition to facilities needed to develop and debug parallel programs, Enterprise also features tools used to gather program trace data and to visualize a program run. However, the analysis mechanisms were strictly post-mortem and lacked any true performance monitoring features. The performance monitoring void prompted the desire to add such performance monitors into Enterprise, leading to the work contained in this document.

The addition of the performance analysis facilities made use of and expanded upon the existing data acquisition mechanisms of Enterprise. The Enterprise root process was responsible for the coordination of program trace events received from each of the Enterprise assets. Each status message sent to the root process by an asset was forwarded to the Enterprise interface where it was readied for the performance analysis procedures. The preprocessing at the interface was necessary to extract the event data and ensure that events that may have been received out of order take their proper place for the analysis phase. Once the analysis was complete, a snapshot which characterizes the programs performance was taken and stored for reference by the four Enterprise visualization tools. The visualization tools would then be used to trace any performance errors and bottlenecks.

6.2 Conclusions

In the course of adding the performance tools to Enterprise several conclusions were drawn.

Load balance is the key

In trade for the ease-of-use, the Enterprise user accepts a programming model that is strict in its requirements. By writing and executing an Enterprise program, the user has limited the number of options with which they can directly influence execution of a program: changing the program graph, using asset replication, modifying user source code and, to a limited degree, choosing which machines are used to run the program.

The user has no control over the communication manager and how communication between processes is accomplished. Likewise, the user has no control over the format of messages being sent. This is the benefit of using Enterprise: the removal of the small, aggravating details required to manage a parallel program.

The practical benefit with respect to performance is that the casual Enterprise user need only be interested in performance at the asset level. Understanding how an individual asset perform provides the user with enough information to modify a program as was covered in Chapters 3 and 4. A user who is trying to optimize the program at lower levels is stepping outside the scope covered by the Enterprise model and thus beyond the capabilities of the Enterprise performance monitors.

The key to the performance problem is to understand that nearly all problems (excluding chance occurrences) can be corrected by modifying the amount of work an asset processes. For example, an over-worked individual can be replicated thus redistributing the workload. Problems due to passing messages over the network can often be corrected by changing the granularity of the problem, a task accomplished by modifying the workload characteristics of an asset. Machine load problems are evident by determining how much work an asset is or isn't doing. Thus a user must concentrate on a solution that optimizes the work done by each asset.

Simple is Best

The user does not need to be bombarded with complex displays containing vast quantities of data in order to make informed decisions. The visualization displays of Enterprise are purposely made as simple as possible. For the most part, simple graphical constructs that relay information is all that is needed as indicated in Chapters 3 and 4.

Abstraction is good

The original implementation of the performance analysis tools saw much of the performance manager's responsibility resting within each of the individual visualization tools. The event manager served as a convenient mounting point for the visualization tools and control panel and fulfilled a role as a data storage facility. This resulted in visualization displays that were self-contained and highly optimized but performed redundant operations amongst themselves. From a developer's perspective, the tools are flexible and could be modified individually without impacting the other visualization tools. However, the addition of a new visualization tool required a great deal of time and effort and introduced even more redundancy into the system.

The original rationale for the distributed approach was that each visualization tool has its own unique requirements and would be best served if it did its own work. The performance manager did a more involved preprocessing of events which were then fed to each of the visualization tools who would take the data and do the analysis as they saw fit. Had the analysis been limited to a post-mortem procedure, this method may have proved sufficient. However, because the analysis was extended to include real-time operations, the dividing of responsibilities became cumbersome. Each tool had to perform its own analysis phase of events and even though this procedure was optimized at the tool level, it still required each tool to do a substantial amount work. The analysis time was thus proportional to the number of tools used, an unacceptable scenario.

The new version of the tools saw the common functionality of the visualization tools abstracted out into the performance manager. This provides a great deal of generality in that all the tools now have a single source of data. Each tool is thus *interpreting* a single data stream. This makes the task of adding a new display easy: write a new routine to interpret the performance data. The benefits are decreased analysis time and greater generality.

The one criticism that could be made is that perhaps the analysis is too general and that not all facets may be covered adequately. However, as outlined above, the problem of program performance hinges on how well a user divides work amongst the processors. The data provided by the analysis tools are more that sufficient for the users needs.

Real-time analysis is possible in Enterprise

Enterprise, and similar systems which use high-level abstractions of parallelism offer a unique opportunity to incorporate real-time analysis facilities into their environment. Enterprise uses asset level parallelism to achieve concurrency and is the abstraction mechanism which allows a novice parallel programmer to write parallel programs quickly and painlessly. It is this same abstraction mechanism which makes real-time analysis possible. In essence, the abstraction simplifies the task of parallelizing the program by imposing restrictions of what the user can and cannot do. This in turn simplifies the type and quantity of performance information which a programmer may find useful. For example, it is possible (with some effort) to re-tool the visualization views to report the number of megaflops, memory cache-hits and other such "low level" quantities, but this information falls outside the "Enterprise" model and defeats the spirit of Enterprise, a simple to use *accessible* system.

The Enterprise programmer uses a high level tool to write programs. Why shouldn't they use a high level tool to analyze these same programs? Limiting the scope of the analysis performed is generally considered a poor solution, but because the model already imposes some restrictions on what the user does, limiting the scope to fall within the restrictions is therefore acceptable. By reducing and simplifying the analysis procedure it then becomes possible to perform the functions quickly enough to match the real-time execution of the program.

Bibliography

[BB93] Thomas Bemmerl and Peter Braun. Visualization of Message Passing Parallel Programs with the Topsys Parallel Programming Environment. *Journal of Parallel and Distributed Computing*, 18:118–128, 1993.

[BDGS93] Adam Beguelin, Jack Dongarra, Al Geist, and Vaidy Sunderam. Visualization and Debugging in a Heterogeneous Environment. *COMPUTER magazine*, pages 88–95, June 1993.

[DD95] Michael A. Driscoll and W. Robert Daasch. Accurate Predictions of Parallel Program Execution Time. *Journal of Parallel and Distributed Computing*, 25(1):43–64, February 1995.

[FJ93] Joan M Francioni and Jay Alan Jackson. Breaking the Silence: Auralization of Parallel Program Behavior. *Journal of Parallel and Distributed Computing*, 18:179–194, 1993.

[HE91] Michael T. Heath and Jennifer A. Ethridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, pages 29–39, September 1991.

[Igl94] Paul Iglinski. An Execution Replay Facility and Event-Based Debugger for the Enterprise Parallel Programming System. Master's thesis, University of Alberta, 1994.

[IMM+95] P. Iglinski, S. MacDonald, C. Morrow, I. Parsons, J. Schaeffer, D. Szafron, D. Woloschuk, and D. Novillo. Enterprise User's Manual Version 2.4. Technical Report 9502, University of Alberta, April 1995.

[Inc92] Isis Distributed System Inc. The Isis Distributed Toolkit: Version 3.0, User Reference Manual, 1992.

[JSP93] G. Lobe J. Schaeffer, D. Szafron and I. Parsons. The Enterprise Model for Developing Distributed Applications. *IEEE Parallel and Distributed Computing*, 1(3), August 1993.

[KC95] Jae H. Kim and Andrew A Chen. Network Performance under Bimodal Traffic Loads. *Journal of Parallel and Distributed Computing*, 28(1):43–64, July 1995.

[KS93] Eileen Kraemer and John T. Stasko. The Visualization of Parallel Systems: An Overview. *Journal of Parallel and Distributed Computing*, 18:105–117, 1993.

[MBS91] T.A. Marsland, T. Breitkeutz, and S. Sutphen. A network multi-processor for experiments in parallelism. *Concurrency: Practice and Experienc*, 3(1):215–29, 1991.

[MMW87] Joanne L. Martin and Dieter Mueller-Wichards. Supercomputer Performance Evaluation: Status and Directions. *The Journal of Supercomputing*, (1):87–104, 1987.

[MT95] Eric Maillet and Cecile Tron. On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 28(1):84–93, July 1995.

[Par93] I. Parsons. An Appraisal of the Enterprise Model. Master's thesis, University of Alberta, 1993.

[RJ93] Diane T. Rover and Charles T. Wright Jr. Visualizing the Performance of SPMD and Data-Parallel Programs. *Journal of Parallel and Distributed Computing*, 18:129–146, 1993.

[SG93] Sekhar R. Sarukki and Dennis Gannon. SIEVE: A Performance Debugging Environment for Parallel Programs. *Journal of Parallel and Distributed Computing*, 18:147–168, 1993.

[SSLI93] J. Schaeffer, D. Szaffron, G. Lobe, and I.Parsons. The Enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, 1993.

[Sun90] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–319, 1990.

[ZNQ93] Xiaodong Zhang, Naga S. Nalluri, and Xiaohan Qin. MIN-Graph: A Tool for Monitoring and Visualizing MIN-Based Multiprocessor Performance. *Journal of Parallel and Distributed Computing*, pages 231–241, 1993.