INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700  800/521-0600
ParaView: Performance debugging through visualization of shared data

Speight, William Evan, M.S.

Rice University, 1994
ParaView: Performance Debugging Through Visualization of Shared Data

by

William Evan Speight

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Dr. John K. Bennett, Chairman
Assistant Professor
Electrical and Computer Engineering

Dr. James B. Sinclair
Associate Professor
Electrical and Computer Engineering

Dr. Willy Zwaenepoel
Professor
Computer Science

Houston, Texas

September, 1993
ParaView: Performance Debugging Through Visualization of Shared Data

William Evan Speight

Abstract

Performance debugging is the process of isolating and correcting performance problems in an otherwise correct parallel program. Problems not immediately visible to the parallel programmer often lead to poor application performance. This thesis describes the design, implementation, and use of ParaView, a tool to locate performance inefficiencies in programs written for shared-memory multiprocessors. ParaView supplies an intuitive, graphical interface based upon the X-windows system. ParaView aids parallel applications programmers in uncovering performance bugs relating to poor cache performance, load balancing, false sharing, and inefficient synchronization. Eleven parallel programs have been analyzed using ParaView, and performance limitations in five of these were improved. Reductions in overall execution times range from 25% to 86% for sixteen processor simulations. Our experience demonstrates that ParaView facilitates parallel program performance debugging by reducing the amount of time required to uncover and correct performance problems relating to poor data partitioning, false sharing, contention for shared data constructs, and unnecessary synchronization.
Acknowledgments

This thesis would still be just an idea were it not for the help of many people. Special thanks go to Dr. John Bennett for helping me with the concept for my research and supporting me during my stay here at Rice. I would also like to express my gratitude to Dr. Bart Sinclair and Dr. Willy Zwaenepoel for the comments and suggestions they gave as members on my thesis committee. Thanks to Jay Greenwood and Kathi Fletcher for helping me on the original ParaView paper that eventually became the kernel of this thesis, and to Ram Rajamony for knowing the answer to any computer-related question I've ever had. Thanks to Sandhya Dwarkadas for the help in instrumenting RPPT for use with ParaView. Thanks to my brother Colin Speight for lending moral support when the going got tough. Finally, I would like to thank Audrey Yu for all her encouragement, help, patience, and understanding during these last two years.
To my parents, Van and Margret Speight

Without whose constant support this thesis would never have been possible
Contents

Abstract ii
Acknowledgments iii
List of Tables viii
List of Illustrations ix

1 Introduction 1
1.1 Background and Motivation 3
1.2 Difficulties in Parallel Programming 4
1.3 ParaView: A Data Visualization Tool 6
1.4 Basic Design Decisions 7

2 Related Work 9
2.1 Parallel Program Performance Debugging 9
  2.1.1 Parallel Program Debugging 10
  2.1.2 Parallel Program Visualization 10
2.1.3 System Performance Monitoring 11
2.1.4 System Performance Visualization 13
2.1.5 Data Performance Monitoring 14
2.1.6 Data Performance Visualization 15
2.1.7 Overall Packages 16
2.2 Related Work in False Sharing 17
3 Overview

3.1 Programming Environment and Architecture Simulator .............................................. 20
3.2 The Preprocessor ........................................................................................................ 21
3.3 Using ParaView .......................................................................................................... 22
  3.3.1 Data Access Graphs ............................................................................................... 23
  3.3.2 Execution Signature Graphs ................................................................................... 27
  3.3.3 Load Balance Plots ................................................................................................. 30
  3.3.4 Execution Miss Rate Plots ....................................................................................... 33
  3.3.5 Hot Spot Display ................................................................................................... 34

4 Results .............................................................................................................................. 36

4.1 Applications Studied .................................................................................................. 36
4.2 Analysis of Common Programming Problems .............................................................. 37
  4.2.1 Poor Data Partitioning/Load Balancing ................................................................. 37
  4.2.2 False Sharing .......................................................................................................... 47
  4.2.3 Contention for Shared Data Constructs ................................................................. 52
  4.2.4 Unnecessary Synchronization ............................................................................... 59

5 Implementation of ParaView ........................................................................................... 63

5.1 Trace Format ............................................................................................................... 63
5.2 Trace Flow ................................................................................................................... 65
5.3 Grouping Data Objects ............................................................................................... 67
5.4 Passing Information from Simulators to Preprocessor .................................................. 68
  5.4.1 Barrier Information ............................................................................................... 68
5.4.2 Initialization Information .................................. 69
5.4.3 Dynamic Memory Information .............................. 69
5.5 Support for Other Architectures .............................. 70

6 Evaluation ......................................................... 71
   6.1 Analysis of the Results ..................................... 71
   6.2 Limitations .................................................. 75
   6.3 Areas for Improvement .................................... 78

7 Conclusion ......................................................... 80
   7.1 Summary ..................................................... 80
   7.2 Future Work ................................................ 81

Bibliography ......................................................... 82
Tables

4.1 Applications Used in Study .................................................. 37

5.1 Trace File Fragment ................................................................. 64
5.2 Data Access Types ................................................................. 64
5.3 Hit/Miss Types Determined by Architecture Simulator .................. 65
5.4 Example Grouping File ......................................................... 68
5.5 Special Trace Lines ............................................................... 68

6.1 Adjusted Speedups for 16 Processors ...................................... 73
6.2 ParaView Overheads ............................................................... 77
Illustrations

1.1 Development of a Parallel Program ........................................ 2

3.1 ParaView and Associated Components ................................. 19
3.2 Sample ParaView Screen .................................................. 22
3.3 Data Access Graph for JACOBI ........................................... 23
3.4 Data Access Graph for JACOBI, variable barrier ................. 26
3.5 Detailed Data Access Graph for JACOBI, variable barrier ...... 26
3.6 Execution Signature Graph for JACOBI ................................. 29
3.7 Zoom of Second Iteration in JACOBI ................................. 29
3.8 Load Balance Plot for JACOBI - View 1 .............................. 32
3.9 Load Balance Plot for JACOBI - View 2 .............................. 32
3.10 Load Balance Plot for JACOBI - View 3 ............................. 32
3.11 Load Balance Plot for JACOBI - View 4 ............................. 32
3.12 Execution Miss Rate Plot for JACOBI ................................. 33
3.13 Hot Spot Display for JACOBI ............................................ 35

4.1 Simulated Architecture .................................................... 36
4.2 Base Case Speedups ....................................................... 38
4.3 SIEVE Data Access Graph ................................................. 40
4.4 Improved SIEVE Data Access Graph .................................... 40
Chapter 1

Introduction

This thesis examines the design and application of ParaView, a tool to aid computer programmers in the process of parallel program performance debugging. Performance debugging involves taking a working parallel program and improving the inefficiencies that lead to less than ideal execution. ParaView reduces the time required to understand and remove performance bugs by focusing the programmer’s attention on problematic areas quickly and effectively. Eleven parallel programs were analyzed using ParaView, and our experience demonstrates that a tool such as ParaView can facilitate parallel program performance debugging.

ParaView provides an avenue for programmers to discover why programs perform poorly by displaying information on data sharing patterns, load balancing, synchronization, and cache performance. Although ParaView is geared toward analyzing shared-memory system performance, distributed-memory systems could also be easily adapted to provide ParaView with information needed for analysis. ParaView provides the following displays to aid parallel programmers:

1. Data Access Graphs show cache accesses during program execution.
2. Load Balance Plots display how long each processor executes barrier code.
3. Execution Miss Rate Plots show how miss rates vary over time.
4. Execution Signature Graphs display cache utilization.
5. Hot Spot Displays highlight accesses to specific data structures.
ParaView provides these displays in an intuitive, X-driven graphical environment with facilities for zooming in on interesting areas of a program's execution, printing charts and graphs to postscript files, and changing the analysis granularity.

Results reported here were obtained by analyzing traces gathered from two different simulation environments: Mpsas [22, 29], an extensively modified version of a cycle-level simulator obtained from SUN Microsystems; and RPPT [17] (the Rice Parallel Processing Testbed), an execution-driven simulator developed at Rice University.

![Flowchart of Parallel Program Development](chart.png)

**Figure 1.1 Development of a Parallel Program**
1.1 Background and Motivation

The development of performance debugging tools for the parallel programmer has lagged behind the development of such tools for sequential programmers. Synchronization between processors and increased data access complexity make effective parallel programming and analysis more complicated than its sequential counterpart. Interactions between the underlying architecture and the parallel application further increase the complexity of parallel programming. Performance debugging is usually a critical step in the successful development of a parallel program.

Figure 1.1 depicts the parallel program development process. Most of the effort is usually concentrated in writing the actual parallel code or in parallelizing existing sequential code. After parallel code has been generated and has been determined to be functioning correctly, the code must be tuned to take into account the interaction of the code with the architecture of the target machine.

The performance debugging process is shown in the shaded region labeled “c” in Figure 1.1. The performance debugging stage of program development is a necessary step to producing a successful program, and performance debugging tools are crucial to the effective tailoring of each application to a specific architecture. To be useful to programmers, performance debugging tools need to be easy to use and reasonably intuitive. Additionally, the tool should not require an intricate knowledge of the system on which the application is running. Clearly, the more a programmer knows about system characteristics, the better the understanding of the program-system interaction will be. Such knowledge will likely facilitate the performance debugging process. However, a good performance tool will convey information about application-architecture interaction to the novice programmer during use of the tool.
Performance tools should also be non-intrusive to reduce perturbations to the program under analysis. The separation of simulation time and actual program execution time in a simulation environment eliminate perturbations that would otherwise be present in an actual multiprocessor system.

1.2 Difficulties in Parallel Programming

After a parallel program produces correct results, much work often remains. It is quite difficult to find sources of inefficiencies in parallel programs without the use of tools specifically designed for that purpose. Simple speedup measurements can provide a measure of parallel program performance, but they do not help the programmer discern where the sources of program inefficiencies lie. The following paragraphs outline the major sources of inefficiencies in parallel programs.

Poor Cache Performance
To a large extent, the performance of shared-memory parallel programs depends upon effective use of the memory sub-system, particularly available cache memory. Programs tailored for a specific memory system design will outperform applications which have been written with an unspecified target machine architecture in mind. Performance tools must be designed to show the user to what extent the program under analysis is effectively exploiting the specific features of the architecture being used. The effect of cache and memory behavior on performance can be due to many different factors, including fine-grained sharing, conflicting mapping problems, and false sharing.

Synchronization Problems
Synchronization is another characteristic of parallel computing that often causes less
than ideal performance in parallel applications. Processors in a multiprocessor environment must communicate with each other to keep each processor's view of data coherent. Insufficient synchronization can result in incorrect execution, and unnecessary synchronization can result in poor performance.

Closely related to synchronization is the underlying coherence or consistency protocol supported by the multiprocessor system. Different consistency protocols vary the points at which shared data is made to be coherent, and tradeoffs between these protocols can drastically alter performance. The reasons for performance gains or losses obtained from these tradeoffs should be made visible to the programmer.

Load Balancing
Efficient partitioning of work in a parallel program can be difficult. Often the easiest partitioning leads to a work load imbalance among processors. A measurement of global program performance such as speedup numbers or overall execution times will reflect a load imbalance, but such measurements give programmers little clue as to the nature of the load balancing problem or how to correct it.

False Sharing
False sharing is a situation in which two or more processors repeatedly write to data found in different portions of the same cache block without actually accessing the same piece of data. This situation causes the cache block to "ping-pong" back and forth between caches, resulting in a considerable waste of communication bandwidth. Programmers can intelligently place or partition data to avoid false sharing if the problem is known to exist.

Adaptive Caching Support
Recent research has indicated that maintaining different caching protocols for data
items depending on how the user or compiler expects the data to be accessed may lead to significant performance improvements [20, 5, 7]. Few tools currently exist to aid the programmer in determining how a particular data structure or element should be categorized for such an adaptive caching strategy. Similarly, designers of adaptive cache controllers need to be able to determine the most useful set of protocols to fit observed data access behavior. Thus, both the programmer and the architect desire a means of categorizing data access behavior in parallel programs.

1.3 ParaView: A Data Visualization Tool

In this thesis, several parallel programs are analyzed and improved through the use of the ParaView parallel program performance debugger. The work is two-fold: demonstrating that performance debugging shared-memory programs is feasible, and designing ParaView, a functional performance debugger. ParaView is the result of an iterative process of analyzing parallel programs for problems, deciding what new information would have been of help in performance debugging the program, working this feature into ParaView, and repeating the process. As a result, ParaView provides programmers with methods for detecting problems with synchronization, load balancing, poor cache performance, and false sharing in an easy-to-use X-driven environment.

ParaView interprets traces from program execution and displays program behavior and resource usage. Users may focus on specific regions of execution or on the entire program run. ParaView shows accesses to user-specified data structures, data or code, and allows the user complete control over what is displayed. Data Access Graphs distinguish read, write, and read-modify-write accesses from one another, and cache hits and misses are clearly marked. Cache misses are broken down by the cause
for the miss: collision, compulsory, or consistency, with false sharing misses clearly 
distinguished from true sharing misses. Execution Signature Graphs categorize 
execution time into time spent doing parallel work, synchronization time, sequential 
time, and time spent stalled while waiting for access to a bus. Load Balance Plots 
show the amount of time each processor spends waiting to be released from a barrier, 
and Execution Miss Rate Plots show just where in the program high miss rates 
lead to performance degradation. Finally, Hot Spot Displays show accesses to 
specific data structures over the course of the program execution.

1.4 Basic Design Decisions

There are three main considerations when designing a performance debugging tool:

1. Data Gathering - How will the data be collected and in what format will it be 
kept? General data, such as time spent in basic blocks, can be maintained, or 
more detailed data such as individual cache accesses can be recorded.

2. Data Analysis - Will the analysis be carried out post-mortem or during the 
execution or simulation of the program? Will it be analyzed for low-level results 
such as cache hit rates, or higher-level results such as speedup?

3. Data Presentation - Will the statistics generated from analysis of the data be 
presented to the user through graphs, tables, or a combination? How much 
control will the user have over display of the data? Is program performance 
visualization the intended method of performance debugging, or is the tool just 
intended to point out the most troublesome spots of execution?

ParaView has been designed to take advantage of the most detailed information 
available in order to better aid the programmer in the performance debugging task.
A trace line format has been developed that can be easily produced from almost any simulation environment. Hardware monitoring can also be used to gather traces, and an effort to design hardware support for ParaView is under way. The trace line allows for a great deal of flexibility depending on how detailed an analysis the user wishes (see Section 5.1). Currently, trace information on every data access is recorded and analyzed on the fly by ParaView. The view of data presented is both at a low level (i.e., Data Access Graphs) and higher levels (e.g., Execution Signature Graphs). Finally, ParaView provides users with graphical output that quickly focuses attention on performance problems. The information presented on the graphs is backed up with separate windows displaying statistical information relating to the current graph.

The demonstration of the use of ParaView was not limited to small programs. Several programs come from the SPLASH [50] benchmark suite and have many millions of shared data references. The majority of improvement studies presented come from smaller programs written in the C programming language. However, unlike other data analysis tools, which restrict the programming language, ParaView provides for analysis of programs written in any language.

Data sizes analyzed in this paper are somewhat small compared to what would normally be used in a production run. This restriction is present in simulation runs even without ParaView due to the large overhead associated with simulating multi-processor environments. ParaView itself places no restriction on the data size of the program under analysis, but simulation time is increased somewhat due to the large amount of on-the-fly analysis performed by ParaView.
Chapter 2

Related Work

Performance debugging is the process of taking a correctly functioning program and solving problems with resource contention, load balancing, and poor memory subsystem performance. Contributing to poor parallel program performance is the problem of false sharing, a situation arising when two processors access different pieces of the same cache block. Finally, poor cache performance may be improved if cache designers supply a set of coherence protocols for cache blocks depending on the expected sharing patterns of each individual block. Prior work in each of these areas is discussed below.

2.1 Parallel Program Performance Debugging

Previous work in this area has taken 6 major directions:

- parallel program debugging
- parallel program visualization
- system performance monitoring
- system performance visualization
- data performance monitoring
- data performance visualization

The first two of these categories stress the production of correct parallel programs through debugging and visualization and are represented by the box labeled “b” in
Figure 1.1. ParaView does not address the issue of producing a correctly functioning program directly, but some features of ParaView offer insight into certain programming problems. After correctness has been achieved, a crucial step in the successful development of any parallel program is the process of performance tuning. The last four of the above categories address recent efforts in this area.

2.1.1 Parallel Program Debugging

The difficulty in debugging parallel programs for correctness arises in part from the fact that the execution of parallel programs is not reproducible. In other words, events are not guaranteed to occur in the same order between successive runs of the program. Therefore, the try-and-retry methods of sequential debugging are not appropriate for parallel debugging. LeBlanc and Mellor-Crummey [39] cite two methods for debugging parallel programs. The first of these methods, used in [14] and [24], involves taking snapshots of the program's execution that the user may then browse in an effort to spot abnormalities. However, the user is required to describe every event that may possibly lead to an error since all data for debugging must be taken on a single execution of the program. This can be quite tedious and time-consuming. The second debugging method is to force the program to behave in a reproducible manner, allowing the cyclic methods employed in sequential debugging to be used. This can be achieved through a variety of methods, as outlined in [39], [16], and [23], but often leads to an inability to exploit all possible parallelism by requiring all operations on shared semaphores to be serialized.

2.1.2 Parallel Program Visualization

Parallel program visualization attempts to graphically depict individual blocks or sections of code and to present program flow through visual means [48, 52, 11]. By using
graphics to elucidate aspects of software such as flow of control, program function, and communication protocols, a program visualization tool may help the user form a correct mental image of a program's structure [11].

PV [11] focuses on dynamic visualization that portrays the progress of a program during execution. PV contains support for a set of graphical symbols to track the progress of common data structures such as numeric variables, trees, linked lists, and arrays. PV also provides such features as graphical manipulation of code, diagrams of computer systems, and user-controlled creation and traversal of multidimensional information space to aid the programmer in visualizing the execution of the application.

2.1.3 System Performance Monitoring

System monitoring tools provide the user with statistics or simple graphs to understand how system elements behave during the course of the program execution. In a distributed environment, factors such as message-counts, resource utilization, and queue lengths may be presented in an effort to aid in understanding the sources for inefficiencies. In a shared-memory environment, bus utilization, cache performance, or memory sub-system overhead may be analyzed to uncover performance problems. More suited to analyzing the underlying system than the application, numerous examples of this type of tool have been developed [35, 47, 43, 54, 44, 30, 46, 2, 26]. System monitoring tools are quite general, and there is a certain degree of difficulty in relating the information presented back to exactly where in the program the problems lie.

Two monitoring packages, Quartz [1] and Parasight [2], were inspired by gprof [28], a Unix utility for profiling sequential program execution. Both Quartz and Parasight aim to help the programmer focus on the most computationally intensive portions of
the code by profiling typical runs of the program. Quartz extends the gprof model of program activity to parallel codes by defining “normalized processor time”, a metric for determining the “hot spots” of a parallel program, and takes care to accurately account for synchronization costs. Parasight is a gprof-like utility that is designed to be non-intrusive, to allow the user fine-grained control over regions of the program to be profiled, and to make use of available processors in a parallel system to collect and analyze data on the fly. In ParaView, the issue of efficient data capture has been transferred to the simulator or multiprocessor generating the traces. ParaView’s analysis is as accurate as the source that generates the traces. ParaView is therefore able to provide more in-depth detailed information on cache hit rates, synchronization, etc., than either Quartz or Parasight. However, the level of detail that ParaView provides comes at the expense of analysis time and trace storage.

IPS [47] is a performance analysis tool designed to collect and analyze data from application programs as well as external sources (such as hardware, networks, and operating systems). Performance metrics are displayed as a function of time. IPS concentrates on displaying resource utilization graphs by providing a common trace format so that data can be collected from external system components.

Mtool [26] instruments basic blocks and loops to log timing information on how much time is spent within each profiled section. Next, Mtool calculates the amount of time the simulation should take in each section, given a perfect memory subsystem. The difference between how much actual time elapses and how much should have elapsed given perfect execution is the memory overhead associated with that section of code. Mtool’s main benefit is its speed. It is very quick and adds little overhead, since no detailed statistics are kept. Mtool does a good job focusing on where the problem areas lie, but does little to help the user understand the nature of
the problem. ParaView provides cache hit rates and breakdowns of cache misses as an added way to determine the causes of poor performance.

2.1.4 System Performance Visualization

Taking system performance monitoring one step further, efforts have been made toward providing more than statistics or simple graphs to focus the user's attention on areas [13, 53, 51, 8, 38, 35, 31] of execution that suffer some performance degradation. By helping the user to visualize what is going on with the various components of a system during a simulation, intuition may be gained for what is causing the performance problems.

PREFACE [8] is a performance monitoring tool for the visualization of parallel applications. A pre-processor inserts calls to a tracing routine for every use of the parallel constructs of interest to the user. Data is then taken each time these constructs are used. The results can be saved to trace files for later study or stored in memory only for current viewing. PREFACE's main output are plots showing projected speedups for each section of code based on trace information captured for the parallel constructs contained in that section of code. As with other tools, PREFACE does a good job of pointing out problem areas, but does little to show the cause of these problems in performance.

ParaGraph [31] provides a visualization tool for parallel distributed systems. ParaGraph interprets traces generated from the Portable Instrumented Communication Library [21] and provides post-mortem animation of many distributed system parameters such as message counts, load balancing among nodes, network utilization, etc. Over 25 different representations of the trace files are presented in an interactive graphical interface. ParaGraph shares many features with ParaView, including post-mortem analysis of trace files that allows repeatable analysis. Both provide graphical
interfaces that facilitate usage by novice and advanced programmers alike. Unlike ParaView, however, ParaGraph integrates network topology and other system features into its analysis techniques, and can therefore only be used in a distributed environment. ParaView is designed for the analysis of shared-memory programs, but can also be used to analyze distributed-memory programs. ParaView offers a more limited amount of basic information, but the information it does give can be used equally well in a distributed or shared-memory environment.

2.1.5 Data Performance Monitoring

Data performance monitoring differs from system performance monitoring in that the focus is on how the data set of a parallel program is accessed during the execution of a parallel program rather than how the system components behave. In other words, statistics are presented in terms of data structures familiar to the programmer and not just raw cache hit rates, bus utilizations, etc. To gain an accurate view of this performance, it is helpful to find out what data structures cause the inefficiencies and where in the code the inefficiencies lie.

MemSpy [45] aids programmers in identifying memory system bottlenecks in shared-memory systems and is directly linked with the Tango execution-driven simulator [27]. The focus is on cache performance, with the notion that poor multiprocessor performance stems mainly from poor cache performance. Statistics such as cache hit rates and cache miss rate breakdowns are presented in terms of both procedure and data object. Thus it is possible to tell which data object is responsible for the poor cache performance, and where the degradation occurs. ParaView provides all the functionality of MemSpy with some added benefits. ParaView's main mode of information transfer to the user is through easy-to-understand plots and graphs. Statistics are provided to backup what the user "sees" on the screen. ParaView also
allows the user to obtain information on a much finer level than the procedure level of information maintained in MemSpy. Finally, MemSpy does not provide help for load balancing, false sharing, or synchronization problems as does ParaView.

2.1.6 Data Performance Visualization

Data performance visualization tools [49, 10, 18, 40, 25, 12] attempt to give the user a sense of program flow through the access patterns of the data present in the program. This sense of program flow may be provided through a dynamic display of accesses to data structures during a program's execution, or a replay from a saved trace file. Program bottlenecks due to inefficient data usage can often be easier to find if some form of visualization is used.

SHMAP [18] provides a graphical display of reads and writes to FORTRAN matrices. Colors are used to represent processors and each data element is given the color of the processor that most recently accessed it. SHMAP, like ParaView, illustrates data accesses to shared objects over time. By limiting the display to FORTRAN matrices, SHMAP is able to provide the user with a graphical representation of the shared data object based on the "shape" of the object. In other words, if the structure under analysis were a two dimensional array, the boundary of the display would appear in the form of a rectangle on the screen with accesses to specific elements highlighted at the time of access. In addition, SHMAP is able to take advantage of knowledge of certain well known code templates, such as the Basic Linear Algebra Subprograms [37], which can be used to reduce the size of the resultant trace file. ParaView, on the other hand, provides support for multiple languages and can display all shared data objects. One advantage of displaying all data accesses is that the synchronization data constructs protecting data access can be displayed along with the data that these constructs protect.
2.1.7 Overall Packages

Finally, some performance debugging tools are part of an entire programming and analysis package. Systems such as [4], [36], [41], [32], and [34] provide simulation or real-time program execution along with debugging tools, trace generation, data compression, performance estimate displays, etc. The box labeled "a" in Figure 1.1 shows that these systems seek to provide a complete environment for producing, running, and analyzing parallel programs. Pablo [4] is a toolkit for building analysis and visualization tools in a portable, scalable, and extensible manner. ChaosMON [36] is a framework for performance information capture and display. Programs, systems, and architectures to be displayed by ChaosMON are described in a special format processed by a database engine. Frameworks such as Pablo and ChaosMON are used to study display methodologies and user interfaces. ParaView is more specifically geared toward providing information to inform the improvement of parallel systems and programs.

PIE [41] maps parallel applications onto specific architectures and allows the user to observe the performance when these applications execute. Designed to run on top of the Mach operating system, PIE can be ported to other systems. The authors provide program visualization through relating textual entries in the code to boxes displayed on the user’s screen. Program flow is shown by arrows connecting the various boxes. Users may “browse” the program by merely clicking on boxes in the visual representation of the program. This action will cause the section of code corresponding to the selected box to be displayed. PIE [41] also provides some system performance visualization capabilities. Visualization of process usage is achieved by graphs showing regions of thread execution over the course of the application’s lifetime. Plotting each thread’s periods of execution vs. time shows where resources
are being underutilized and helps to point out load balancing problems. ParaView
shows similar information, but also provides specific performance data to accompany
such analysis, which PIE does not. Basically a load balancing tool, PIE does not
provide ParaView’s level of detail for performance analysis.

2.2 Related Work in False Sharing

False sharing hinders parallel performance in both shared-memory and distributed-
memory systems. When two processors write repeatedly to the same cache block, the
block tends to “ping-pong” back and forth between the two processors although they
may be writing to different variables.

Three main algorithms exist for the detection of false sharing in multiprocessor
systems. A discussion of the differences between the three methods is presented
in [19], the main points of which are outlined below.

The first false sharing detection algorithm was proposed by [55]. In this scheme,
the first reference to a word by a given processor is labeled as a cold-start miss. A
true sharing miss is one in which the following sequence of accesses to the same cache
block must occur:

- a reference to the cache block that misses,
- a reference to a different word in the same cache block that has been accessed
  before, and
- a reference to the cache block that would miss in a system with a block size of
  one.

If a consistency miss does not follow the outline above, it is considered a false sharing
miss. A somewhat different approach is taken in [20]. Here, the only the first reference
to a cache block is counted as a cold-start miss. Thereafter, invalidation misses are
counted as true sharing misses if the word has been modified since the last reference.
The algorithm implemented in ParaView is derived from the work done by [19]. In this scheme, false sharing misses are defined as those that do not bring any new information into the cache that is then used before the next invalidation to the cache block. Thus, if a false sharing miss as defined in [20] brought in an updated value to some other word in the block, and this value was used before the next invalidation, the miss is not counted as a false sharing miss. This requires delaying classification of misses until it is known whether any new value is used. In ParaView, extra instrumentation has been implemented to account for finite caches. The method presented in [19] is developed for infinite caches only. In addition, ParaView maintains extra state to determine exactly which references are false sharing misses for graphical display. The original algorithm only maintained a running total of all miss types.

2.3 Data Access Grouping

Several studies have shown that data is accessed in multiprocessors in a few distinguishable patterns [6, 3, 33, 56]. Performance gains can be achieved if methods are provided to maintain coherency in the system depending on how the user (or compiler) expects a certain piece of data to be shared among the various processors present.

Bennett et al. [5] display aggregate shared data access behavior in tabular format in their study of several parallel programs. Aggregate information is useful for determining common access behavior but does not show how programs and processors access data objects during program execution. ParaView provides this capability through detailed graphical depiction. Sharing patterns are easily discernible from ParaView's displays.
Chapter 3

Overview

This section presents an overview of the ParaView experimental environment, whose organization is illustrated in Figure 3.1. For the studies presented in this thesis, parallel programs are simulated using one of the two available simulation environments. Each simulator is capable of providing the detailed traces analyzed by ParaView, but ParaView is general enough to allow analysis of any traces that follow the standard trace format described in Section 5.1.

Each of the functional components of the parallel simulation environment used in this thesis is briefly described below.

![Diagram of ParaView and Associated Components](image)

Figure 3.1  ParaView and Associated Components
3.1 Programming Environment and Architecture Simulator

The ParaView programming environment provides a parallel programming macro library that is based on the Parmacs suite [9]. The use of a standardized macro package hides the implementation of parallel programming support from application programmers. Information that helps ParaView efficiently analyze trace data may be embedded within the macros and therefore be transparent to programmers. This information includes flags to indicate barrier entry and exit, a signal at the end of the initialization phase of program execution, and information used to trace dynamically allocated data.

To support analysis by ParaView, parallel programs written in C are processed by m4, a standard macro processor, and compiled using cc. The resultant object code is either executed directly on Mpsas, the cycle-level architecture simulator, or profiled for use with RPPT, the execution-driven simulator. Both simulators are capable of simulating cache protocols, various processor-cycle to bus-cycle ratios, bus bandwidths, cache sizes, and multi-level memory hierarchies. The simulators have been instrumented to generate traces that provide ParaView with a record for each data access. The following elements are included in this record: the unique id of the cache processing the data access, the type of access (read, write, or read-modify-write), the physical address*, the cycle in which the access was requested, the category of cache hit or miss, and in the advent of a miss, the cycle in which the bus was acquired and released. The current implementation of ParaView does not support split-transaction busses.

---

*In both simulators, the physical address was identical to the virtual address
3.2 The Preprocessor

The trace preprocessor shown in Figure 3.1 performs initial trace analysis either concurrently with the simulation, or post-mortem on a trace file generated during execution. During this analysis, ParaView computes the information necessary to produce the Execution Signature Graph, the Load Balance Plot, and the Execution Miss Rate Plot (see Section 3.3); detects all false sharing accesses; and splits the trace information from the simulator into separate, reduced trace files distinguished by whether a particular reference is to shared data, program text, or private data. Private data includes all data that is neither shared data nor text. User controlled options specify the amount of information saved for later analysis. For example, if disk space is at a premium, only cache misses during the parallel portion of the program's execution may be saved. Since the majority of cache accesses should be hits, this option reduces the amount of space needed for trace storage, while still providing enough information to debug the application's performance problems. These trace files are then used by ParaView for the Data Access Graph (see Section 3.3) and the Hot Spot Display.
3.3 Using ParaView

ParaView provides a full color, X-driven graphical interface for the results analyzed by the pre-processor. User control is achieved through mouse movements, push buttons, pull-down menus, etc. All displays produced by ParaView may be viewed on the computer screen or saved in postscript format for printing. Figure 3.2 shows a snapshot of the screen during a typical interactive session with ParaView.

![Sample ParaView Screen](image)

Figure 3.2 Sample ParaView Screen

The following paragraphs outline each type of display ParaView produces and gives postscript examples of each. The examples in these sections are taken from an execution of JACOBI, an iterative, finite differencing algorithm used to solve partial differential equations. A matrix consisting of \( m \times n \) elements is allocated, and the
boundary values are set to constant values at the beginning of the algorithm. Interior cells are updated according to the equation:

$$n_{r,c} = \frac{(n_{r+1,c} + n_{r-1,c} + n_{r,c+1} + n_{r,c-1})}{4}$$

The algorithm partitions the matrix into squares that are then assigned to each processor. Within each square, the matrix is divided into "red" and "black" squares in a checker-board fashion. During the first iteration, each "red" cell is updated as the average of its four neighbor "black" cells. The processors then synchronize and begin work on all black cells. This example uses 4 processors to iterate 10 times over a 128 × 128 input matrix.

3.3.1 Data Access Graphs

![Data Access Graph: JACOBI(Red/Black)](image)

Figure 3.3 Data Access Graph for JACOBI
After the application program has completed execution, ParaView's data access analyzer generates detailed displays of data references throughout the program's execution. The Data Access Graph shown in Figure 3.3 selectively shows either all cache accesses, or cache misses only. Cache misses are displayed according to the reason for the miss: compulsory, collision, or consistency. False sharing misses are shown in the Data Access Graphs with boxes around the symbol for a consistency miss as in Figure 3.3. Accesses are classified as either reads, writes, or read-modify-writes. The x-axis shows the cycle count as the execution progresses, and the y-axis represents the address space of the parallel program and is divided into 3 regions. The first region shows references to text, the second region displays shared data references, and the third region shows private data references. The text and shared data address space begins with the first address of program text at *x* = 0 and goes through the last shared data reference. All accesses to private data are displayed on one line at the top of the graph since the private space is prohibitively large. In addition, it is not possible to link private addresses to actual program constructs without instrumenting the user code directly. This limitation does not seem to be much of a problem, since the major trouble with private data is that it may map on top of critical section text or shared data space in a finite cache (especially a direct-mapped cache). This condition can easily be seen in the Data Access Graphs.

The default Data Access Graph depicts accesses to all data by cache 0 in the program over the entire execution. The user may selectively expand periods of execution time or address ranges of particular data. Pushbuttons allow viewing of other caches in the system. Zooming can be specified by outlining the desired region with the mouse or specifying exact cycle numbers or address ranges for the zoom to cover. Additionally, information relating cycle numbers to procedures in the program's execution can be obtained by outlining the portion of the graph with
the mouse and selecting a push button. In the same manner it is possible to obtain information linking a portion of the graph to the data constructs displayed.

As discussed in [5], multiprocessor systems that provide for multiple cache protocols can show better performance than a system providing only one coherence protocol. In order to enable programmers to identify which sharing pattern a particular data object may exhibit, ParaView produces a display depicting data accesses for each variable to each cache in the system. All accesses to the variable in a certain cache are shown on one line, and all accesses by all caches are shown together. Variables displayed may be a shared data object, a procedure in the parallel code, or all private data. Examination of accesses to data constructs by all caches at one time enables the user to quickly identify the sharing pattern exhibited by a certain data object.

Figure 3.4 is an examples of this type of **Data Access Graph** for the *barrier* variable used in our programming environment. The barrier structure consists of 4 elements: two counters, a lock protecting access to the counters, and a sense variable. The sense variable points to which of the two counters is presently being used to track how many processors have arrived at the barrier. Processors spin on a local copy of this sense variable. When the last processor arrives, it changes the sense to point to the other counter, thereby releasing all processors currently waiting at the barrier.

It is difficult to tell what sharing pattern is exhibited by the entire barrier structure in Figure 3.4. ParaView can be instructed to split the barrier structure into its various components and display these separately, as is shown in Figure 3.5. Using the classifications described in [5], we can tell that the lock variable exhibits the behavior of synchronization data (reads, write, and read-modify-writes); the two counters fall into a migratory category since processors access them each once per iteration; and the barrier sense shows read-mostly behavior.
Figure 3.4 Data Access Graph for JACOBI, variable barrier

Figure 3.5 Detailed Data Access Graph for JACOBI, variable barrier
ParaView automatically groups variables for this type of access display. Each shared variable is grouped in its own group, as well as each procedure. All private data are grouped together. As demonstrated above, viewing of these data groupings is not restricted to those automatically determined by ParaView. Users may place a file in the working directory containing address ranges and labels for groups to display together. This feature can be useful for combining data constructs believed to behave similarly or splitting up large data structures to view accesses to individual components as in Figure 3.5.

3.3.2 Execution Signature Graphs

Execution Signature Graphs show how effectively programs make use of program execution time. Figure 3.6 shows an example Execution Signature Graph, and Figure 3.7 shows a zoom of the second iteration of the algorithm. The x-axis shows the processor cycle number, and the y-axis shows the percentage of program execution during which the cache controller is busy.

To produce the Execution Signature Graphs, cache usage is categorized into one of 7 categories:

1. **Stalled at Barrier** - The time spent waiting at a barrier is calculated as the time between subsequent reads of a special shared variable by the same processor. The barrier macro causes each processor to read a known address upon each entry into and exit from a barrier. By looking for these reads, ParaView can determine how much time is being spent executing barrier code.

2. **Stalled for Lock** - Time spent waiting for access to a lock is taken to be the time between a failed read-modify-write operation and the last reference to the lock variable by the same processor.
3. **Acquiring Lock** - Time spent executing lock acquire code is the time from the last reference to the lock variable to the first reference to shared data.

4. **Sequential Time** - Time spent executing critical section code is counted as the time between the end of the lock acquire phase and the beginning of the lock release phase.

5. **Releasing Lock** - The time spent executing lock release code is the time between the last reference to shared data in a critical section and the write to a lock, indicating a lock release.

6. **Stalled for Bus** - Time spent stalled while waiting for access to a bus is the time the cache is idle while waiting for control of a shared bus.

7. **Other** - This category represents time not placed in one of the above categories. During parallel portions of execution, this category is indicative of useful work being performed by the cache. During non-parallel portions, the other category may not reflect useful cache work since idle caches will be counted here.
Figure 3.6 Execution Signature Graph for JACOBI

Figure 3.7 Zoom of Second Iteration in JACOBI
At the end of the simulation, the caches connected to each bus are averaged at a variable processor-cycle resolution to provide a separate view for each bus specified by the user in the "bus_info" file (see Section 5.5). A push button provides a method to change the view among busses and to examine the individual cache execution profile of any cache in the system. In Figure 3.6, the ten red spikes mark the points at which the processors encounter a barrier between iterations. Any area of these displays may be examined in more detail by selecting and zooming in on the region of interest as shown in Figure 3.7.

3.3.3 Load Balance Plots

Barriers are a relatively easy form of synchronization both to understand and implement, and many applications make extensive use of barriers as the main or only form of synchronization. Time spent waiting for release from a barrier can be a major source of performance loss in parallel applications. If some processors are given a larger amount of work to perform between barriers than others, the application can suffer from a load balancing problem. ParaView's Load Balance Plots provide a view of the length of time that each cache spends providing data relating to barrier code for each barrier instance in the program's execution. ParaView's preprocessor contains mechanisms for determining the amount of time a cache spends providing data for barrier-related code as described in the previous section. A file is maintained during the program's execution that tracks these time intervals for each barrier instance in the program's execution, and the contents are plotted in 3-D space. The x-axis denotes the barrier instance number. The y-axis is the cache number, and the z-axis shows the number of cycles each processor spends executing barrier code.

This three dimensional view has four orientations from which the data may be viewed to allow the user to "see" all portions of the graph. A pushbutton provides the
mechanism for switching between views. Figures 3.8 to 3.11 show the four views for
an execution of JACOBI. Several facts about the program execution can be observed
from these graphs. First, all processors appear to wait at least a short amount of
time at every barrier. Because the barrier wait time includes the time to update
barrier locks and access the barrier counter, the time shown in the graph includes the
time to actually process barrier code along with the amount of time actually spent
spinning at the barrier. Second, processors appear to wait longer at the first barrier
than other barriers. This longer waiting period arises because of cold start misses
to barrier code in the first barrier instance. The long barrier wait time for processor
0 at barrier instance 2 results from the fact that processor 0 fills the global matrix
structure at the beginning of the program and does not incur cold-start misses to its
portion of the matrix during the first iteration. Other caches must be filled during
this first iteration. Finally, we can see the results of a high number of collision misses
between the global matrix and parallel code in cache 2 as all caches wait on processor
2 at all barrier instances.
Figure 3.8 Load Balance Plot for JACOBI - View 1

Figure 3.9 Load Balance Plot for JACOBI - View 2

Figure 3.10 Load Balance Plot for JACOBI - View 3

Figure 3.11 Load Balance Plot for JACOBI - View 4
3.3.4 Execution Miss Rate Plots

Execution Miss Rate Plots have the same format as the Load Balance Plots, except that miss rates over program intervals are plotted for each cache in the system. These graphs can give clues that a certain section of code is experiencing problems with cache hit rates, possibly due to collisions between code and data or due to fine-grained sharing. Often the Load Balance Plot will indicate a load balancing problem, and the cause will be a higher miss rate for some processors that leads to a
longer barrier waiting time for others. This situation is shown in Figure 3.12 by the significantly higher miss rates for cache 2 during each iteration. This graph may also be rotated through the views shown in the case of the Load Balance Plot.

3.3.5 Hot Spot Display

The Hot Spot Display is a dynamic display showing accesses to large data structures during the course of a program's execution. The "shape" of the structure can be preserved during the course of the replay, with accesses by each processor highlighted in a different color. As execution progresses, the colors gradually fade to black. Hits and misses are displayed concurrently in separate regions to facilitate an understanding of the access patterns. Hot spots, or heavily accessed areas of a structure, will remain highlighted for a long time and easily stand out during the course of the replay. Figure 3.13 shows a snapshot of the Hot Spot Display during the course of JACOBI's execution. The misses around the section of the global matrix assigned to each processor are due to the sharing of boundary values, and the horizontal line of misses in the blue and yellow areas are the results of the collision of the matrix data with the parallel code.

Because this feature requires complete control of X color maps, a window manager (such as the motif window manager) must be used that allows the installation of a user-defined color map.
Accesses to Data Structure 'array' Over Life of Simulation

Figure 3.13 Hot Spot Display for JACOBI
Chapter 4

Results

4.1 Applications Studied

ParaView has been used to analyze the performance of 11 parallel programs obtained from several sources. All programs were simulated on a 16-processor single bus architecture such as that pictured in Figure 4.1. Each processor has a 64Kbyte, direct-mapped, write-back cache with 32-byte cache lines. A sequential consistency model following that used on the Sequent Symmetry [42] is enforced. The bus used in the simulations does not support split-transactions.

![Simulated Architecture Diagram]

Figure 4.1 Simulated Architecture

Table 4.1 gives information on the programs analyzed with ParaView. The applications range from small programs written by undergraduate students to larger
programs obtained from the SPLASH benchmark suite. Due to simulation time constraints, the problem sizes are relatively small, but the information ParaView generates can be applied to larger size data sets.

<table>
<thead>
<tr>
<th>Application</th>
<th>Lines</th>
<th>Shared Bytes</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIMERGE</td>
<td>750</td>
<td>140000</td>
<td>Parallel sort/merge list of integers</td>
</tr>
<tr>
<td>FFT</td>
<td>250</td>
<td>390000</td>
<td>Perform fast fourier transform</td>
</tr>
<tr>
<td>GAUSS</td>
<td>550</td>
<td>130000</td>
<td>Solve system of linear equations</td>
</tr>
<tr>
<td>LIFE</td>
<td>300</td>
<td>1000000</td>
<td>Solve the game of life</td>
</tr>
<tr>
<td>MMULT</td>
<td>180</td>
<td>50000</td>
<td>Multiply two matrices</td>
</tr>
<tr>
<td>MP3D (SPLASH)</td>
<td>1500</td>
<td>600000</td>
<td>Simulate rarefied hypersonic air flow</td>
</tr>
<tr>
<td>JACOBI</td>
<td>350</td>
<td>520000</td>
<td>Solve partial differential equations</td>
</tr>
<tr>
<td>SIEVE</td>
<td>550</td>
<td>280000</td>
<td>Find prime numbers</td>
</tr>
<tr>
<td>STRIPED JACOBI</td>
<td>550</td>
<td>66000</td>
<td>Solve partial differential equations</td>
</tr>
<tr>
<td>TSP</td>
<td>850</td>
<td>260000</td>
<td>Solve traveling salesman problem</td>
</tr>
<tr>
<td>WATER (SPLASH)</td>
<td>1400</td>
<td>200000</td>
<td>Simulate a system of water molecules</td>
</tr>
</tbody>
</table>

Table 4.1 Applications Used in Study

Figure 4.2 shows the initial speedup curves for these applications. WATER, LIFE, and MMULT scale fairly well, but other applications exhibit poor performance as the number of available processors is increased. The sections that follow describe the use of ParaView in determining and correcting the sources of inefficiencies for the worst of these programs.

4.2 Analysis of Common Programming Problems

4.2.1 Poor Data Partitioning/Load Balancing

SIEVE

SIEVE uses the Sieve of Erostasthanes to calculate all prime numbers below a number \( n \) given by the user. The sequential version proceeds by allocating and clearing an
Figure 4.2 Base Case Speedups

array of characters for each number below the input value. In the first step, all multiples of 2 (excluding 2) less than n are marked as non-prime. On the next pass through the array, all multiples of 3 are marked. This process continues until the square-root of n is reached. Proceeding past the square-root of n only generates redundant work since all non-primes have already been marked. The elements not marked at the end are the primes below the number n.

In the parallel implementation of SIEVE, the first process marks all multiples of 2, the second process marks all multiples of 3, the third process marks multiples of 5, etc. To reduce redundant work during the first iteration, each process begins on
a known prime number. The following code fragment shows the parallel procedure loop of the algorithm:

```c
void
findPrimes(mypId)
int myPid;
{
    int i, j;
    for(i=start[mypId];i<=aSieve.sieveLimit;i += numprocessors) {
        if( *(aSieve.sieveArray+i)) {
            for( j=i+i; j<size; j+=i) {
                *(aSieve.sieveArray+j) = (unsigned char)0;
            }
        }
    }
    BARRIER(bar, numprocessors);
}/* findPrimes */
```

Figure 4.2 shows that the original parallel implementation of SIEVE performs quite poorly with no speedup past 2 processors. ParaView offers insight into why the program performs so badly. Figure 4.3 shows the Data Access Graph for the 4 processor implementation of SIEVE with \( n = 800000 \).

Several features of this graph stand out. First, a significant number of consistency misses occur during the last half of the program's execution. Because the array is made up of characters (to save space), there are 32 elements per cache line. This situation leads to a high degree of fine-sharing among processors throughout the program.
Figure 4.3  SIEVE Data Access Graph

Figure 4.4  Improved SIEVE Data Access Graph
Second, because the array will not fit into any one processor’s cache, a significant number of collision cache misses are present as pieces of the array map on top of one another in each processor’s cache. The collision misses can also be seen in the Data Access Graph. However, the most noticeable problem is that processor 0 completes its work at cycle 11 million and remains idle for the rest of the program. Processors 3 and 4 finish around cycle 14.5 million, while processor 1 continues for another 2.5 million cycles. This indicates a severe load balancing problem. A quick examination of the algorithm reveals the source of the problem. On the first pass through the array, processor 0 marks all multiples of 2 as non-prime. Concurrently, processor 1 marks all multiples of 3 as non-prime. During the next iteration, processor 0 should mark all multiples of 6 as non-prime. However, these numbers are also multiples of 3 and were marked by processor 1 during the previous iteration. During iteration 3, processor 0 should work on all multiples of 10, but these numbers were marked by process 2 during iteration 1, as well as being marked by process 0. This situation continues with processor 0 finding all numbers already marked.

The best solution to solve this load balancing problem and reduce the amount of fine-sharing is to block the data array. When the program begins, regions of the array are assigned to each processor. During the first iteration, processor 0 invalidates all elements that are multiples of 2 up to \( n/\text{num}\_\text{processors} \), processor 1 invalidates all elements that are multiples of 2 and greater than \( n/\text{num}\_\text{processors} \) but less than \( 2^*n/\text{num}\_\text{processors} \), and so on. This partitioning will ensure a fair distribution of work among the processors and reduce the amount of sharing of the global array.
void
findPrimes(myPid)
int myPid;
{
    int i, j, start, end = (myPid+1)*size/numprocessors;
    for(i=2;i<=aSieve.sieveLimit;i++) {
        if( *(aSieve.sieveArray+i)) {
            if(myPid == 0)
                start = i+i;
            else
                start = i*((int)(aSieve.init[myPid]/i));
            for( j=start; j<end; j+=i ) {
                *(aSieve.sieveArray+j) = (unsigned char)0;
            }
        }
    }
    BARRIER(bar, numprocessors);
}/* findPrimes */

Figure 4.4 shows the results of this data blocking. Blocking reduces the overall cache
miss rate from 8.94% to 2.54% for the shared data array and eliminates the load
balancing problem, leading to a greater than 38% reduction in overall execution time.
Without ParaView, the user could tell that there was a large amount of consistency
and collision misses taking place, but the load balancing problem would not have
been readily apparent.

GAUSS

Gaussian elimination is a method for solving linear equations. The goal is to reduce
a square matrix into an upper-triangular matrix that may then be solved using back
substitution. For the matrix shown in Figure 4.5, the algorithm proceeds as follows.
Every element in row j is divided by element $a_{jj}$. Then the correct fraction of row j
is subtracted from each row below the current row so as to make all other \( a_{ij} \)'s equal 0, where \( i \) is a column whose value is \( > j \). Processors work on every \( n \)th column of the same row simultaneously, where \( n \) is the number of processors available. Barriers provide synchronization at the end of each row. As the algorithm progresses, the amount of work available becomes smaller as the input matrix is reduced.

\[
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
\cdot & \cdot & \cdot & \cdot \\
0 & a_{22} & a_{23} & a_{24} \\
0 & 0 & a_{33} & a_{34} \\
0 & 0 & 0 & a_{44}
\end{pmatrix}
\]

Figure 4.5 GAUSS Algorithm

Figure 4.6 shows the Data Access Graph for a run of GAUSS on a 64-variable input matrix with 10 processors. The progression of the algorithm as columns of the matrix are eliminated can clearly be seen in the figure. A significant number of consistency misses appear because the algorithm is partitioned such that each processor will miss once per row when accessing the pivot element. The consistency misses are shown in Figure 4.6 in blue.
Figure 4.6 GAUSS Data Access Graph

Figure 4.7 GAUSS Load Balance Plot
Additionally, Figure 4.7 shows a load imbalance at the beginning of the simulation for processors with higher numbers. This problem decreases as the simulation progresses. Since the amount of work performed by each processor depends upon the processor's number and the amount of work per iteration is not constant, some processors will receive more work than others to perform. This results in processors remaining idle when there is clearly more work to perform.

To reduce the granularity of sharing of the global matrix and eliminate the load imbalance, we can block the data as was done in the SIEVE example above. Now, however, the regions of the matrix assigned to each processor must be continually re-evaluated since the amount of work diminishes as the algorithm proceeds:

```c
for(i = 0; i < mat->dim; i++) {
    if(my_pid == 0) {
        ...
        dist = (mat->dim - i)/numprocessors;
    }
    BARRIER(bar, numprocessors);
    for (j = i+1+my_pid*dist; j < mat->dim &&
         j < i+1+(my_pid+1)*dist; j++) {
        if(j < mat->dim) {
            op(i,j);
        }
    }
    BARRIER(bar, numprocessors);
}
```

As Figures 4.8 and 4.9 indicate, blocking the data again reduces the overall execution time for the program's execution. The load balance problem has been eliminated for all processors except processor 7. Because the number of columns to process in each row is not always a multiple of the number of processors available, processor 7 may have less work to do during some iterations.
Figure 4.8 Improved GAUSS Data Access Graph

Figure 4.9 Improved GAUSS Load Balance Plot
4.2.2 False Sharing

FFT

The Discrete Fourier Transform (DFT) is an important method for transforming a complex function in the time domain, \( h(t) \), to a complex function in the frequency domain, \( H(f) \). Consider a set of \( N \) consecutively sampled points in the time domain,

\[
h_k \equiv h(t_k), \quad t_k \equiv k\Delta, \quad k = 0, 1, 2, \ldots N - 1
\]

The DFT of such an input set is given by the following formula:

\[
H(f_n) = \sum_{k=0}^{N-1} h_k e^{2\pi i k n / N}
\]

The Fast Fourier Transform (FFT) algorithm reduces the amount of time needed to compute the DFT of a given input set from \( O(N^2) \) to \( O(N \log_2 N) \) by using the Danielson-Lanczos Lemma. This lemma states that the formula for a DFT can be written as a sum of two such formulae, one for even numbered points of \( N \) and one for odd numbered points of \( N \). These two formula can again be split into even and odd components, which can again be split, etc. Eventually we will have split the data down into transforms of length 1, whose transform is just the copying of the input value into the output slot. This method is known as the decimation-in-time or Cooley-Turkey algorithm.
Figure 4.10  FFT Execution Signature Graph

Figure 4.11  FFT Data Access Graph
Figure 4.10 shows the default Execution Signature Graph for an FFT execution with 16 processors and an input size of $2^{12}$ data points. The region at the beginning of the execution is dominated by stall time while waiting for access to the bus. Due to the partitioning of the algorithm, this massive contention for the data array is the main reason for FFT’s poor scalability, since all processors must access elements spread throughout the entire data array during the first portion if the algorithm. Figure 4.11 shows the Data Access Graph for this execution of FFT that confirms that some kind of problem exists. The blue squares mark false sharing accesses. The large number of false sharing misses wastes communication bandwidth as cache lines ping-pong between caches, driving up the bus stall time in Figure 4.10. The following statistics provided by ParaView give exact numbers for the degree of false sharing that this algorithm exhibits:

<table>
<thead>
<tr>
<th>Cache Num</th>
<th>Miss Rate</th>
<th>%Comp</th>
<th>%Conf</th>
<th>%Cons</th>
<th>%False</th>
<th>Num</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Misses</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.66%</td>
<td>0.00%</td>
<td>0.11%</td>
<td>0.07%</td>
<td>0.48%</td>
<td>368497</td>
<td>340215/28237</td>
</tr>
<tr>
<td>1</td>
<td>0.78%</td>
<td>0.19%</td>
<td>0.08%</td>
<td>0.03%</td>
<td>0.48%</td>
<td>347544</td>
<td>319294/28219</td>
</tr>
<tr>
<td>2</td>
<td>0.92%</td>
<td>0.21%</td>
<td>0.17%</td>
<td>0.03%</td>
<td>0.51%</td>
<td>337107</td>
<td>308855/28221</td>
</tr>
<tr>
<td>3</td>
<td>0.86%</td>
<td>0.23%</td>
<td>0.05%</td>
<td>0.06%</td>
<td>0.52%</td>
<td>338062</td>
<td>309772/28221</td>
</tr>
<tr>
<td>4</td>
<td>0.89%</td>
<td>0.24%</td>
<td>0.07%</td>
<td>0.05%</td>
<td>0.53%</td>
<td>334472</td>
<td>306200/28219</td>
</tr>
<tr>
<td>5</td>
<td>0.86%</td>
<td>0.22%</td>
<td>0.07%</td>
<td>0.05%</td>
<td>0.52%</td>
<td>337270</td>
<td>308993/28221</td>
</tr>
<tr>
<td>6</td>
<td>0.84%</td>
<td>0.23%</td>
<td>0.06%</td>
<td>0.03%</td>
<td>0.51%</td>
<td>340181</td>
<td>311928/28219</td>
</tr>
<tr>
<td>7</td>
<td>0.85%</td>
<td>0.23%</td>
<td>0.06%</td>
<td>0.04%</td>
<td>0.52%</td>
<td>340854</td>
<td>312588/28219</td>
</tr>
<tr>
<td>8</td>
<td>0.84%</td>
<td>0.21%</td>
<td>0.06%</td>
<td>0.05%</td>
<td>0.52%</td>
<td>341911</td>
<td>313632/28223</td>
</tr>
<tr>
<td>9</td>
<td>0.97%</td>
<td>0.20%</td>
<td>0.21%</td>
<td>0.03%</td>
<td>0.53%</td>
<td>335132</td>
<td>306872/28223</td>
</tr>
<tr>
<td>10</td>
<td>0.84%</td>
<td>0.19%</td>
<td>0.14%</td>
<td>0.02%</td>
<td>0.49%</td>
<td>337861</td>
<td>309618/28221</td>
</tr>
<tr>
<td>11</td>
<td>0.85%</td>
<td>0.19%</td>
<td>0.15%</td>
<td>0.02%</td>
<td>0.48%</td>
<td>340227</td>
<td>311986/28219</td>
</tr>
<tr>
<td>12</td>
<td>0.81%</td>
<td>0.20%</td>
<td>0.11%</td>
<td>0.02%</td>
<td>0.49%</td>
<td>342549</td>
<td>314312/28219</td>
</tr>
<tr>
<td>13</td>
<td>0.84%</td>
<td>0.20%</td>
<td>0.11%</td>
<td>0.04%</td>
<td>0.49%</td>
<td>342111</td>
<td>313848/28219</td>
</tr>
<tr>
<td>14</td>
<td>0.84%</td>
<td>0.21%</td>
<td>0.08%</td>
<td>0.04%</td>
<td>0.51%</td>
<td>335412</td>
<td>307150/28219</td>
</tr>
<tr>
<td>15</td>
<td>0.81%</td>
<td>0.19%</td>
<td>0.07%</td>
<td>0.04%</td>
<td>0.51%</td>
<td>342399</td>
<td>314139/28219</td>
</tr>
<tr>
<td>Avg:</td>
<td>0.84%</td>
<td>0.20%</td>
<td>0.10%</td>
<td>0.04%</td>
<td>0.50%</td>
<td>5461589</td>
<td>5009402/451538</td>
</tr>
</tbody>
</table>
In order to alleviate this false sharing problem, the main loop of the parallel procedure \texttt{pc.fast.fft} must be changed. In the original implementation, processors traverse the global data array by beginning with the element corresponding to the processor number and proceeding through the array accessing data elements that are multiples of the processor number. This partitioning ensures that processors access the same data elements during each iteration, thereby eliminating the need for synchronization between passes through the array. With a 32-byte cache line, four processors will access data in each cache line but only use one element in each. This fine-grained sharing is responsible for the high rate of false sharing. By giving each processor a contiguous section of the array to compute, we relieve the fine-grained sharing and eliminate the false sharing problem. However, this repartitioning also requires the insertion of an additional barrier since processors now share portions of the global array. Figures 4.12 and 4.13 depict the new algorithm. Figure 4.12 shows that by reducing the granularity of the sharing, the average bus stall time has dropped from the original implementation, and Figure 4.13 indicates that the false sharing problem has been eliminated. With this simple change to the code, we have achieved a 19\% reduction in parallel execution time despite the insertion of an additional barrier.
Figure 4.12  Improved FFT Execution Signature Graph

Figure 4.13  Improved FFT Data Access Graph
4.2.3 Contention for Shared Data Constructs

MP3D

MP3D solves problems involving the study of forces on bodies as they pass through the upper atmosphere at hypersonic speeds. Particles are moved through a sample space that contains the object under study. The particles' trajectories are computed according to a Monte-Carlo method and are subject to collisions with other molecules, the boundary of the space, and the object under study.

The principle data structures in MP3D are the cell array, which makes up the samples' study space, and the particle array, which contains information for each particle in the simulation. Processors are assigned particles to move through the sample space at the beginning of the execution. The entire cell array is shared by all processors during the course of the simulation. Figure 4.14 shows the Data Access Graph for cache 0 for a MP3D execution simulating 10000 particles in a 24x7x14 space for 10 time steps.

The vertical collision stripes are the accesses during each of the 10 time steps to the particle structure for each particle assigned to be moved by processor 0. Due to the large amount of data space required by MP3D, these accesses collide with each other and the cell array.

The wide band of consistency misses are to the cell array data structure. Although each processor is assigned certain particles to move through this cell array, there is no restriction on which cell a particle may move through. This situation leads to a lack of processor locality for accesses to the cell array. False sharing is also prevalent as the cell array is actively shared by all processors in the system. A different scheme for partitioning particles along with irregularly shaped portions of the space array
Figure 4.14 MP3D Data Access Graph

(depending on the expected flow of each particle) has been proposed by [15] as a possible way to alleviate the problem shown in Figure 4.14.

Figure 4.15 shows the **Load Balance Plot** for MP3D. Looking closely, it is apparent that all processors wait at barriers 1, 7, 13, etc. for processor 4 to complete. There are 6 barriers in the parallel portion of MP3D’s execution, with the second of these occurring at the end of the *move* procedure.
Figure 4.15  MP3D Load Balance Plot

Figure 4.16  Improved MP3D Load Balance Plot
During this procedure the molecules are moved through the active sample space, and 93% of the program's execution is spent in this procedure [50]. By examining the Data Access Graphs, it can be seen that accesses to one shared variable, clump, continually results in a cache miss by processor 4 throughout program execution. Checking the code, clump is indeed used during the particle move portion of the execution. Directing the loader to re-map this one variable leads to a significant reduction in the barrier waiting times for all processors, as seen in Figure 4.16. This change does not greatly affect overall program execution time in this small example, but when run for a realistic number of time steps, this minor change could have significant impact.

TSP

TSP implements a solution to the Traveling Salesman Problem in which a list of cities and the distances between them are given as the input, and the output is the shortest tour that visits each city exactly once, starting and stopping in the same city.

A priority queue of partially evaluated tours is kept as a heap that is sorted in inverse order of a lower bound of overall length. This sorted heap allows subtours most likely to be shortest to be evaluated first. If the shortest tour can be found near the beginning of the execution, subsequent tours will only be evaluated until they become longer than the shortest tour found so far, thereby reducing the time needed to eliminate longer tours.

Locks protect the priority queue, the variable containing the minimum tour found so far, the next tour to be evaluated, and the tour stack. Processes repeatedly remove a partially evaluated tour from the priority queue and recursively evaluate it, pruning the search space by using the global minimum tour length. To obtain a new tour, a process must acquire the priority queue lock, get the next tour, expand the tour
until it is short enough to solve recursively, and then release the lock protecting the priority queue. During the course of the expansion, processes must create new tours and place them on the priority queue. This sharing of the priority queue necessitates the lock protecting this data structure. Figure 4.17 shows the **Data Access Graph** for a TSP execution with a 13-city input and 4 processors.
Figure 4.17  TSP Data Access Graph

Figure 4.18  TSP Execution Signature Graph
There are three main data regions that demonstrate poor cache performance. The first of these, shown near the bottom of Figure 4.17, is the priority queue that maintains the partially evaluated tours. The second band of cache misses shows accesses to the tour stack. The final band is the data structure that contains the weights of each segment of the tours. As can be seen from Figure 4.18, the TSP algorithm exhibits a high degree of contention for access to the main data structures. The statistics provided by ParaView give the following cache-time breakdowns:

**Statistics on Cache Usage for TSP**

<table>
<thead>
<tr>
<th>Cache</th>
<th>Bus</th>
<th>Bar</th>
<th>Lock</th>
<th>Acq</th>
<th>Seq</th>
<th>Rel</th>
<th>Init</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num</td>
<td>Wait</td>
<td>Wait</td>
<td>Wait</td>
<td>Wait</td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
</tr>
<tr>
<td>=====</td>
<td>===</td>
<td>===</td>
<td>====</td>
<td>===</td>
<td>===</td>
<td>===</td>
<td>===</td>
<td>=====</td>
</tr>
<tr>
<td>0</td>
<td>3.8%</td>
<td>0.0%</td>
<td>29.1%</td>
<td>0.4%</td>
<td>20.0%</td>
<td>0.2%</td>
<td>0.0%</td>
<td>46.5%</td>
</tr>
<tr>
<td>1</td>
<td>3.7%</td>
<td>0.0%</td>
<td>29.9%</td>
<td>0.4%</td>
<td>19.6%</td>
<td>0.2%</td>
<td>0.0%</td>
<td>46.2%</td>
</tr>
<tr>
<td>2</td>
<td>3.7%</td>
<td>0.0%</td>
<td>30.0%</td>
<td>0.4%</td>
<td>19.9%</td>
<td>0.2%</td>
<td>0.0%</td>
<td>45.8%</td>
</tr>
<tr>
<td>3</td>
<td>3.9%</td>
<td>0.0%</td>
<td>29.9%</td>
<td>0.4%</td>
<td>19.8%</td>
<td>0.2%</td>
<td>0.0%</td>
<td>45.8%</td>
</tr>
</tbody>
</table>

As is evident by these statistics, a large amount of time is spent by each cache waiting for access to one of the locks protecting the shared data structures. The central work queue model gives rise to a high amount of contention for the lock protecting the priority queue and the tour stack. Processors spend 30% of their entire execution time attempting to access the locks that protect these data structures, and 20% executing critical section code (shown in the "Seq Time" column above). Clearly these structures must be distributed in some way to reduce the amount of contention in order to improve the performance of TSP.
4.2.4 Unnecessary Synchronization

Striped JACOBI

Striped JACOBI is another form of the JACOBI algorithm presented in Section 3.3. In this implementation, processors work on \( M/N \) contiguous rows of the matrix, where \( M \) = total rows in the matrix, and \( N \) = the number of processors. Processors calculate new values for each element in their stripe, and intermediate results are written to scratch data structures. When all calculations during an iteration have completed, all processors write the results contained in the scratch matrices back to the original shared matrix.

![Aggregate Cache Usage: JACOBI](image)

**Figure 4.19** Striped JACOBI Execution Signature Graph

This form of JACOBI does not suffer from the false sharing problems encountered in the red-black implementation and subsequently has better speedup numbers as evidenced in Figure 4.2. Nevertheless, ParaView can still be used to improve the algorithm's overall performance. Figure 4.19 is the **Execution Signature Graph**
of Striped JACOBI with 8 processors and a $128 \times 128$ input matrix. The five taller regions of *stalled for barrier* time correspond to the end of the computation phase of the iteration. When the last of the processors is done with computation, all processors are released from the barrier, and start to copy the computed values back into the matrix. When the new values are copied, the processors synchronize, and a new phase of computation can begin.

Figure 4.19 depicts a program characteristic that can be improved: the presence of the synchronization event between the compute and copy phases. We can improve performance by restructuring the algorithm to eliminate both the copy phase and the associated synchronization event. This restructuring is accomplished by storing the results of the computation phase directly into a second global matrix. After the iteration’s compute phase, the application can then use the result matrix as the source matrix for a new compute iteration, and the old source matrix can be used as the new result matrix. The results of a first attempt at this restructuring are shown in Figure 4.20.
Figure 4.20  Striped JACOBI Execution
Signature Graph with Two Global Matrices

Figure 4.21  Striped JACOBI Data Access
Graph with Two Global Matrices
Surprisingly, this implementation exhibits worse behavior. The Data Access Graph shown in Figure 4.21 shows where the problem lies: almost every access to either global matrix results in a collision miss.

The precise number of collision misses (and all other types of misses) is obtained from a text log file generated by ParaView during program analysis. This log indicates that the loader has placed the two matrices so that the direct-mapped processor cache maps the source and result matrices to the same cache lines. We can remove this inefficiency if we change the address of the result matrix. Figure 4.22 shows that a 30% reduction in execution time has been obtained from the original implementation when we instruct the loader to offset the two global matrices.

![Aggregate Cache Usage: JACOBI_without_Copying](image)

**Figure 4.22** Improved Striped JACOBI Execution Signature Graph with Two Global Matrices
Chapter 5

Implementation of ParaView

At the outset, the main design objective of ParaView was simply to perform an analysis of shared-memory programs to categorize shared data accesses as had been done in [5]. Additionally, a display to inform the user about the overall time a program spends performing synchronization was desired. We believed that a visual tool would help in this endeavor. The first step in the design process was to determine the format needed for the traces to be passed between simulators and ParaView.

5.1 Trace Format

In order to correctly classify data access patterns visually, the user must be able to distinguish different types of cache accesses. This need was the driving force behind the trace format eventually adopted for ParaView. An attempt was made to provide as much information as possible in each trace line, but to not force the trace to be so detailed that other less detailed simulation environments could not make use of it.

Important information about a single data access includes the type of access, the time the access occurred, what is being accessed, what the results of the access were, and when the access completed. Therefore the trace lines are structured to provide ParaView with this information. A trace file fragment is given in Table 5.1.

In Table 5.1, the first column contains the address being accessed for a particular trace line. The second column contains the unique number of the cache in the system fulfilling the request. For processor level caches, this number is the processor number.
<table>
<thead>
<tr>
<th>Address</th>
<th>Processor</th>
<th>Access Type</th>
<th>Bus Request</th>
<th>Bus Acquire</th>
<th>Bus Release</th>
<th>Hit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x9863188</td>
<td>2</td>
<td>1</td>
<td>6634</td>
<td>6691</td>
<td>6703</td>
<td>1</td>
</tr>
<tr>
<td>0x25ac</td>
<td>4</td>
<td>0</td>
<td>6634</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0x29ac</td>
<td>0</td>
<td>1</td>
<td>6634</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0x9863188</td>
<td>0</td>
<td>1</td>
<td>6635</td>
<td>6667</td>
<td>6679</td>
<td>3</td>
</tr>
<tr>
<td>0x9863188</td>
<td>5</td>
<td>6</td>
<td>6635</td>
<td>6643</td>
<td>6655</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.1 Trace File Fragment

For secondary caches, this number is just a unique identifier. The third column contains the type of access and can be one of the values indicated in Table 5.2.

<table>
<thead>
<tr>
<th>Access Type Value</th>
<th>Access Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Write</td>
</tr>
<tr>
<td>1</td>
<td>Read</td>
</tr>
<tr>
<td>6</td>
<td>Read-Modify-Write Success</td>
</tr>
<tr>
<td>10</td>
<td>Read-Modify-Write Failure</td>
</tr>
</tbody>
</table>

Table 5.2 Data Access Types

The fourth column in Table 5.1 indicates the time (in program cycles) at which the access request was presented to the cache. If the access resulted in a cache hit, this column also represents the time the access was serviced. If the cache is unable to fulfill the request immediately, columns five and six provide the cycle numbers at which the cache gained and released control of the bus, respectively.

Finally, the last column indicates the result of the access and may be one of the values shown in Table 5.3. Since the trace format is designed to be as flexible as possible, a simulation environment does not need to make use of all values in Table 5.3. For simple traces, a simulator would just need to choose one of the hit values for hits and one of the miss values for misses. The results presented will not
be as detailed, but ParaView can still be productively used even if this level of detail is not available.

<table>
<thead>
<tr>
<th>Hit Type Value</th>
<th>Hit Type</th>
<th>Qualifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Cache Hit</td>
<td>Hit to Unknown Data Type</td>
</tr>
<tr>
<td>0x1</td>
<td>Cache Miss</td>
<td>Compulsory Miss</td>
</tr>
<tr>
<td>0x2</td>
<td>Cache Miss</td>
<td>Collision Miss - Victim's State Unknown</td>
</tr>
<tr>
<td>0x3</td>
<td>Cache Miss</td>
<td>Consistency Miss</td>
</tr>
<tr>
<td>0x4</td>
<td>Cache Hit</td>
<td>Hit to Private Data</td>
</tr>
<tr>
<td>0x6</td>
<td>Cache Miss</td>
<td>Collision Miss - Victim’s State Invalid</td>
</tr>
<tr>
<td>0x8</td>
<td>Cache Hit</td>
<td>Hit to Shared Data</td>
</tr>
<tr>
<td>0xa</td>
<td>Cache Miss</td>
<td>Collision Miss - Victim's State Valid and Dirty</td>
</tr>
<tr>
<td>0xe</td>
<td>Cache Miss</td>
<td>Collision Miss - Victim's State Valid and Clean</td>
</tr>
</tbody>
</table>

Table 5.3 Hit/Miss Types Determined by Architecture Simulator

5.2 Trace Flow

Trace lines are analyzed by ParaView either concurrently with simulation or analyzed post-mortem from a trace file produced during execution. Figure 5.1 shows the path every trace line takes as it is processed by ParaView.

Trace lines with negative values in the type field are used to pass information to ParaView (see Section 5.4). These “special trace lines” do not contain any information on data accesses, but may inform ParaView of barrier information, initialization information, or information on dynamically allocated memory.

Data traces can be one of three types: private traces, shared data traces, or program text traces. Only shared data traces proceed to the false sharing analysis routines since text data should never be written. Next, hit and miss running totals for all three data trace types are kept for the Execution Miss Rate Plots for each 50000 cycle interval. If the access is a miss, the “stalled for bus” analysis for use in
the Execution Signature Graphs is performed. If the data access is not to private data, the routines for determining the other categories in the Execution Signature Graphs are called. These include lock and barrier time analysis. Finally the trace line is buffered for writing to file. All private references that result in cache misses are placed together in one file to reduce disk overhead. All text and shared data references are placed in files according to what group they belong, whether the access
is a hit or a miss, and whether the access occurred during initialization or the parallel portion of the code.

5.3 Grouping Data Objects

Since the primary goal of the initial effort was to provide a categorization tool, it was necessary to determine at what level level data would be categorized. That is, should users or the compiler attempt to label each cache line or each data object? Since users are most used to dealing with standard code constructs like integers, structures, etc., the preprocessor automatically groups the address space of the program by data object. The preprocessor uses the individual ‘.o’ files of the program to automatically extract the size of every global shared variable and text segment in the program. Therefore, the global variables must be uninitialized for their size to appear correctly. If a variable has to be initialized when declared, the user may place a file with the variable's size in it in the current directory, and ParaView will group the variable. Shared and text data's labels from each individual file are then matched with the corresponding label in the symbol table of the executable to obtain the range of addresses containing each data segment. ParaView produces a file, such as the one in Table 5.4, that provides the analysis tools with the range in the address space for each shared variable and text segment. In this way, the preprocessor can keep track of which accesses should be grouped together. By comparing the address contained in a trace line with those address ranges in Table 5.4, the preprocessor knows to which data object each address refers.
<table>
<thead>
<tr>
<th>Start Address</th>
<th>End Address</th>
<th>Variable Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>774560</td>
<td>774564</td>
<td>bc_count</td>
</tr>
<tr>
<td>774564</td>
<td>774568</td>
<td>gama</td>
</tr>
<tr>
<td>774568</td>
<td>774572</td>
<td>num_res</td>
</tr>
<tr>
<td>774572</td>
<td>774576</td>
<td>Ares</td>
</tr>
<tr>
<td>774576</td>
<td>774580</td>
<td>space_lengthm4</td>
</tr>
<tr>
<td>774580</td>
<td>774584</td>
<td>space_heightm4</td>
</tr>
<tr>
<td>181152224</td>
<td>181158687</td>
<td>Ares.d_0</td>
</tr>
</tbody>
</table>

Table 5.4  Example Grouping File

5.4  Passing Information from Simulators to Preprocessor

In addition to the trace lines representing data accesses, three special trace lines have been developed to pass necessary information from the simulator to the preprocessor. A special line has a negative value in the type field of the trace line, and values ranging from -1 to -4 mean the following things to the preprocessor:

<table>
<thead>
<tr>
<th>Type Field Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Passes barrier information to preprocessor</td>
</tr>
<tr>
<td>-2</td>
<td>Indicates end of initialization phase of program</td>
</tr>
<tr>
<td>-4</td>
<td>Passes dynamic memory information to preprocessor</td>
</tr>
</tbody>
</table>

Table 5.5  Special Trace Lines

5.4.1  Barrier Information

The current implementation of the preprocessor uses the first special trace line in Table 5.5 to give ParaView the address of the variable used to indicate when a process has entered or left a barrier. In the barrier initialization, a shared variable is declared that is read each time a process enters and exits a barrier. By informing ParaView of this variable's address, ParaView can look for accesses to this variable and compute
the amount of time spent performing barrier-related code. As discussed in Section 6.2, this detection scheme introduces minimal perturbation into the program under study.

5.4.2 Initialization Information

Many programs exhibit significantly different behavior during the initialization phase of execution, before any sub-processes are spawned. Often this phase of the program is concerned with the filling of global structures and is of little interest to most users. Therefore, the "initialization done" trace line allows simulators to inform the preprocessor when the end of the initialization phase occurs, so that data analysis and collection may begin. Alternatively, ParaView can be instructed to ignore this trace line and collect and display information on all data accesses, including data accesses during initialization.

5.4.3 Dynamic Memory Information

For programming languages like C that make use of dynamic memory allocation, a method must be used to link the memory allocated with the pointer requesting it. This match is achieved through the use of a trace line whose type field is set to -4. Upon encountering a global memory allocation call, the simulator constructs a trace line that tells the preprocessor the address of the pointer requesting the memory, the address of the beginning of the returned memory region, and the size requested. ParaView then determines if the pointer is shared or private. If shared, the dynamic region is added to the grouping file and named a derivative of the pointer's name. For example, the group in Table 5.4 labeled $Ares_d.0$ is the dynamically allocated piece of memory requested for the pointer $Ares$, which is also shown in Table 5.4.
5.5 Support for Other Architectures

ParaView can be used to provide average Execution Signature Graphs for any number of busses present in any multiprocessing system. Before trace analysis begins, ParaView searches for a file (bus_info) that contains text informing ParaView which caches are connected to which busses. For example, a bus_info file that contained

```
3
bus 0 1 2 3
bus 4 5 6 7
bus 8 9 10 11
end
```
tells ParaView that

1. There are three busses in the system to which caches are connected.
2. Bus 1 connects caches 0, 1, 2, and 3.
3. Bus 2 connects caches 4, 5, 6, and 7.
4. Bus 3 connects 8, 9, 10, and 11.

Averages are then calculated for each bus in the system to allow for analysis of many different architectures, such as a multiprocessor with hierarchical busses. If no bus_info file is found, ParaView assumes the system under study is a single-bus system.
Chapter 6

Evaluation

6.1 Analysis of the Results

ParaView has met the design goals determined at the beginning of the research: usability and functionality. ParaView's user interface provides an easy mechanism to aid the programmer in the complex task of performance debugging, and it takes little time to become acquainted with ParaView's various features. Furthermore, because of the long period of testing and revising, ParaView does a good job of pointing out inefficient areas of a program's execution quickly and effectively. The user is not required to examine a large number of statistics in order to determine the source of inefficiencies in a parallel application. Helpful information relating the graphics presented to the code being analyzed makes it especially easy to understand what the problems were and how to best remove them.

The most effective method of using ParaView has turned out to be a combination of several features. The Execution Miss Rate Plots and Load Balance Plots provide an excellent overview of the program execution and give high-level results as to the application's performance. Overt problems with cache performance and load balancing are quickly brought to the forefront, and the problems can then be looked at in greater detail with the Data Access Graphs and the Hot Spot Displays.

Figure 6.1 shows the improvement in performance of four of the parallel programs introduced in Section 3.3.
Figure 6.1 Comparison of Original and Improved Speedups

The speedups observed for SIEVE and FFT improved dramatically, especially as the number of processors increased. For SIEVE, superlinear speedup occurs at 12 processors because at that point, each section of the global array is small enough to fit entirely in each processors cache, eliminating most cache misses. Also, there is no explicit synchronization in SIEVE to limit the speedup obtained as the number of processors increases. The argument can be made that this superlinearity is due to the increased availability of resources (mainly available cache space), as the number of processors in the system increases. In order to make a fair comparison, the sequential version of SIEVE was run using an infinite cache for both the original and improved base case. The speedup for a 16-processor version was then recalculated using this result as the base for comparison. Table 6.1 shows that SIEVE achieves a near perfect speedup of 15.9 with 16-processors when the base case is that of an infinite cache. All other programs show minimal changes in the 16 processor speedup.
<table>
<thead>
<tr>
<th>Application</th>
<th>Cache Size</th>
<th>64KByte</th>
<th>Infinite</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>Original</td>
<td>8.31</td>
<td>8.1</td>
</tr>
<tr>
<td></td>
<td>Improved</td>
<td>11.0</td>
<td>10.7</td>
</tr>
<tr>
<td>GAUSS</td>
<td>Original</td>
<td>6.6</td>
<td>6.5</td>
</tr>
<tr>
<td></td>
<td>Improved</td>
<td>8.1</td>
<td>7.9</td>
</tr>
<tr>
<td>SIEVE</td>
<td>Original</td>
<td>3.0</td>
<td>2.1</td>
</tr>
<tr>
<td></td>
<td>Improved</td>
<td>21.9</td>
<td>15.9</td>
</tr>
<tr>
<td>STRIPED JACOBI</td>
<td>Original</td>
<td>14.3</td>
<td>13.3</td>
</tr>
<tr>
<td></td>
<td>Improved</td>
<td>13.4</td>
<td>12.8</td>
</tr>
</tbody>
</table>

Table 6.1 Adjusted Speedups for 16 Processors

For FFT, the false sharing observed in the original implementation only becomes worse with increasing numbers of processors. With more processors, a greater demand is placed on the global bus due to the fine-grained sharing of the problem. Repartitioning the data set reduces the demand on the global bus and improves the application's performance.

GAUSS also shows improved speedups, but the similarity of the two speedup curves suggest that the original limiting factor is still a problem. Even with the new data partitioning, most of the global matrix is actively shared among the processors. As the number of processors increases, the contention for the global bus will become a hindrance to scalability. This problem is made worse by the inherent inequity in workload division on each iteration. This load balancing problem grows with the number of processors.

Figure 6.1 indicates that striped JACOBI's speedups are actually worse for 16 processors after the synchronization has been removed. Because there are two global
matrices in the new version, there is increased contention for shared resources than when using the local scratch storage. However, this nonintuitive result points out one of the difficulties with only using one metric as a basis for comparison. Although Figure 6.1 indicates worse performance at 16 processors relative to 1 processor, the new version still takes less time to execute than the original implementation.

To illustrate this point, Figure 6.2 shows the ratio of the improved execution time to the original execution time for these 4 programs. In every instance, the improved
version of the parallel algorithms outperforms the original version regardless of how well the program scales. Even striped JACOBI, at 16 processors, requires only 75% of the time taken by the original version even though it exhibits a poorer speedup. As discussed above, Figure 6.2 confirms that the improvements to the programs become more pronounced as more processors are used, indicating that the improvements are actually helping rather than hurting scalability as might be inferred from Figure 6.1.

6.2 Limitations

Perturbations to Programs

ParaView has been designed to introduce as few perturbations as possible into the behavior of the parallel programs under study. Because of the separation of execution time into program and simulator execution time, it is far easier to remove perturbations in simulation environments than real systems. The few sources of perturbations in the ParaView simulation environment arise from the passing of the element on which processors spin at a barrier to ParaView, the signalling of initialization end, and the reporting of dynamically allocated data. All of these perturbations involve the construction of a single trace line to pass specific information to ParaView, thereby impacting the program's execution minimally. Furthermore, all appear during the initialization portion of a program's execution and do not alter the parallel execution of the algorithm.

In addition, ParaView requires the declaration of an extra shared variable to detect barrier entry and exit as described in Section 5.4.1. By detected reads to this variable, ParaView knows how long a processor spends performing barrier code. Perturbations to the parallel portion of a program's execution are therefore limited to the reading
of this shared variable twice per barrier instance per processor. This extra reading will add approximately 2 cycles per barrier instance.

There is one interesting source of perturbations associated with the RPPT simulator that is not present in Mpsas. The ParaView preprocessor may be run as a subprocess (forked by the simulator), or compiled directly into the simulator. The latter is substantially faster, as described below. However, because the RPPT simulator and user program are compiled together, the amount of stack space and dynamically allocated memory used by each affects the placement of the other. Thus, when ParaView's preprocessor is directly linked in with the simulator, the addresses of the user program’s stack space and heap are different than without the analysis tool present. This leads to different cache mappings, different cache miss rates, and different overall cycle counts. As an example, an execution of the FFT algorithm presented earlier ran for 8,919,607 cycles with the analyzer present, and 10,103,256 cycles without the analyzer. Thus the different mapping introduced by the preprocessor actually reduced overall program execution time by 12%! While it may be possible to circumvent this problem through clever loading of the simulator and analysis tool, a better solution is to fork the analysis program and pass trace lines through a pipe. Thus, ParaView's trace preprocessor has been altered to be a stand-alone program that is forked as a sub-process to the simulator. A pipe is then used to pass information from the simulator to ParaView. Since each module now has its own address space, the presence or absence of the preprocessor does not affect the simulated program’s execution.

Overhead Associated with ParaView

Overheads associated with ParaView arise due to the large amount of analysis performed by the trace preprocessor concurrent with simulation. Overhead numbers in
<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>Gauss</th>
<th>MP3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slowdown Factor With Analysis</td>
<td>5.04</td>
<td>3.74</td>
<td>5.22</td>
</tr>
<tr>
<td>File I/O</td>
<td>71.6%</td>
<td>70.2%</td>
<td>59.8%</td>
</tr>
<tr>
<td>Collecting Traces</td>
<td>17.7%</td>
<td>16.0%</td>
<td>17.8%</td>
</tr>
<tr>
<td>False Sharing Analysis</td>
<td>1.2%</td>
<td>.9%</td>
<td>1.1%</td>
</tr>
<tr>
<td>Cache Utilization Analysis</td>
<td>.2%</td>
<td>.4%</td>
<td>.6%</td>
</tr>
<tr>
<td>Other</td>
<td>9.3%</td>
<td>12.5%</td>
<td>20.2%</td>
</tr>
<tr>
<td>Added Overhead for Hit Trace Capture</td>
<td>44.0%</td>
<td>60.1%</td>
<td>20.5%</td>
</tr>
<tr>
<td>Reduced Overhead for Compiled Analysis</td>
<td>44.0%</td>
<td>35.0%</td>
<td>46.0%</td>
</tr>
</tbody>
</table>

Table 6.2 ParaView Overheads

this section were derived using the **RPPT** simulator. The base case invoked ParaView to analyze only cache misses during the parallel portions of the programs studied.

Table 6.2 indicates that the majority of the overhead can be found in the large amount of file operations ParaView performs during the course of analysis. ParaView continually updates files containing statistics on cache performance, synchronization, and false sharing. During a typical analysis, ParaView keeps track of approximately 100 - 130 files, depending on the number of shared variables present in the program under study. The files most frequently used remain open throughout the entire analysis to reduce time spent manipulating file pointers.

The time the simulator spends actually collecting traces for analysis by ParaView makes up the second highest percentage of overhead. Approximately 17% of the overhead is spent filling trace structures, sorting trace lines in time-order, and passing the traces to ParaView.

As expected, false sharing analysis and cache utilization analysis add very little overhead to the simulation. Routines associated with these analyses are only called if the trace line contains a shared data reference. Additionally, the most expensive
portion of the false sharing analysis involves linked-list manipulation that only occurs if the present access is a consistency miss.

The rest of the overhead associated with ParaView lies in the manipulation of access buffers, decoding trace lines, maintaining miss rate statistics, and classifying data accesses.

As Table 6.2 indicates, if information on cache hits is maintained as well as information on cache misses, the overhead could increase by as much as 60%. This overhead varies widely depending on the cache hit rates for a given program. By eliminating cache hit analysis, overhead shrinks along with the amount of disk space needed to store information. For example, one trace line requires 24 bytes of storage, and the execution of MP3D presented in Section 4.2.3 made 47 million data references. Had all references been kept, $47 \times 24 = 1.1 GB$ bytes would have been required just to store the information for the Data Access Graphs, as opposed to the 2% of that figure needed to store only cache miss information.

Finally, the ParaView trace preprocessor may be compiled into the simulator directly or forked as a sub-process. Table 6.2 shows that compiling in the preprocessor can reduce overhead by 35-46%. However, this method suffers from the data perturbation problem discussed above when used in conjunction with RPPT, but can be employed without side effects when used with Mpsas since Mpsas maintains a distinct separation between the simulator and the user application.

6.3 Areas for Improvement

There are several aspects of the ParaView system that can be improved. First, although the trace line format derived for this initial implementation of ParaView captures all information needed for a bus-based system, other trace fields may become
necessary to allow ParaView to be used with systems such as distributed-memory systems and directory-based shared-memory systems. For examples, in a distributed system, reporting causes for page faults may be more effective than cache hit and miss rates. Additionally, other graphical system and data displays may be needed that would aid programmers in analyzing programs written for such alternate multiprocessing systems. Other displays might include message queue lengths, usage of other system components such as networks, or load balancing between nodes of a massively parallel system as well as load balancing between individual processors. ParaView has been written to be modified and added to easily, thereby allowing additions to both the trace line format and the available graphical displays.

Two other areas of improvement are the result of large amount of I/O required to save trace files for post-simulation viewing: disk space and speed. Although the large amount of disk space required for trace files can be reduced by saving just cache misses, a parallel program with real data sets could require several gigabytes to store references to cache miss data alone. Additionally, the slow speed of ParaView's graphical displays is also a direct reflection of the speed of the I/O devices on which the results are stored. Both of these problems could be alleviated by making ParaView's graphical displays concurrent with the simulation, saving just what is immediately needed and saving little or nothing for post-mortem analysis. By storing temporary results in memory rather than on disk, both the time required to access the data and the disk space requirements would diminish significantly. This solution is an attractive one, except for the fact that a simulation of a parallel program could take many hours to execute, requiring the user to be present for the entire simulation run. In a real system, the time factor may not be as restrictive, but the slow speed of the graphical displays and the large amount of data produced will greatly hinder concurrent analysis and simulation.
Chapter 7

Conclusion

7.1 Summary

Performance debugging is an essential part of the development of efficient parallel programs. Problems with poor cache performance, inefficient synchronization, false sharing, load imbalance, and fine-grained sharing can contribute to less than ideal application performance. General performance indicators, such as speedup numbers and execution times, indicate the relative performance of an application but do little in aiding a parallel programmer in uncovering the source of inefficiencies.

We have described the development and use of ParaView, a tool designed to aid parallel programmers in performance debugging. Our goal was to provide users with an interactive, X-driven tool that would facilitate the understanding and correcting of inefficiencies in parallel applications. A simple trace format developed specifically for use with ParaView allows any simulation environment to provide users with detailed knowledge of program performance through the visualization of accesses to data structures. Additionally, ParaView provides methods for the analysis of cache utilization, load balance, and the variation of cache miss rates over the course of execution of a parallel program.

ParaView provides an effective means of understanding common parallel programming problems. Through an in-depth analysis of 5 parallel programs, we used ParaView to show problems with poor cache mapping, false sharing, contention for shared constructs, and inefficient synchronization. Reduction in execution times for
the 16 processor simulations ranged from 77% for the SIEVE application to 18% for GAUSS. Because of ParaView's ability to point out quickly where programming problems lie, the time required to spot and correct inefficiencies was minimal and further underscores the need for tools like ParaView to aid in performance debugging.

7.2 Future Work

There are many avenues available for further research into the issue of performance debugging. In the immediate future, ParaView needs to be extended to provide capability to analyze not only shared-memory systems but also distributed systems. Additionally, architectural features such as split-transaction busses and weaker consistency models should be supported by ParaView. Finally, effects of I/O upon system performance should be made visible by ParaView.

ParaView represents a step toward displaying aspects of parallel program behavior that can point to where possible inefficiencies may lie. Actually analyzing the performance of a program and perhaps making suggestions to the programmer on what could be done to improve performance is an intriguing research topic. This process would entail the analysis of a vast quantity of parallel applications, the development of a set of common problems, and the derivation of methods to detect and correct them. Basically, the entire performance debugging problem could be made transparent to the user, as long as modifications to the basic algorithm were not required. This system would again need to be versatile enough to be used equally well on all multiprocessor architectures.
Bibliography


