2004

Infrastructure For Performance Tuning MPI Applications

Kathryn Marie Mohror
Portland State University

Recommended Citation

Mohror, Kathryn Marie, "Infrastructure For Performance Tuning MPI Applications" (2004). *Dissertations and Theses.* Paper 2660.
https://doi.org/10.15760/etd.2661

ABSTRACT

An abstract of the thesis of Kathryn Marie Mohror for the Master of Science in Computer Science presented November 13, 2003.

Title: Infrastructure For Performance Tuning MPI Applications

Clusters of workstations are becoming increasingly popular as a low-budget alternative for supercomputing power. In these systems, message-passing is often used to allow the separate nodes to act as a single computing machine. Programmers of such systems face a daunting challenge in understanding the performance bottlenecks of their applications. This is largely due to the vast amount of performance data that is collected, and the time and expertise necessary to use traditional parallel performance tools to analyze that data.

The goal of this project is to increase the level of performance tool support for message-passing application programmers on clusters of workstations. We added support for LAM/MPI into the existing parallel performance tool, Paradyn. LAM/MPI is a commonly used, freely-available implementation of the Message Passing Interface (MPI), and also includes several newer MPI features, such as dynamic process creation. In addition, we added support for non-shared filesystems into Paradyn and enhanced the existing support for the MPICH implementation of MPI. We verified that Paradyn correctly measures the performance of the majority of LAM/MPI programs on Linux clusters and show the results of those tests. In addition, we discuss MPI-2 features that are of interest to parallel performance tool developers and design support for these features for Paradyn.

INFRASTRUCTURE FOR PERFORMANCE TUNING MPI APPLICATIONS

by

KATHRYN MARIE MOHROR

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
2004

ACKNOWLEDGEMENTS

I thank Dr. Karen L. Karavanic for her guidance, patience, support, and encouragement at every turn in this endeavour. Without her, I would not have achieved so much.

I also thank my other committee members, Dr. Bryant York and Dr. David McClure. Dr. York gave considerable feedack on this work and provided me with inspiration and direction for future work. I thank Dr. McClure for his counsel over the years and for understanding my desire to leave Chemistry and pursue a degree in Computer Science.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ADI	Abstract Device Interface
APART	Esprit Working Group on Automatic Performance Analysis: Resources and Tools
ASCI	Advanced Simulation and Computing Program
CHAOS	Cluster High Availability Operating System
GER	Guaranteed Envelope Resources
HPC	High Performance Computing
LAM	Local Area Multicomputer
LLNL	Lawrence Livermore National Laboratory
MCR	Multiprogrammatic Capability Resource
MPD	Multi-Purpose Daemons
MPE	MultiProcessing Environment
MPI	Message Passing Interface
NASA	National Aeronautics and Space Administration
NCSA	National Center for Supercomputing Applications
PC	Performance Consultant
PNNL	Pacific Northwest National Laboratory
PVM	Parallel Virtual Machine
RMA	Remote Memory Access
RPI	Request Progression Interface
SMP	Symmetric Multi-Processor
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

1 Introduction

The goal of this thesis is to strengthen the parallel performance tool base for MPI programmers on Linux clusters. This work is important for several reasons. First of all, Linux clusters are rapidly gaining popularity as supercomputing platforms. They are useful for testing software intended for the more expensive and specialized supercomputers, as well as for computing real programs themselves. Secondly, there is a significant lack of software tools, including parallel performance tools, to help programmers on supercomputers complete their work efficiently and correctly. The scientists dependent upon the results of programs run on these platforms need such tools, so that they can develop applications more quickly, and spend less time optimizing their code. Thirdly, performance tuning MPI applications is important because MPI is commonly used to write scientific programs. These programs will continue to be used in the future because rewriting them in a newer parallel programming paradigm is not likely to be cost-effective.

This introduction will elaborate on these points by first discussing the history and status of Linux clusters as supercomputers. Then, it will expound upon the lack of software tools for such platforms and explain why more work needs to be done in this area. Following that discussion, the importance of MPI in scientific programming, and the newer features of MPI that lack performance tool support will be examined. Last, the specific goals of this thesis will be outlined and the steps for achieving those goals will be presented.

In 1994, the first cluster of commodity parts was built at NASA's Goddard Space Flight Center. It was the result of price and performance constraints placed on the scientists there [SSB+99]. Now, less than ten years later, clusters of workstations running the Linux operating system are gaining serious recognition as some of the fastest systems in the world. In June of 2003, a Linux cluster made with commodity processors was ranked as the third fastest system in the world with a peak computing speed of 11 teraflops, according to the June 2003 Top500 Supercomputer Sites list [Top503]. These systems are quickly losing their reputation as simply a low-cost alternative for high-performance computing, and are fast becoming the system of choice to fulfill high performance needs.

A key effort towards the development of supercomputing-caliber Linux clusters is found in the Department of Energy's national laboratories. Publications from Lawrence Livermore National Laboratory (LLNL) give reasons for moving in this direction. "The Linux Project Report" from LLNL argues that the low price/performance ratio when using commodity or near-commodity parts, compared to the expense of purchasing and maintaining proprietary hardware, make Linux clusters more attractive high-performance alternatives [GD02]. The report also says that the open-source nature of the operating system is appealing because it can be tailored in-house to meet the specific needs of high-performance computing (HPC). A paper outlining the CHAOS project at LLNL states that another advantage of clusters is overall increased availability and manageability of the system as compared to proprietary systems [BGG03]. In a press release in September of 2002, Bill Feiereisen, the leader of Los

Alamos' Computer and Computational Sciences Division, speaking of the Linux cluster called the "Science Appliance," said:

> "Future supercomputers must be cost-effective, efficient and easy to enhance and scale. Scalable supercomputing systems that run proprietary operating systems clearly are a thing of the past. Instead of buying a complete proprietary computing system, we are looking toward a future in which a robust set of integrated, open source software tools enables us to assemble a truly scalable supercomputer from components that best meet our needs." [Dann02]

The cost-effectiveness of these systems is evident when you consider the price of some of the other supercomputers on the Top500 Supercomputer Sites list. For instance, the proprietary Hewlett-Packard system, ASCI Q, housed at Los Alamos National Laboratory, and ranked second on the June 2003 Top500 list, cost $215 million to build [LANL02]. The fourth ranked supercomputer, ASCI White, a proprietary system built by IBM and located at LLNL, required a contract of $110 million [Schw01]. In contrast, the Linux cluster, Multiprogrammatic Capability Resource (MCR) at LLNL, which ranked third in the June 2003 list, cost under $14 million. The LLNL 2002 Annual Report states that the MCR cluster, when regarded in terms of operations per dollar, is number one among its supercomputers [LLNL02]. At Pacific Northwest National Laboratory (PNNL), the fastest Linux cluster in the world to date at 11.8 teraflops, and the fastest unclassified supercomputer in the United States, cost only $24.5 million. When ranked by peak performance, the PNNL cluster is the fifth fastest supercomputer in the world [Malo02].

A contributing factor in lower costs for these systems is the use of the freely-available, open-source operating system, Linux. "The Linux Project Report" from

LLNL argues several other advantages to using open-source software besides initial cost. When using proprietary software, if an error or other trouble is found, the users must petition the software vendor for fixes. With open-source software, it is no longer necessary to make a feature request to an external company, not knowing when, or even if, it will be implemented. The changes to the software can be made in-house, keeping the users' needs in mind. Also, the use of the same operating system on different supercomputers means that if operating system modifications can be made portable to different hardware, the HPC improvements can be freely shared for use on other clusters. The report states that the sharing of open-source software has proved to be beneficial to the Linux Project and also to the Linux cluster community at large [GD02]. The result of this sharing is the CHAOS Linux distribution, used as the operating system for clusters at LLNL.

Another factor that determines overall cost of a supercomputer is the ongoing expense of maintaining the system. Experiences at LLNL have shown that Linux clusters are generally more available and easier to maintain than their proprietary counterparts [BGG03]. Availability in computer systems refers to the amount of time that the system is ready and accessible to users. The researchers at LLNL state that because clusters are made from commodity or near-commodity parts, it is more cost-effective to keep spares on hand in case of part failure. Also, the very nature of clusters make the system independent of individual part failure; the failure of one node in the cluster does not necessarily affect the running capability of the cluster. The maintenance of cluster systems can be simpler than that of proprietary systems. Researchers at LLNL compared some routine tasks performed on the MCR cluster and ASCI White and found some surprising results. For instance, a reboot of the MCR cluster took about thirty minutes, whereas ASCI White took five hours for a reboot. A complete reinstall of the MCR cluster required about eighty minutes, while the same maintenance operation on ASCI White lasted about one week [BGG03].

There is a great need for system software to support programmers of applications for use on supercomputing systems. System software is any software that supports application programs but is not specific to any particular application and typically includes the operating system, user environment software, development tools, debugging, profiling and monitoring tools, and utility programs. The National Science Foundation finds the need for support software for users to be urgent and recommends that more work be done to develop software tools for supercomputing platforms. They have found that software tools are necessary for computational researchers to complete complex and innovative work [NSB03]. The researchers in the Linux Project at LLNL report that there is a need for system software to support Linux clusters [BGG03]. Baden points out the multitude of difficulties that scientific programmers have on these types of systems; they must manage shared-memory, parallelism, locality in the application, processes, and message-passing. He points out that the lack of software tools to help with these problems hinders efficient implementations of application programs [BF00].

An example of system software for which there is a deficiency on supercomputing systems is parallel performance tools. A parallel performance tool is a software tool that helps application programmers understand the performance issues of their programs. One reason these tools are important is because it is quite difficult to achieve the peak advertised performance of parallel systems. In general, the actual performance of the system can be orders of magnitude less than its peak advertised performance. According to a communication referenced by Gordon Bell, clusters usually deliver 5-15% of their peak advertised performance [BG02]. It is likely these numbers could be improved through performance tuning of applications. Dan Reed, Director of the NCSA, states that many users view performance optimization as an “unavoidable evil” because existing performance tools are not intuitive to use [RAD+98]. Reed expects the need for research in this area will increase as the complexity of the systems for parallel programming will increase.

Evidence of the need for performance tool research can be found in a recent study that looked at communication behavior in message-passing programs. In that study, the researchers were forced to limit the problem size for one of their experiments because the trace files generated by the performance tool they were using grew to be unmanageable in size [VM02]. In another study, the authors pointed out that for only a 48 task run, the binary tracefile that was generated by their performance tool was 225 MB [Vett02]. Reed estimated that instrumentation to record function entry and exits on a parallel system with hundreds of processors could easily generate a data volume of 10 MB/second [RAD+98]. This issue of problematically large data files generated by performance tools is likely to be more important in the future as supercomputers become capable of running even more concurrent processes and thus generate even more performance data. Another example of the need for performance tool research is from Portland State University. A research project was stalled and could not continue because there wasn't an existing performance tool that could fulfill its needs [Kear03]. The goal of the study was to examine the performance differences in two different MPI implementations on Linux clusters. When it came time to do more detailed analysis so the researchers could fully explain the performance differences they saw, they were unable to find a performance tool to help them. The product of this thesis will allow that project and other similar projects to continue.

The Message Passing Interface (MPI) emerged as a standard in 1994 as MPI-1.2 and was widely accepted by the scientific programming community. There are several features of MPI that account for its widespread acceptance and use. One is that MPI permits efficient implementations of the interface regardless of machine characteristics. Another selling aspect of MPI is that it allows for MPI programs to be run transparently on heterogeneous systems. Yet another feature is that the same MPI source code should be able to run without change on different computing platforms, given that MPI libraries exist on those platforms. The interface was attractive to parallel programmers and was used to build many scientific applications. Even though there are new parallel programming paradigms today that are arguably easier for application programmers to use, such as data parallel languages and parallelizing compilers, the cost of translating legacy MPI code to a newer paradigm can be prohibitive and would in some cases require a complete rewrite of the code [BKS+00]. For this reason, it is arguable that performance tuning for MPI applications will continue to be important in the future.

In 1997, another version of MPI was released that extends the functionality of the original interface. This version is called MPI-2. Some of the new features this version provides for are parallel file access, dynamic process creation, and one-sided communications. Among the freely-available MPI implementations, complete support for the MPI-2 Standard has not yet been achieved. However, the LAM/MPI implementation supports most of the new standard. There is little performance tool support for these new features, likely because the MPI implementations had not yet provided for them, so there wasn't much demand. Interest in performance tuning MPI-2 features will likely increase as the MPI implementors provide support for the standard. Application programmers may adopt the new features as the performance of their programs can be increased. For instance, NASA's Goddard Space Flight Center reports a 39% improvement in throughput after replacing MPI-1.2 non-blocking communication with MPI-2 one-sided communication in a global atmospheric modeling program [PCL+02].

The goal of this thesis is to strengthen the parallel performance tool base for MPI programmers on Linux clusters. To achieve our goals, we chose to increase the level of support for MPI into an existing parallel performance tool, Paradyn [MCC+95]. The specific contributions of this thesis are: the implementation of support for the MPI-1 features of LAM/MPI into Paradyn; the implementation of support for non-shared filesystem clusters into Paradyn; the investigation of items of interest in

MPI-2 for parallel performance tool developers; and the design of support for MPI-2 features into Paradyn.

The steps necessary for achieving the thesis contributions were to: understand Paradyn and LAM/MPI startup procedures; make necessary changes to Paradyn to accommodate LAM/MPI's startup needs; verify that existing Paradyn functionality for MPI-1.2 correctly reports values for LAM/MPI; and identify/design necessary changes to Paradyn for support of MPI-2 features

The choice was made to use the Paradyn Performance Tool instead of creating a new tool to serve this purpose for Linux clusters. Paradyn is an established research tool with appealing characteristics. Paradyn is freely available, well-documented, and relatively easy to use. Paradyn already supported the MPI implementation, MPICH. A primary feature of Paradyn is its Performance Consultant, which drills down automatically into the user's program and finds performance bottlenecks. Paradyn also supports dynamic instrumentation, which is the insertion of performance measurement instructions into a running program. This dramatically decreases the amount of data that must be collected over the course of the program and is a convenience feature for the application programmer. Paradyn was developed to aid in solving grand challenge problems and is used as a research tool in national laboratories.

Chapter 2 of this document reviews the background information necessary for understanding this thesis. Chapter 3 covers the specifics of the Paradyn Performance Tool and its initial level of support for message-passing on clusters of workstations. Chapter 4 is a literature review of related work. Chapter 5 discusses the preliminary changes that were necessary to make Paradyn operational for message-passing on a cluster of workstations with a non-shared filesystem and outlines the alterations to Paradyn that enabled performance measurement of LAM/MPI applications. In Chapter 6, we identify items of interest to performance tool developers in MPI-2 and design support for MPI-2 for Paradyn. Chapter 7 shows and examines the results of various tests of the changes in Paradyn. In Chapter 8 we conclude and discuss future work.

2 Background

This chapter provides some of the background information needed to understand our work. Section 2.1 gives some basic information about clusters of workstations and also discusses their advantages and disadvantages. Next, Section 2.2 describes the parallel programming paradigm, message-passing, and the Message Passing Interface (MPI). Section 2.3 outlines the additions to MPI from MPI-2 and explains some details of a few important features. Section 2.4 provides background information about the MPICH implementation of MPI. Finally, Section 2.5 discusses the LAM/MPI implementation of MPI.

2.1 Clusters of Workstations

A cluster of workstations is a group of complete computers that are connected by a communication network and are able to work together as a single unit [HX98]. The computers that make up the cluster are individually known as *nodes*. The collection of nodes are said to make up a *loosely-coupled* system, in contrast to a *tightly-coupled* one in which the processors in the system are directly associated with the multiple memories. In other words, in a tightly-coupled system, the different processors are able to use high-speed communication mechanisms, such as shared memory. The interconnects between the processors in tightly-coupled systems are generally proprietary. In the loosely-coupled cluster, the processors on different compute nodes must use another method of communication, such as TCP sockets over Ethernet, which is much slower than shared-memory communication. Note that a cluster could be a loosely-coupled collection of nodes, where each node contains multiple processors that are tightly-coupled.

The appeal of cluster systems is multi-fold, and largely stems from their composition of commodity, off-the-shelf parts. In general, commodity parts have very low price/performance ratios. For example, we can compare the relative peak compute speeds and costs of ASCI White, a tightly-coupled supercomputer of proprietary IBM design, and the MCR Linux cluster, both of which are at Lawrence Livermore National Laboratory. The price/performance ratios are \$8.93 million/teraflop for ASCI White versus \$1.27 million/teraflop for the MCR cluster.

Another advantage of a cluster of commodity computers is that these types of systems can be upgraded relatively easily. For instance, because the compute nodes are made from commodity parts, some of those parts could be replaced with newer components. This type of upgrade was performed at Pacific Northwest National Laboratory on its Linux cluster, taking its peak compute speed from 6.2 teraflops to 11.8 teraflops [McMi03].

Yet another appeal of clusters is the ease of providing spares in case a compute node should fail, adding to the overall increased availability of the system. The idea is that the cluster is not dependent upon any one node. Upon a particular node's failure, it can be replaced or the work load can be redistributed among the remaining compute nodes. In contrast to this, if the shared memory of a large symmetric multi-processor (SMP) fails, the entire system will be brought down.

While the advantages of cluster systems are compelling, there are drawbacks which largely involve the loosely-coupled nature of the systems. Perhaps the largest difficulty that arises is how the compute nodes should communicate in order to emulate a shared-memory environment between the processors on different nodes. The difficulty lies in making this emulation as efficient as possible to minimize the incurred overhead. The common choices for achieving this are to use a software layer to emulate distributed shared memory, use remote procedure calls, or use message-passing. This type of communication is most easily accomplished using a message-passing model, whereby the nodes share information with each other by passing messages. Another model, most suited to the object-oriented programming paradigm, is a distributed object approach. In this model, different objects in the program would reside on different nodes of the cluster. One object would utilize a remote procedure call to invoke a method of another object on another node, thereby distributing the workload.

The experiments for this thesis were run on the Wyeast Cluster in the High Performance Computing Lab at Portland State University. Wyeast consists of forty-eight compute nodes. Each compute node is a symmetric multi-processor machine comprised of two 866 MHz processors. The nodes each have two Fast Ethernet network cards and are connected by two identical switches, one for each set of network cards. This setup could allow traffic to divide across the two networks, potentially doubling the network speed. Each compute node has 512 MB of SDRAM. The operating system on each node is Linux RedHat 7.2, kernel version 2.4.7-10smp.

2.2 Message-Passing and the Message Passing Interface (MPI)

In the message-passing model, the separation of the address spaces of the processes on different compute nodes is plainly visible to the application programmer. Separate processes are not capable of transparently manipulating or reading each other's variables. The processes must execute explicit send/receive or read/write operations if the sharing of data is necessary. The cooperation of all processes involved in the exchange is required. The programmer must also resolve any interaction issues, such as mapping the data across the compute nodes and the synchronization of computation and communication.

The Message Passing Interface (MPI) is the leading standard for the message-passing model [SOH+99]. It was designed to be portable to a wide-variety of parallel computing systems, including clusters of workstations. Version 1 of this interface, MPI-1, was released in 1994 and Version 2, MPI-2, was released in 1997 [MPI03]. In the early 1990's, before the design of MPI, several message passing libraries had been developed. Some examples of such libraries are: PVM [GBD+94], P4 [BL94], Chameleon [GS93], Zipcode [SSD+94], Express [FK94], and PARMACS [CHH+94]. The development of MPI was inspired and influenced by these message-passing implementations. The designers of the MPI Standard sought to preserve the desirable features of the existing implementations and to avoid the pitfalls uncovered by these earlier works [SOH+99].

Key goals of the MPI Forum were: to provide a degree of portability across hardware platforms, to give the ability to run transparently on heterogeneous systems, and to allow efficient implementation on machines with different characteristics [SOH+99]. The first of these goals means that the same MPI source code should be able to run on different platforms, given that MPI libraries exist on those platforms. The second says that a single MPI program should be able to be run across heterogeneous systems, or collections of processors with different architectures. The conversions necessary for the internal representation of datatypes of the system is done implicitly by the MPI implementation. The last goal is reflected in that the MPI Standard does not specify how its communication operations will take place, but simply defines the semantics of those operations. This allows implementors of MPI to carry out the operations in the most efficient way possible for each system. These aims were reached and are primary factors in the continued popularity of MPI. Implementations of MPI exist for nearly all major computer platforms and have language bindings for C, C++, and Fortran. These achievements mean that, for the most part, an MPI program written in one of these languages can be run without change on different platforms. Implementations of MPI support the Standard to varying degrees. In this work, we focus on two freely-available implementations, LAM/MPI and MPICH. In general, the degree to which the implementation supports the MPI Standard is publicly available. However, it is possible to attempt to validate a particular implementation with MPI validation suites. A list of freely-available validation suites can be found on Argonne National Laboratory's MPI web page [ANL03].

An MPI program is made up of one or more MPI processes that can run either on one machine or across multiple machines. A process in MPI is defined by a (group,rank) pair. A *group* refers to a collection of processes that share an intracommunicator. An *intracommunicator* is an identifier that defines the group and is generally used for communication purposes within the group. Within each group, or intracommunicator, each process is assigned a rank. This *rank* is an identifier of the process with respect to the intracommunicator. It is important to note that a process can belong to more than one group and can thus be defined by more than one (group,rank) pair. While a (group,rank) pair uniquely defines a process, a process does not define a unique (group,rank) pair. An MPI programmer is able to create new intracommunicators at runtime if subdivisions of groups are desired. It is also possible for two intracommunicator groups to establish an *intercommunicator* between them. Another possibility allows the merging of the groups sharing the intercommunicator into one new intracommunicator. The generic term for an intercommunicator or intracommunicator, when the specific type does not matter, is *communicator.*

In general, data in MPI programs is exchanged between processes by explicit send and receive operations. There are several variations of these operations, including blocking and non-blocking functions. According to definitions in the MPI specification, a *blocking* send or receive call is one that does not return until the arguments used in the call can safely be reused. For the send call, this means that the function could return even if the matching receive has not actually finished or even received the data; it merely means that, upon the return of the function, the programmer is free to reuse the send buffer without fear of corrupting the data that is intended to be sent. For the blocking receive call, the function return guarantees that the receive buffer will hold the data that was sent, regardless of whether the matching send operation has completed. A *non-blocking* operation in MPI is one that returns immediately, whether or not the send or receive buffers can safely be reused. It is up to the MPI programmer to make sure that the operations have completed with explicit test and wait MPI function calls. The test calls will inform the programmer as to whether the send or receive has finished. The wait calls will block until the operation in question has finished. While non-blocking operations could potentially decrease the overhead of communication by allowing the overlap of computation and communication, their use causes increased complexity for the MPI programmer.

In MPI, there are point-to-point communications and collective communications. *Point-to-point* communication refers to the exchange of data between a pair of MPI processes. Point-to-point operations can be blocking or non-blocking. The senders and receivers of these operations are identified by their rank with respect to a communicator. *Collective* communication refers to the simultaneous exchange of data between a group of processes that share a communicator. There are many types of collective operations including barrier synchronization across all members of the group, gathering data from all members of the group to one member, scattering data from one member to the group, and reduction variations of the scatter and gather methods that perform an arithmetic operation on the data, such as a sum, max, min, or a user-defined function. The number of MPI routines for collective communications is quite extensive.

2.3 MPI-2

This section describes the MPI-2 additions to MPI. First, we give an overview of what is new in MPI-2. Afterward, we discuss what we consider to be the most important additions to MPI-2 from the perspective of this work.

The MPI Forum released MPI-1 knowing that they had omitted several topics important to parallel programming [GHL+98]. Their motive was to release the first version so that people could start using it, because they knew it was going to take a while for them to design the rest of it properly. The major additions to MPI from MPI-2 are dynamic process creation, parallel I/O, and one-sided communication. MPI-2 also defines thread support for MPI programs, explains the semantics of collective communication over intercommunicators, and provides methods for establishing communication between non-related MPI processes and applications. Other contributions from MPI-2, are mixed language convenience features, standard C++ bindings, recommendations for using MPI with Fortran90, and clarification of ambiguities in the MPI-1.2 Standard [MPI03].

Another addition to MPI from MPI-2 is the Info object. This object is a parameter to many of the new MPI-2 routines. It is a variable length string of (key,value) pairs. The content in the string is MPI implementation and platform dependent. It is intended to be a way for the programmer to provide information to an MPI implemenation. Even though this feature decreases the portability of MPI programs, the Forum felt the addition was necessary so that the MPI implementations could optimize the new functionalities of MPI-2. For instance, the Info argument to the MPI-I/O routines can be used for specify file access patterns to the MPI implementation, so that it can potentially optimize file manipulation.

2.3.1 Process Management

MPI-1 says that MPI programs consist of a fixed number of processes all started simultaneously. In other words, the number of processes in an MPI-1 application is determined at the beginning of the application and cannot be changed afterward. Some parallel programmers find this restrictive and desire the ability to dynamically create and terminate MPI processes at runtime. In designing the interface for dynamic process creation, the MPI Forum sought to maintain the platform independence of the Standard. To do this, they chose to not address resource control in the interface. Examples of resource control the Forum did not define were the addition and removal of nodes in the parallel virtual machine, the reservation and scheduling of resources, and the return of information about available resources. Fortunately, the Forum was able to learn about the advantages and pitfalls of dynamic process creation from PVM. There were several requirements the Forum sought to uphold in the design of process management for MPI-2. First, the MPI-2 process model must be valid across different computing platforms. Second, MPI must not take over operating system responsibilities; there must be a clean interface between system and application software. Third, MPI must guarantee communication determinism; it can’t introduce race conditions. Last, MPI-1 programs must work under MPI.

The functionality the Forum gave MPI for process management was to: allow for creation and termination of processes after MPI application has started, permit communication to be established between newly started processes and existing processes even if they share no parent-child relationship, and provide for communication between existing, non-related MPI applications.

The ability to dynamically start MPI processes was defined in two functions: `MPI_Comm_spawn` and `MPI_Comm_spawn_multiple`. Each of these starts a specified number of processes and returns an intercommunicator for information interchange between the parent communicator and the child communicator. All children created by a spawning operation share an intracommunicator. `MPI_Comm_spawn_multiple` allows the loading of several different binaries, while `MPI_Comm_spawn` requires that all processes be an instance of the same executable. The spawning operations are defined to be collective over the parent and child communicators. This means that all processes in the parent communicator must call the function, which will not return until all the child processes have been created and initialized. This also means that a child process has a communicator for a parent (i.e. a group of processes).

Given the many different types of parallel computing systems, the MPI Forum realized that there was not a way to specify where and how the new processes should be created in a platform independent way. In order to allow this flexibility in process creation, they said that an argument to the spawn functions could be used to specify this information in a platform and MPI implementation dependent way.

Another part of the process management features of MPI-2 is the ability to establish communication between MPI processes that don't have a parent-child relationship and between existing, non-related MPI applications. Some of the difficulties of this lie in knowing how to contact the other process with no prior established communication. MPI-2 provides function interfaces for publishing and retrieving contact information from a name server or other such service: `MPI_Publish_name` and `MPI_Lookup_name`. The procedures created by the Forum for this functionality are reminiscent of socket functions: `MPI_Open_port`, `MPI_Close_port`, `MPI_Comm_accept`, and `MPI_Comm_connect`. They also define the ability for two MPI processes connected on a Berkeley socket to get an intercommunicator with `MPI_Comm_join`.

2.3.2 MPI-I/O

Another important addition to MPI-2 is parallel file I/O. This definition does not refer to terminal I/O (stdout,stdin,stderr), but to file access. The Standard does not specify the library interface to access data, how files can be accessed by non-MPI programs, how files are organized in directories, what filenames are allowed, file protection policies, or file storage mechanisms. It was designed to work with a wide range of existing file systems. MPI derived datatypes are used to partition a file for use by multiple processes. This allows the use of heterogeneous systems to be transparent to the user. Again, the use of the Info argument is allowed so that the programmer can give hints to the MPI implementation on how to set up the data file for efficient use. This is in terms of both file access patterns and I/O hardware.

There are two different types of file pointers in MPI-I/O: shared and individual. The shared file pointer is common to all processes in the communicator that opened the file. The individual file pointer belongs to one specific process only. This allows for flexible file access operations.

There are many different functions defined in the MPI-2 interface for file access. They are varied to allow many efficient access patterns. The opening and closing of a file are collective operations over a communicator and are defined as `MPI_File_open` and `MPI_File_close`. There are several types of routines for reading and writing in MPI: collective, ordered collective, split collective, individual blocking, and individual non-blocking. Table 1 lists the file access operations that are collective, ordered collective, or individual.

Table 1: MPI-I/O Individual and Collective File Access Operations

Operation	File Pointer	Collective	Blocking
`MPI_File_read/` `MPI_File_write`	Individual	No	Yes
`MPI_File_read_all/` `MPI_File_write_all`	Individual	Yes	Yes
`MPI_File_read_at/` `MPI_File_write_at`	Explicit Offset	No	Yes
`MPI_File_read_at_all/` `MPI_File_write_at_all`	Explicit Offset	Yes	Yes
`MPI_File_iread/` `MPI_file_iwrite`	Individual	No	No
`MPI_File_iread_at/` `MPI_File_iwrite_at`	Explicit Offset	No	No
`MPI_File_read_shared/` `MPI_File_write_shared`	Shared	No	Yes
`MPI_File_iread_shared/` `MPI_File_iwrite_shared`	Shared	No	No
`MPI_File_read_ordered/` `MPI_File_write_ordered`	Shared	Yes	Yes
`MPI_File_seek`	Individual	No	Yes
`MPI_File_seek_shared`	Shared	Yes	Yes

The MPI Forum also defines operations for file access that are split collective. In other words, a single collective file access operation is split into two function calls: a begin routine and an end routine. This essentially allows for collective non-blocking file access. After the begin routine returns, the processes can do useful work before calling the end routine. The buffers given to the begin routine cannot be used until the matching end call completes. The split collective file access routines are shown in Table 2.

Table 2: MPI-I/O Split Collective File Access Operations

Operation	File Pointer
`MPI_File_read_all_begin/` `MPI_File_write_all_begin` `MPI_File_read_all_end/` `MPI_File_write_all_end`	Individual
`MPI_File_read_at_all_begin/` `MPI_File_write_at_all_begin` `MPI_File_read_at_all_end/` `MPI_File_write_at_all_end`	Explicit Offset
`MPI_File_read_ordered_begin/` `MPI_File_write_ordered_begin` `MPI_File_read_ordered_end/` `MPI_File_write_ordered_end`	Shared

There are many other operations in MPI-I/O that are not discussed here. There are several books that discuss this topic in detail [May01, GHL+98, GLT99].

2.3.3 Remote Memory Access

The last major feature of MPI-2 is one-sided communication, or Remote Memory Access (RMA). This allows the exchange of data between processes in such a way that only one process needs to specify the sending and receiving parameters. This is helpful for programs that may have data access needs that change at runtime. It saves all involved processes from having to do computation to discover the new data access parameters. Only one process needs to know the parameters and can perform the data exchange operation on its own. This form of message passing is achieved by separating the synchronization from the communication. There are three data exchange operations: `MPI_Put` (remote write), `MPI_Get` (remote read), and `MPI_Accumulate` (remote update). There are many synchronization routines that enable the efficient use of RMA on different computing platforms. The memory access model is similar to a weakly coherent memory system [GHL+98]. The correct ordering of accesses to memory must be enforced by the user with synchronization calls.

There are two types of remote memory operations. One is *active target*, which means that data moves from one process's memory to the memory of another, and both processes are explicitly involved in the communication. This is similar to message passing except all data transfer information is provided by one process only, and the second process participates only in synchronization. The other is *passive target*, which means that data moves from the memory of one process to the memory of another process, and only the origin process is explicitly involved in transfer. This is similar to a shared memory model.

Two types of time periods are defined for this model. The first is an *access epoch*. This is the time between synchronization calls when remote memory access is allowed. This refers to the origin process (the process executing a put or get operation). The second type is an *exposure epoch*, which is the time between synchronization calls when a process's memory is exposed. This refers to the target process (the target of a put or get operation).

The memory exposed or accessed by a process is called a "window." Each process in the communicator used to create a window specifies the region of memory that they wish to share. The creation of a window is a collective operation over a communicator. The function `MPI_Win_create` returns a window object that will be used by the processes in subsequent RMA operations. When the group of processes is finished with the window object, they all call `MPI_Win_free` with the window object as a parameter. This free operation is collective over all processes in the communicator that created the window object.

The synchronization operations for the active target model can either involve the entire group of processes represented by the communicator or a subset of that group. There are four functions that are used to coordinate subsets of processes in the group: `MPI_Win_start` (start an access epoch), `MPI_Win_Complete` (ends an access epoch), `MPI_Win_post` (starts an exposure epoch), and `MPI_Win_wait` (ends an exposure epoch). The function `MPI_Win_fence` is collective and is used to coordinate all the processes in the communicator. The function is called twice, once to open the window for access/exposure, and again to stop the access/exposure epoch. The second call to `MPI_Win_fence` will not return until all memory access functions on that window have completed.

The passive remote memory model uses `MPI_Win_lock` and `MPI_Win_unlock` to coordinate memory access. These functions can be used to lock the exposure window on one process without its explicit cooperation. The unlock function will not return until all memory access operations on the specified window have completed.

The last addition to MPI that we discuss is the ability to name MPI objects. The MPI routines in this feature family allow the user to associate printable identifiers with MPI objects such as communicators, windows, and datatypes. This is useful anytime the programmer needs to receive detailed information about the MPI program: when debugging, reporting errors, or measuring its performance. For example, if the programmer doesn't name a communicator, then it is given an implementation dependent identifier, perhaps an integer. If there are many communicators in the program, it is difficult for the programmer to match the integer identifier to a particular group of processes. However, if the group is identified by a human readable string provided by the programmer, the matching of process group to communicator object is much simpler.

2.4 MPICH

This section provides information about the MPICH implementation of the MPI Standard. Section 2.4.1 gives an overview of MPICH. Section 2.4.2 tells how an MPI application starts with the MPICH ch_p4 device. Last, Section 2.4.3 explains the details of starting an application with the MPICH ch_p4mpd device.

2.4.1 Overview of MPICH

The MPICH implementation is developed at Argonne National Laboratory. It was first released in May 1994, which was the same year as the MPI standard. The MPICH implementors were able to achieve such an early release date by working closely with the MPI Forum and developing their implementation alongside the development of the MPI Standard [GLD+96]. Another reason that the MPICH implementation was quickly released is that it was built on top of existing message-passing libraries: P4 and Chameleon. The current version of MPICH fully supports MPI-1.2 and a very little of MPI-2. A beta version of MPICH2 is available, which is the MPI-2 version of MPICH. This beta version currently supports active-target RMA and MPI-I/O [MPIC03].

The architecture of MPICH is designed to be portable and to allow for performance optimizations on different platforms [GLD+96]. The MPI functions are implemented on top of the Abstract Device Interface (ADI). The ADI is the means by which MPICH achieves portability and performance. All MPI functions are implemented in terms of macros and functions that make up the ADI. There are many implementations of the ADI in MPICH. One of these is the channel interface. The channel interface can be very small, and is the quickest way to provide support for a new environment. Only five functions in the channel interface need to be implemented to support a new system. The most important channel interface implementation is Chameleon; the “CH” in MPICH stands for Chameleon. Chameleon provides portability in terms of macros, which incur no runtime overhead, because the macros are resolved at compile time Chameleon macros exist for most vendor message-passing systems, including P4.

An MPICH installation is described in terms of the ADI implementation that it uses. There may be more than one ADI implementation, or device, available for a given system. For instance, we use both the ch_p4 and ch_p4mpd devices on our cluster. The ch_p4 device is a P4 implementation. The MPICH development team considers the ch_p4 device to be outdated and is in the process of perfecting a replacement for it, the ch_p4mpd device. The ch_p4mpd device uses Multi-Purpose Daemons to provide enhanced process management and quick process startup [Thak00].

2.4.2 The MPICH ch_p4 Device

The procedure for starting an MPICH ch_p4 program is relatively simple. The `mpirun` command creates a procgroup file that includes all the nodes that are to participate in the computation. It then launches the first copy of the application, giving the procgroup file as an argument. The application runs until it comes to `MPI_Init`. At this point, it analyzes the procgroup file and creates the slaves. If the slave is created on the same node as the master process, the slave is created using `fork`. Otherwise, a remote shell command is used to start the slave on the remote node. Each slave is started with a parameter that tells it that it is a slave process. It is also given the hostname of the master node along with the port number on the master node to be used for communication. Each of the slaves runs until `MPI_Init` is encountered. Figure 1 shows these events.

At this point the slaves parse the command line arguments and communicate back to the master node. Port information and any user supplied command line arguments are exchanged. At the end of `MPI_Init`, the distinction between master and slave processes ceases to exist and the computation begins.

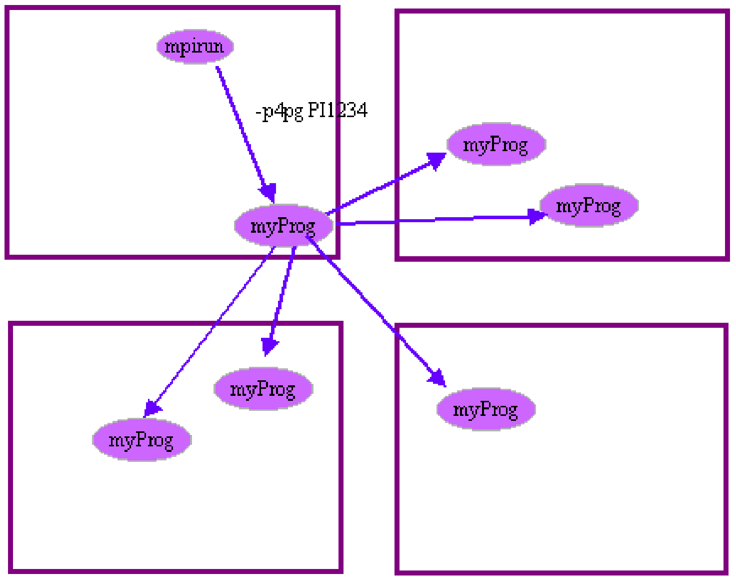


Figure 1: MPICH ch_p4 Process Startup
This shows the startup procedure for the MPICH ch_p4 device. The mpirun process starts the master MPI process, which in turn, starts the slaves.

2.4.3 The MPICH ch_p4mpd Device

The MPICH ch_p4mpd device uses daemon processes to help control the MPI application. As a result the procedure to start an application with this device is slightly more complicated. First of all the user must start the mpd daemons on the nodes. The mpd daemons are connected in a ring. Then, the user starts the mpirun process which connects to its local mpd through a unix socket. These events are shown in Figure 2.

The mpd daemons fork manager processes called mpdman, one for each MPI process to be started. The mpdman processes are started consecutively around the ring, beginning at the "next" mpd daemon, unless otherwise specified. Then, the manager processes themselves form a communication ring. These steps are shown in Figure 2.

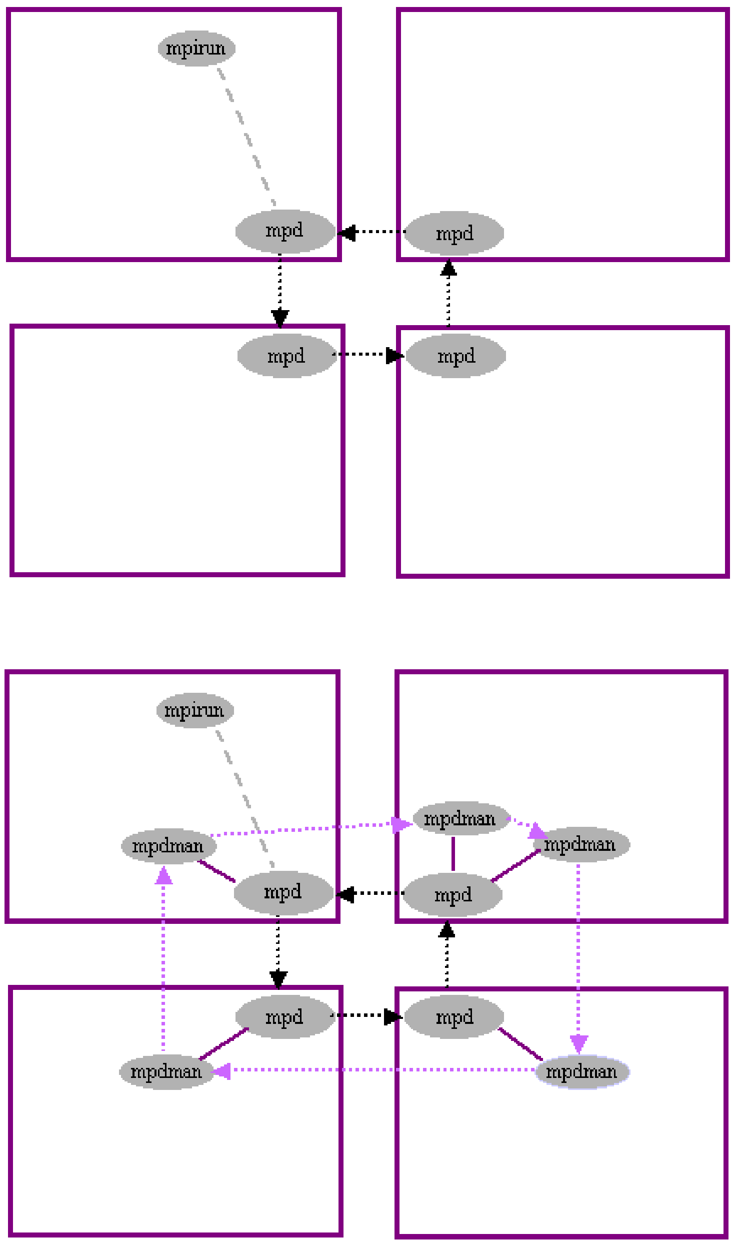


Figure 2: MPICH ch_p4mpd Process Startup

This shows the first steps in starting an MPICH ch_p4mpd application. The top figure shows that the mpirun process connects to the mpd daemons. The bottom figure shows the mpd daemons starting mpdman processes, one for each MPI process that will be started. The mpdman processes are connected in a communication ring.

The mpdman processes each spawn an MPI process using `fork`. The mpirun process disconnects from the mpd daemon and connects to the first mpdman process. Stdin from mpirun is redirected to the client of this manager process. The mpdman processes intercept standard I/O from the MPI processes and also deliver command line arguments and environment variables from mpirun to them. Figure 3 depicts these events. After this initialization is finished, the computation begins.

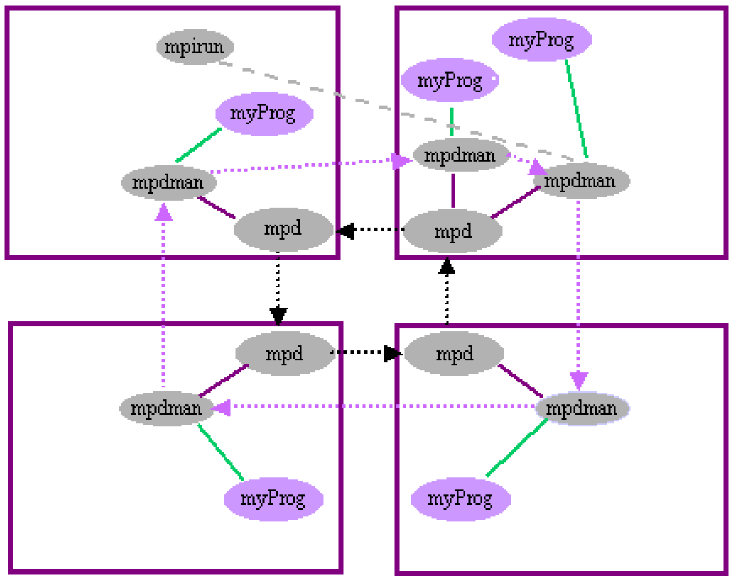


Figure 3: MPICH ch_p4mpd Computation Begins
The last steps in the startup procedure for MPICH ch_p4mpd programs are shown here. The mpdman processes spawn the MPI processes. Input and output from the MPI processes is redirected through the mpdman processes.

2.5 LAM/MPI

This section discusses the LAM/MPI implementation of the MPI Standard. Section 2.5.1 gives an overview and some history of the implementation. Section 2.5.2 discusses its architecture. Last, Section 2.5.3 describes the LAM runtime environment and how MPI programs are started with LAM/MPI.

2.5.1 Overview of LAM/MPI

LAM/MPI is an implementation of the MPI Standard. It was originally developed at the Ohio Supercomputer Center. Later, LAM/MPI became the responsibility of the Laboratory for Scientific Computing (LSC) at the University of Notre Dame under the direction of Dr. Andrew Lumsdaine. In the fall of 200, LAM/MPI moved with Dr. Lumsdaine to Indiana University and the Open Systems Laboratory, which is where the project resides today [LTA03].

LAM/MPI is currently in version 7.0 and is freely distributed as an open-source implementation of the MPI standard. It is a full implementation of the MPI-1.2 Standard and a partial implementation of the MPI-2 Standard. The MPI-2 functionality supported includes dynamic process creation (`MPI_Spawn`), MPI Client/Server, one-sided communication, C++ bindings, and MPI I/O. LAM/MPI exceeds the MPI Standard by offering Guaranteed Envelope Resources (GER) [Saph97]. GER is a promise to the user of how much buffer space is available for pending communication. The MPI Standard makes no mention of such a guarantee.

The are other features of LAM/MPI that make it appealing to parallel programmers, especially those working with clusters of workstations. For instance, LAM/MPI contains hooks that enable specialized debuggers to examine MPI message-passing queues and the state of programs with respect to MPI communicators. Supported debuggers include TotalView (Etnus) and The Distributed Debugging Tool (Streamline Computing). Another feature is its support for several different communication transport layers, including Myrinet [Seit01]. LAM/MPI also contains collective algorithms for efficiently utilizing SMP clusters, using network transport to communicate between processes on different nodes and shared memory to communicate between processes on the same node. LAM/MPI offers support for heterogeneous clusters of workstations as well as for Globus [FKN+02] and Interoperable MPI [GHD00], which allow an MPI application to span clusters of clusters which may have heterogeneous hardware as well as heterogeneous MPI environments.

The acronym LAM stands for Local Area Multicomputer. LAM is based upon the Trollius project from the Ohio Supercomputer Center [SLG+00]. The goals of the Trollius project were to provide support for general process management and communication (process to process message passing) in a multicomputer, while striving to provide portability across topologies and hardware [Burn99]. LAM/MPI is built upon the LAM that grew from the Trollius project. The LAM layer is independent of MPI. For instance, PVM was implemented on top of LAM [BDV94].

The LAM layer is evident in the LAM/MPI implementation as the lamd daemon. An instance of this daemon runs on every node in the multicomputer. This daemon adds functionality for process monitoring and debugging. It is possible to take snapshots of the progress and states of the processes in the MPI application with information gathered by the lamd daemons. They also lend fault tolerance by “shrinking” the multicomputer as nodes fail and have the capability of “growing” the multicomputer as nodes are added. It has been argued that the existence of the lamd daemon makes LAM/MPI the choice MPI implementation for development and debugging of applications [Saph97].

Communication between processes in the multicomputer can happen one of two ways. The communications can either be routed through the lamd daemon or they can go directly to the target process. When the messages are passed through the lamd daemons, process monitoring and debugging is enabled. However, this mode is not recommended for most production environments as the indirection may lead to added communication overhead. The other mode of communication, called client-to-client (c2c), is deemed to be more efficient. However, its implementation is not portable and would likely require modification of the LAM/MPI code to accommodate a new system. The c2c mode uses TCP as its default protocol. The TCP sockets are connected between processes at initialization and are kept open for the duration of the application to avoid the overhead of reconnecting the sockets [LT00]. The user of LAM/MPI can switch between these two message-passing modes at application startup via a command line argument. LAM/MPI does not need to be recompiled for the switch. In this way, LAM/MPI gives the user an opportunity to use support for process management and debugging with a platform independent implementation, as well as a means for efficient communication that can be tailored for speed on the target system.

LAM/MPI has been proclaimed by some to be the “clear choice” for MPI applications on Beowulf clusters [ASQ99]. Several studies have shown that LAM/MPI outperforms MPICH on clusters of workstations when running in c2c mode [ASQ99,Nevi96,OF00].

2.5.2 LAM/MPI Architecture

The architecture of LAM/MPI is layered. The upper stratum is the MPI layer, which is portable and completely separate of the actual means of communication [LT00]. This upper layer uses the Request Progression Interface (RPI) to access the machine and protocol dependent lower layer, the Trollius core. The RPI is responsible for all communication between the MPI ranks. All MPI communication functionality is implemented in this interface by ten primitives [LT00]. Messages are viewed as "requests" by the RPI. The state of the request progresses from start to active to done as the message is processed.

There are two versions of the RPI, LAMD-RPI and C2C-RPI. These correspond to the communication modes mentioned earlier, either through the lamd daemon or client-to-client. The LAMD-RPI is portable and provides for process monitoring and debugging. It uses UDP for message passing, implementing its own time-out and retransmission policy [CLMR99]. The C2C-RPI, on the other hand, is not portable, but provides a means for more efficient message passing. The C2C-RPI has three flavors: tcp, sysv, and usysv. The user is able to choose between these communication methods via a command line argument to mpirun at runtime. In tcp, the ranks communicate solely through TCP sockets. The sysv choice uses TCP sockets to send messages to ranks that are on different nodes, but uses shared memory to do so to ranks on the same node. This method uses SYSV semaphores for locking the shared memory. The usysv mode is the same as sysv, but uses spin-locks to protect shared memory. In C2C-RPI, the TCP connections are made at initialization time (`MPI_Init`) and are kept open for the duration of the application.

2.5.3 LAM/MPI Runtime Description

The LAM environment must be established before any MPI programs can be run. The LAM environment is started by issuing the `lamboot` command. This command takes a text file listing of machine names and from it forms a multicomputer. It does this by invoking the `hboot` command on the remote nodes. This attempts to start a lamd process on each machine in the multicomputer. These events are depicted in Figure 4.

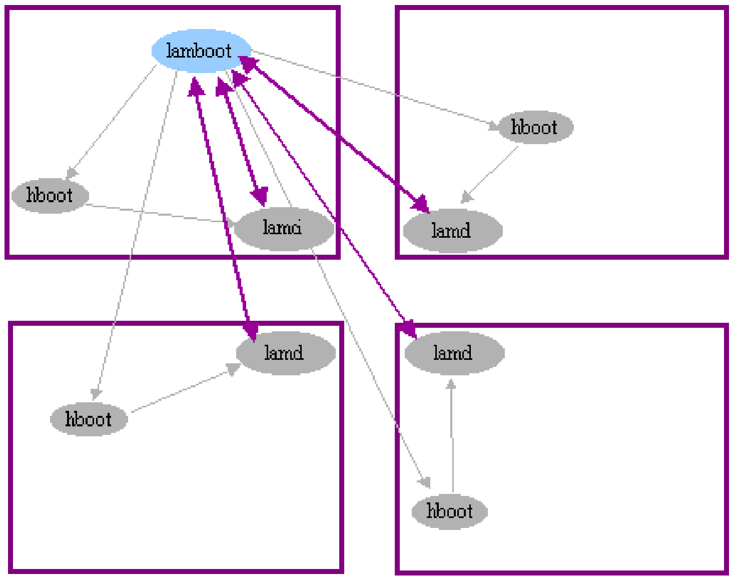


Figure 4: LAM/MPI Starting the LAM Daemons
This figure shows the steps for starting the LAM environment. The user invokes the `lamboot` command, which starts `hboot` on each node. The hboot processes each start a LAM daemon.

Then each node communicates a dynamic port back to the lamboot process. The lamboot process collects each of these ports, then sends the list of ports, along with the respective hostnames, out to each node in the configuration file. If a machine fails to respond to the lamboot process (i.e. fails to send back its dynamic port) within a certain time limit, lamboot uses the wipe command to terminate the LAM environment and reports the error.

In order to run MPI processes under LAM, the user must invoke the `mpirun` command with the appropriate arguments. The mpirun process establishes itself with the local lamd process, setting up a unix domain socket for communication with the lamd daemon. The arguments given to mpirun are used to set up the MPI application's

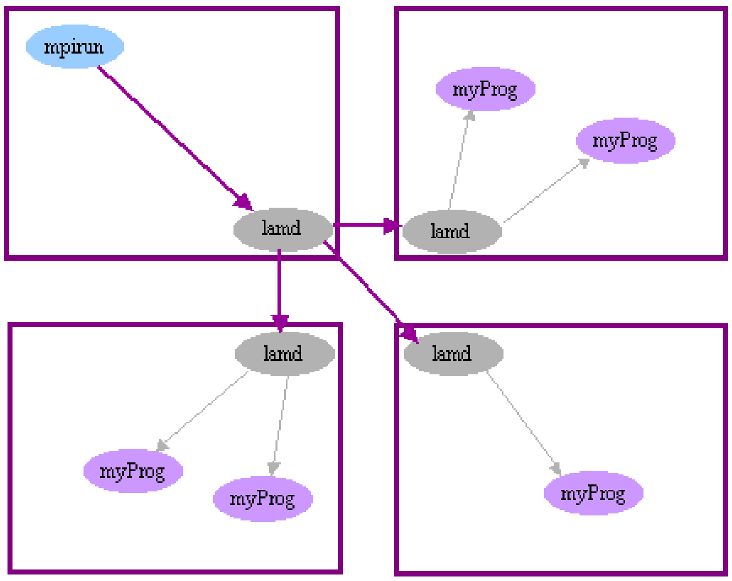


Figure 5: LAM/MPI Starting the MPI Processes
The mpirun process instructs the LAM daemons to start the MPI processes.

environment. Once mpirun has parsed and processed all the arguments, a data structure representation of the MPI application's environment is made. Using this data structure, the mpirun process contacts its local lamd daemon, giving it the name of each executable that is to be part of the MPI application, the node on which the executable should be run, and any necessary arguments for that executable. Figure 5 shows these steps.

The lamd daemon local to the mpirun process sends messages to the lamd daemons on the other nodes informing them of the executables they should start and any runtime information they might need. After the remote MPI processes are started, the remote lamd daemons return the process identifiers for the newly started executables. If there were any unsuccessful attempts to start the executables, the entire MPI application is terminated and an error is returned to the user. After the MPI processes have been started, mpirun is able to gather information about all the processes in the MPI

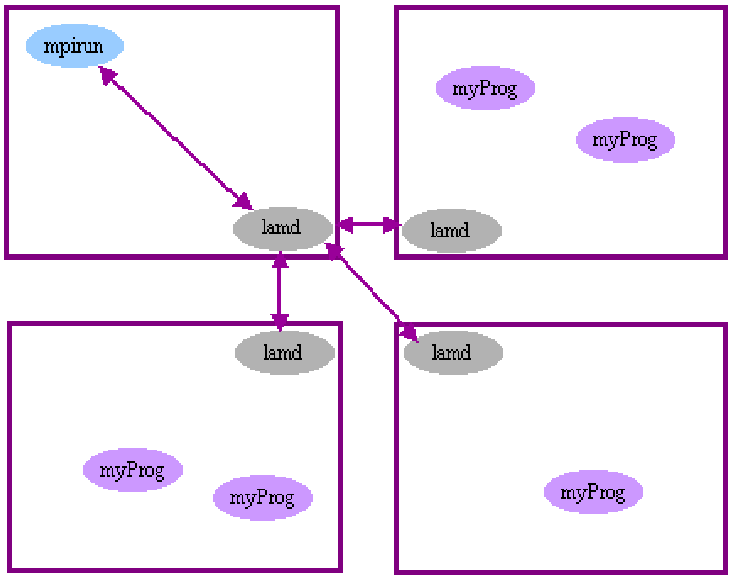


Figure 6: LAM/MPI MPI Computation Begins
Information about all the processes in the MPI application is distributed to all the LAM daemons.

application. The mpirun process instructs its local lamd daemon to distribute this information to the remote lamd daemons. This information will be provided to the MPI processes when necessary. Figure 6 depicts these events.

The mpirun process waits on the termination of all processes in the MPI application before it exits. If any of the processes die with an error the mpirun process kills the entire MPI application and reports the error.

3 Paradyn

This chapter will discuss the Paradyn Parallel Performance Tool. Section 3.1 begins by giving background information about Paradyn. Section 3.2 outlines the existing level of support Paradyn had for MPI before our changes.

3.1 Background

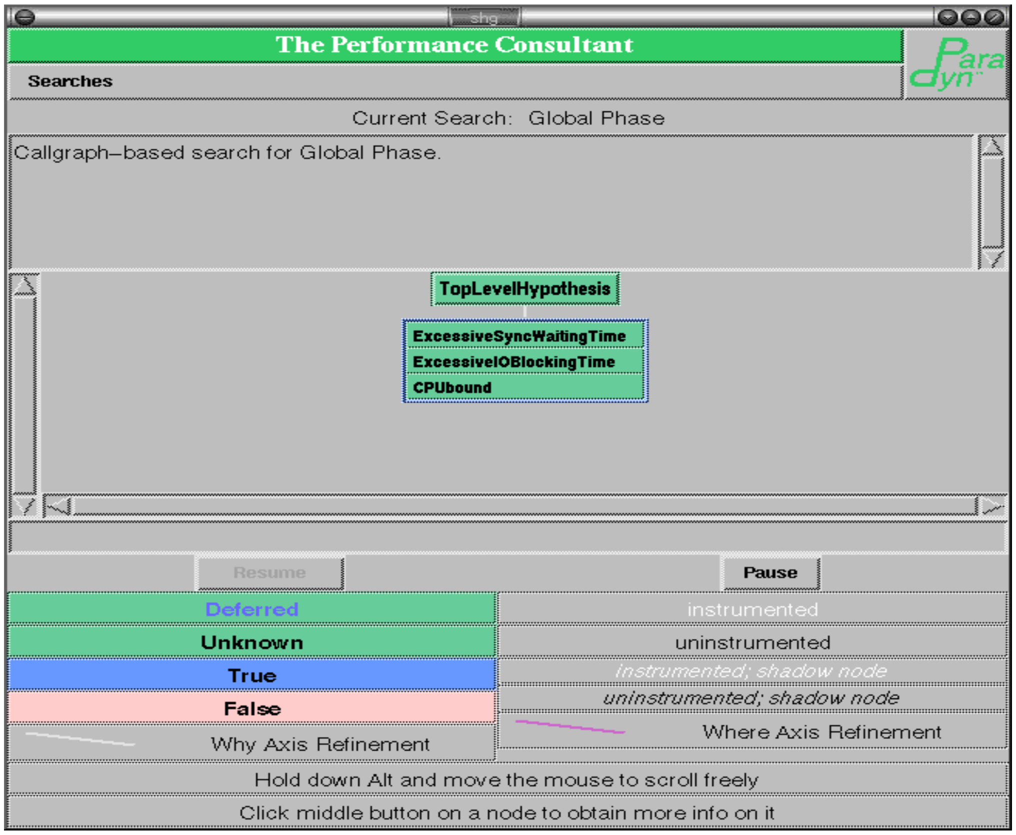
In this section, we discuss the most relevant aspects of the Paradyn Parallel Performance Tool. For more complete information, the reader is invited to explore the Paradyn User's Guide [PG03].

Paradyn is an automated parallel performance tool developed at the University of Wisconsin. Paradyn was chosen for this project because of its ease of use, appealing features, and existing support for MPICH. Paradyn is freely available and well-documented. Paradyn was developed to aid in solving grand challenge problems and is used as a research tool in national laboratories.

Paradyn employs dynamic instrumentation to insert performance measurement instructions into programs at runtime. This method is more convenient for the user than is found with traditional performance measurement tools. The user does not need to insert the instructions on his/her own or recompile the code whenever performance measurement is desired. Dynamic instrumentation also allows for the reduction in the amount of performance data that must be collected from the parallel application, as the decision on what to instrument can be made dynamically. This allows performance measurement instructions to be removed from "uninteresting" code segments at run time. The reduction in performance data is significant when one considers that data must be collected from every process in the application, possibly a very large number. The ability to change what performance measurements are taken at runtime is a convenience feature for the application programmer.

Paradyn is a profiling tool, which means that it collects summary information about program runs, such as execution times and the number of calls. This aggregate data is typically used to characterize program behavior and find where a program is spending most of its time. A profiling tool can be contrasted with a tracing tool, which records information about significant events in the execution of a program in such a way that the events can be reconstructed later. A tracing tool generally provides more detailed information about program execution than does a profiling tool. However, tracing tools tend to generate large data files due to the volume of information that they collect. Dan Reed, Director of the NCSA, estimated that a tracing tool that records function entry and exits on a parallel system with hundreds of processors could easily generate a data volume of 10 MB/second [RAD+98]. Paradyn is scalable in that the profile data it collects is kept in a pre-set amount of memory. If Paradyn collects more data than will fit in the allocated memory, it aggregates the data that it has already collect into a smaller space and then continues to collect data into the newly freed space.

Paradyn automatically detects performance bottlenecks for the user with its Performance Consultant (PC). The PC starts by investigating several common metrics at a high level in the program. If any appear to be bottlenecks, the PC investigates them further. The results of this search are displayed at runtime in the Performance Consultant window. Figure 7 shows the Performance Consultant window before the bottleneck search begins. We see that the three top-level hypotheses, ExcessiveSyncWaitingTime, ExcessiveIOBlockingTime, and CPUBound, are all green, which means their values are unknown.



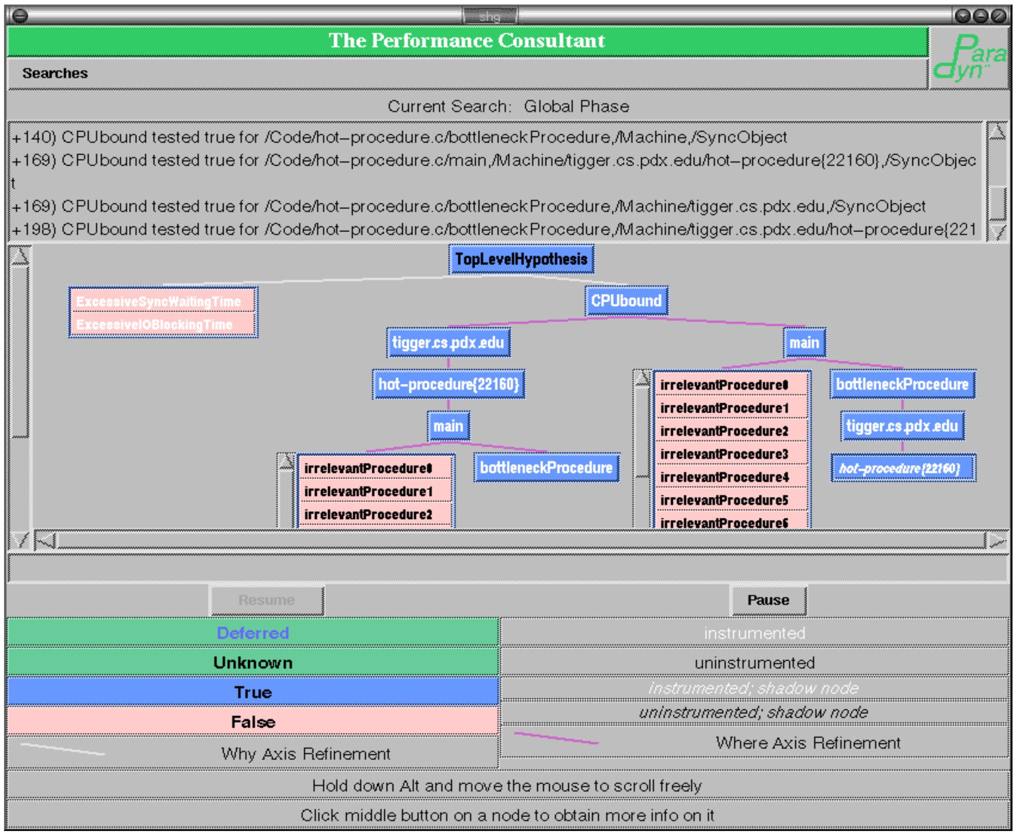
Callgraph–based search for Global Phase.

Hold down Alt and move the mouse to scroll freely

Click middle button on a node to obtain more info on it

Figure 7: The Paradyn Performance Consultant at Program Start
This figure shows the Performance Consultant window before program execution begins. The test results of the three top-level hypotheses are unknown.

In Figure 8, the Performance Consultant window display for the program's end is shown. Here we see that the top-level hypothesis CPUBound has tested true, so the box representing it has turned blue. We also see that the Performance Consultant has drilled down into the user’s program to find the performance bottleneck. Now that it has established *what* the problem is, it now needs to find *where* the problem exists.



```
+140) CPUbound tested true for /Code/hot–procedure.c/bottleneckProcedure,/Machine,/SyncObject
+169) CPUbound tested true for /Code/hot–procedure.c/main,/Machine/tigger.cs.pdx.edu/hot–procedure{22160},/SyncObjec
t
+169) CPUbound tested true for /Code/hot–procedure.c/bottleneckProcedure,/Machine/tigger.cs.pdx.edu,/SyncObject
+198) CPUbound tested true for /Code/hot–procedure.c/bottleneckProcedure,/Machine/tigger.cs.pdx.edu/hot–procedure{221
```

Figure 8: The Paradyn Performance Consultant at Program End
This figure shows the Performance Consultant window after the Performance Consultant has finished its analysis. It has found the program to be CPUBound and the bottleneck location to be the function `bottleneckProcedure`, so its box is blue. It has also correctly determined that the other procedures are not bottlenecks for this program, so their boxes are pink.

We see that the PC has investigated two locations for the bottleneck: Machine and Code. In this simple example, there is only one machine used for the program, tigger.cs.pdx.edu. The Performance Consultant has determined that the bottleneck exists on this machine and has refined further to search the Process locations. This example only has one process and it has tested true for being a bottleneck location. Then, we see that the search has continued from Process to Code. There, the Performance Consultant determined that the functions `main` and `bottleneckProcedure` are locations where the program is CPU bound. The Performance Consultant also discovered that the other functions in the program, named `irrelevantProcedureX` are not CPU bound, so their boxes are pink, for false. Similarly, for the top-level hypothesis CPUBound, the search in the Code hierarchy continued. The Performance Consultant drilled down to find than `main` and `bottleneckProcedure` were computational bottlenecks. Then, it searched Machine locations, and found tigger.cs.pdx.edu to be the location of a bottleneck. Next, it searched Process locations and determined that the process hotprocedure is CPU bound. Note that the text in this box is in italics. This indicates that it is a shadow node, or a copy of the other node representative of the same process, and will not be refined further.

3.2 Existing Paradyn Support for MPI on Clusters of Workstations

When we started this project, Paradyn did not support clusters with a non-shared file system. We were unable to start any MPI programs that used multiple nodes without this support. Also, it did not measure certain metrics, such as number of messages, or message bytes sent and received, for MPICH programs written in C/C++ on our system. Paradyn did have some support for the MPICH ch_p4mpd device. MPICH ch_p4mpd programs could be started and run with Paradyn. However, some important command line arguments for `mpirun` were not supported, such as `-m` and `-wdir`, which allow the user to specify a machinefile and a working directory, respectively. The

MPICH ch_p4 device was not supported by Paradyn on our system. We discovered a bug in Paradyn that precluded us from running any ch_p4 programs. Paradyn did start and run LAM/MPI programs. However, it only supported the `-np` argument to `mpirun`. Given LAM/MPI's extensive and flexible arguments to `mpirun`, this was quite limiting. Paradyn did not support multiple executables in an MPI program.

3.2.1 Paradyn and the MPICH ch_p4 Device

Here we explain the startup procedure for MPICH ch_p4 programs under Paradyn. In order to run an MPICH ch_p4 program under Paradyn, the user invokes the Paradyn frontend. On the Paradyn user interface, the user specifies several parameters such as working directory, host of the master MPICH program, and the `mpirun` command itself. If the master MPICH process is to be run on the localhost, Paradyn spawns the mpirun process using `fork`. Otherwise, Paradyn uses a remote shell command to start mpirun on another host. Instead of telling mpirun to start the MPICH program that the user specified, Paradyn tells mpirun to start a script that was generated by Paradyn, which is represented by *pdd-scr* in Figure 9. Thus, the mpirun process starts the Paradyn script. The script contains commands to start a Paradyn daemon. The arguments to the Paradyn daemon startup command inform the Paradyn daemon of the MPICH program that the user specified. Next, the Paradyn daemon opens up a communication socket with the Paradyn frontend. This chain of events is illustrated in the first box of Figure 9, below. In the diagram, the squares represent compute nodes and the ovals represent running processes on the compute nodes. The arrows symbolize a parent-child relationship between the processes. The solid line depicts a communication socket.

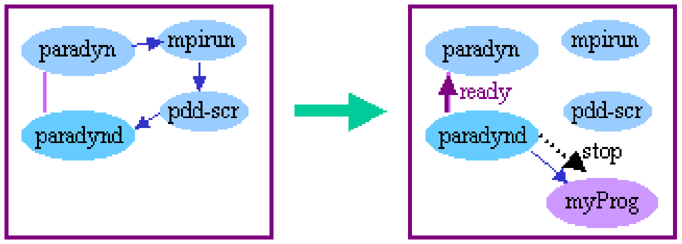
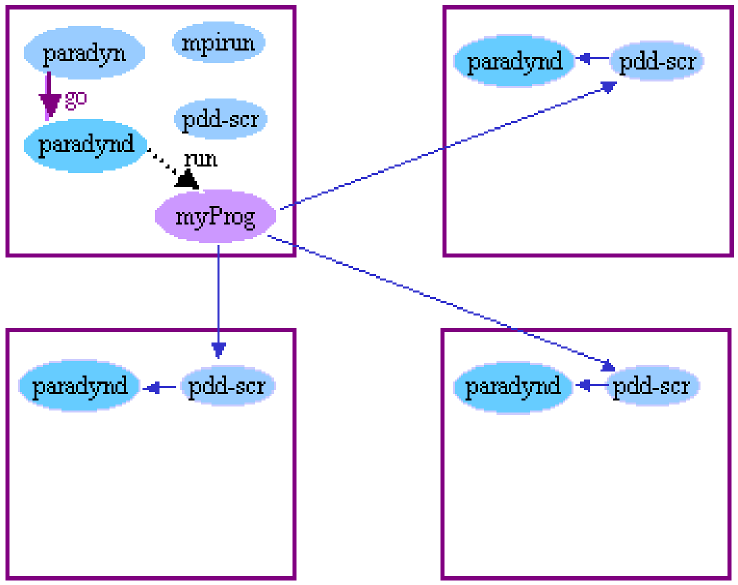


Figure 9: Paradyn/MPICH ch_p4 Initial Startup
This figure shows the initial process startup for Paradyn and the MPICH ch_p4 device. We see that the paradyn process starts mpirun, which, in turn, runs a script that starts a Paradyn daemon. Then, the Paradyn daemon starts the master MPICH process, stops in it `main` and reports to the Paradyn frontend that it is ready. At this point, the Paradyn frontend displays the “Run” button to the user.

The Paradyn daemon then forks the master MPICH process. A communication pipe is established between the MPICH process and the Paradyn daemon. The MPICH process is stopped in the beginning of `main`. The Paradyn daemon communicates back to the frontend that it is ready. The second box in Figure 9 shows these steps. In this diagram, a thicker arrow is used to portray communication over a socket. The thin arrow still represents a parent-child relationship between processes, and the dotted arrow shows control of the MPICH program by Paradyn.

At this point the “Run” button on the Paradyn user interface is enabled. Once the user clicks on the “Run” button, the Paradyn frontend communicates to the Paradyn daemon that it can proceed. The Paradyn daemon continues the master MPICH process. The master MPICH process is responsible for starting the other MPICH pro-



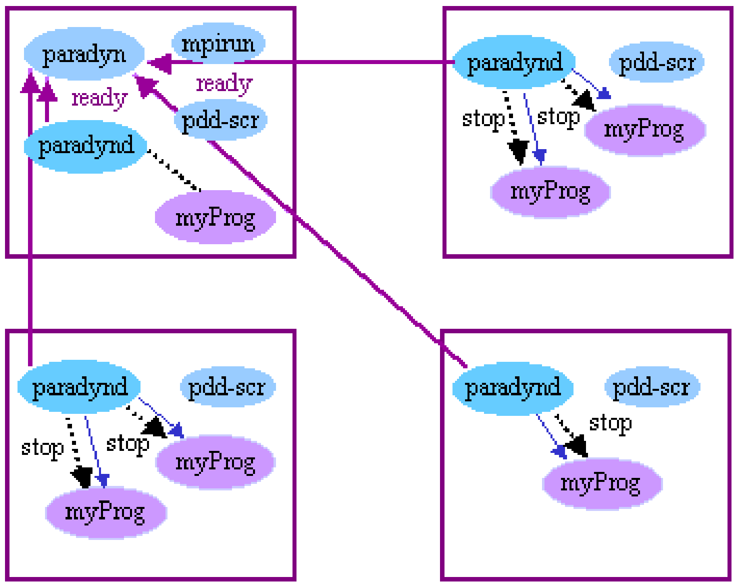


Figure 10: Paradyn/MPICH ch_p4 Starting Remote Paradyn Daemons and MPI Processes
The top figure shows what happens after the user hits the “Run” button on the Paradyn user interface. The Paradyn frontend instructs the daemon to run the process. The master MPICH process then starts Paradyn generated scripts. These scripts execute Paradyn daemons. In the bottom figure we see that the Paradyn daemons on the remote nodes start the MPI processes and then stop them in `main`. They report to the Paradyn frontend that they are ready.

cesses that make up the parallel application. However, the master MPICH process is told that it is supposed to start the Paradyn generated script. As a result, a script is started on each node, which in turn starts a Paradyn daemon. The top diagram in Figure 10 shows this series of events. Note that in this diagram, the thin arrows that span two compute nodes do not represent a parent-child relationship between processes, but depict a remote shell command to start those processes.

Communication sockets are opened between each Paradyn daemon and the Paradyn frontend. Each of these Paradyn daemons forks at least one MPICH process and possibly more, depending upon user specifications in the command line arguments to `mpirun`. The slave MPICH processes are stopped in `main`. The Paradyn daemons communicate back to the frontend that they are ready. This is illustrated in the bottom diagram of Figure 10.

The Paradyn frontend communicates to the daemons that they can proceed. As a result, the daemons continue the MPICH processes, as seen in the top diagram of Figure 11.

At this point, internal MPICH initialization is done, such as the exchange of hostnames and port numbers between each of the slave MPICH processes and the master MPICH process. When initialization is complete, the program exits `MPI_Init` and begins to execute the user's code. The bottom diagram of Figure 11 shows these events. The bidirectional dotted arrows represent the exchange of information between the MPICH processes. The dotted lines are used to symbolize that the MPICH processes are running under the control of the Paradyn daemons.

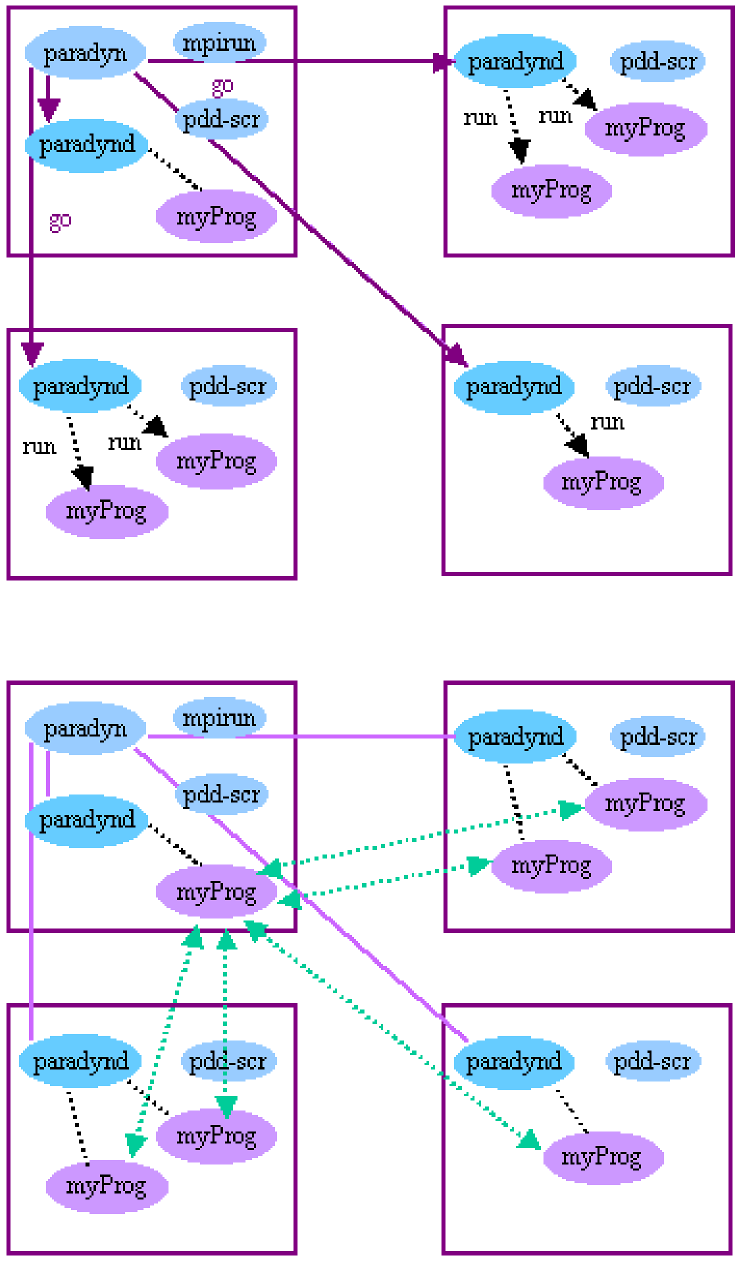


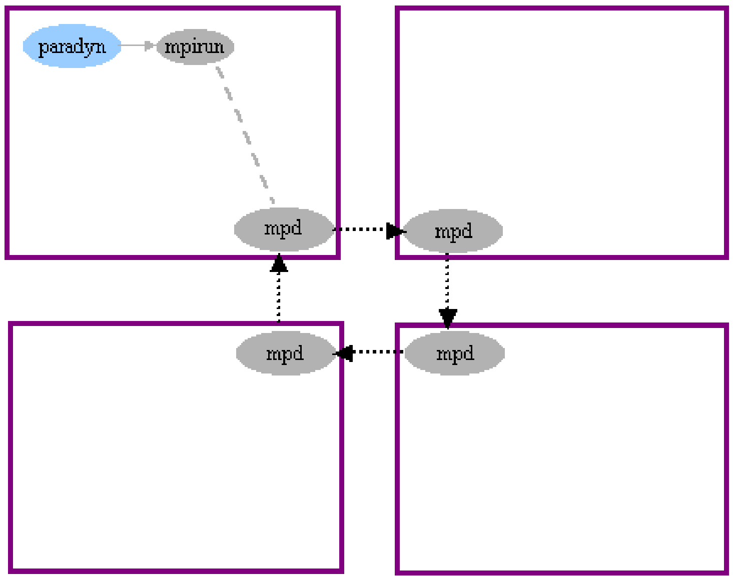
Figure 11: Paradyn/MPICH ch_p4 Starting the MPICH Application

The top figure shows that the Paradyn frontend instructs the daemons to start the MPI processes. The bottom diagram shows the exchange of information between the MPI processes during `MPI_Init`. The processes are still under the control of the Paradyn daemons.

3.2.2 Paradyn and the MPICH ch_p4mpd Device

Here we discuss the startup procedure for the MPICH ch_p4mpd device under Paradyn. On the surface, running MPICH using the mpd daemons with Paradyn is the same as running with the non-daemon form of MPICH. The user still specifies startup parameters on the Paradyn user interface, such as working directory, host of the master MPICH program, and the `mpirun` command. However, behind the scenes, the situation is much different. In the first place, the user must make sure that the mpd daemons are running before invoking the `mpirun` command. An mpd daemon must be running on every node that is to be a part of the MPICH application. These daemons are connected in a ring as seen in the top diagram of Figure 12.

After the mpd daemons are started, the user invokes Paradyn. Paradyn begins by forking the mpirun process if it is to be started on the same node, or by executing a remote shell command if it is to be started on a different node. As in the case with the MPICH ch_p4 device, Paradyn tells mpirun that the executable to be started is the Paradyn generated script that will start the Paradyn daemons. The mpirun process then connects to the mpd daemons and informs them to start the script. The dotted arrows between the mpd daemons in the top diagram of Figure 12 demonstrate their ring of communication. The solid arrow from the Paradyn frontend to the mpirun process denotes a parent-child relationship, and the dashed line between the mpirun process and the mpd daemon shows a communication connection.



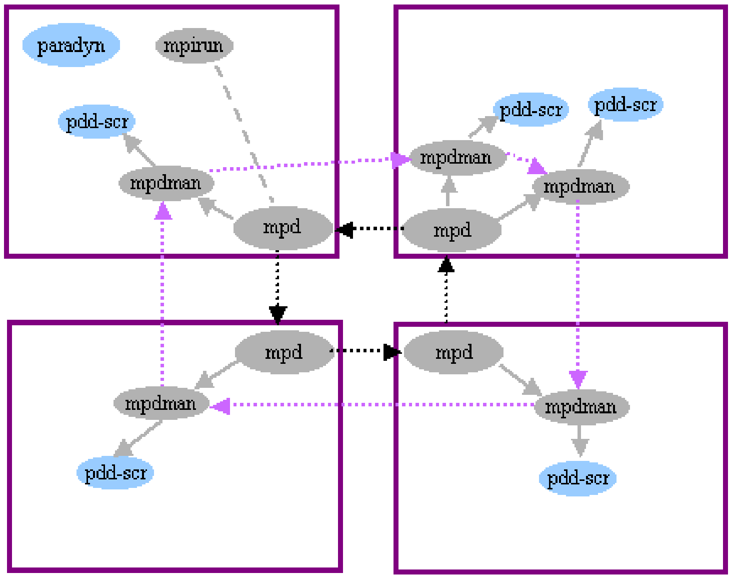


Figure 12: Paradyn/MPICH ch_p4mpd Initial Startup
The top figure shows the first steps in starting an MPICH ch_p4mpd program with Paradyn. First, the user sets up the mpd daemons. Then the user invokes Paradyn, giving it the `mpirun` command to start the MPI program. Paradyn starts mpirun, which connects to its local mpd daemon. The bottom diagram shows that the mpd daemons have launched mpdman processes, one for each MPI process. The mpdman processes each start a Paradyn generated script.

The mpd daemons then fork manager processes called mpdman. One mpdman process is forked for each instance of the MPICH program that will make up the application. The manager processes are forked consecutively around the ring, starting with the “next” node from the one on which the mpirun process was started, wrapping around the ring if necessary. The manager processes form a communication ring among themselves. Each manager then starts an instance of the Paradyn generated script. This series of events is portrayed in the bottom diagram of Figure 12. In the figure, the situation where more than one MPI process is to be started on a node is shown in the upper right node, where two mpdman processes are forked. The dotted arrows between the mpdman processes shows their communication ring.

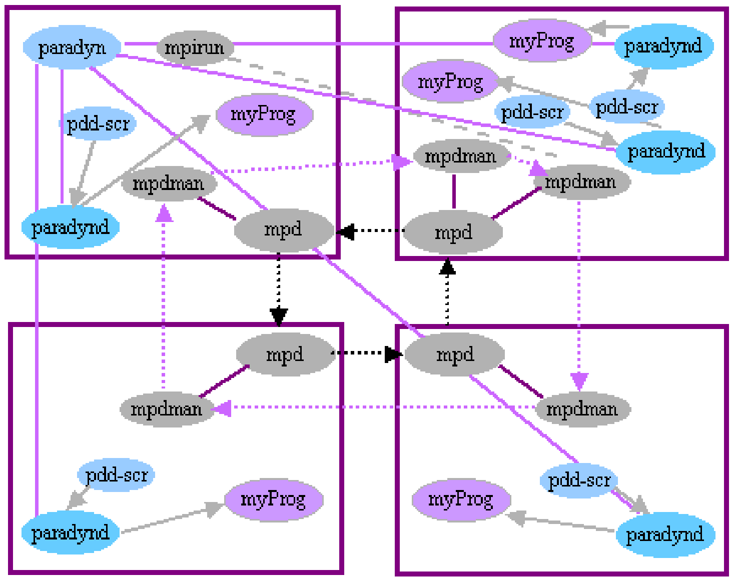
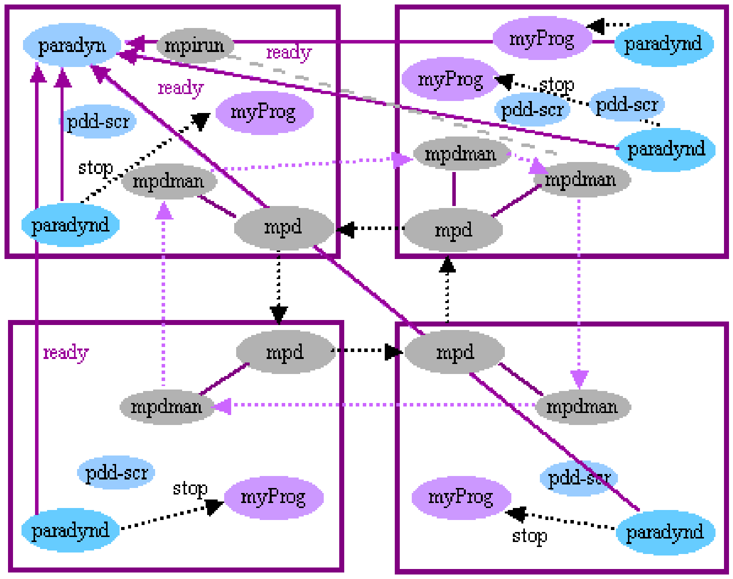


Figure 13: Paradyn/MPICH ch_p4mpd Starting the MPICH Processes
Each Paradyn script starts a Paradyn daemon. Each Paradyn daemon starts an MPI process.

At this point the mpirun process disconnects from the mpd daemon and connects to the first mpdman process that was started. This mpdman process is the manager of the master MPICH process. Each instance of the Paradyn generated script starts a Paradyn daemon. These daemons then open up communication sockets with the Paradyn frontend. The Paradyn daemons then fork the slave MPICH processes. These events are illustrated in Figure 13. The solid lines between the mpdman processes and the mpd daemons show open communication between them. The solid lines between the Paradyn daemons and the Paradyn frontend represent sockets for communication.

A communication pipe is established between the MPI processes and their parent Paradyn daemons. The MPI processes are stopped in `main`. The Paradyn daemons communicate to the Paradyn frontend that they are ready. In the top diagram of Figure 14, the dotted arrows between the Paradyn daemons and the MPI processes indicate Paradyn control of the MPI processes.

At this point, the “Run” button in the Paradyn user interface is enabled. When the user clicks on it, the Paradyn frontend instructs the Paradyn daemons to continue the MPI processes. The MPI processes are instructed to continue and begin to execute. Communication between the MPI processes is managed by the mpdman processes. The bottom diagram of Figure 14 depicts this series of events.



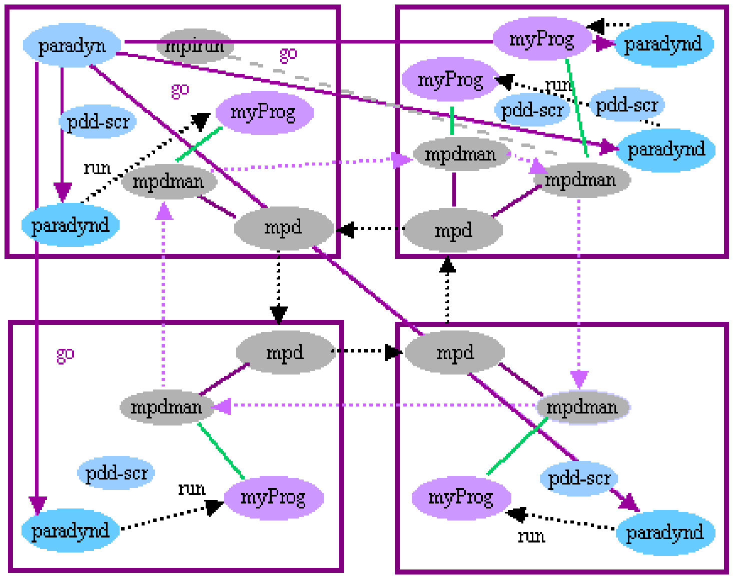


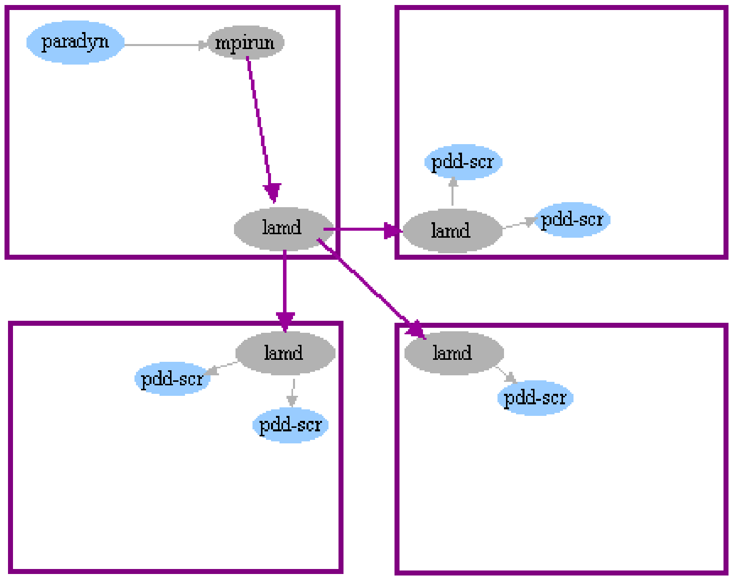
Figure 14: Paradyn/MPICH ch_p4mpd Initializing Paradyn Runtime

The top figure shows that the Paradyn daemons stop the MPI processes in `main`. They report back to the Paradyn frontend that they are ready. Then, the Paradyn frontend enables the "Run" button on the user interface. The bottom diagram show that when the user presses the "Run" button, the Paradyn frontend instructs the Paradyn daemons to continue the MPI processes.

3.2.3 Paradyn and LAM/MPI

The steps for Paradyn startup of LAM/MPI programs is similar to the startup of MPICH ch_p4mpd programs by Paradyn. The LAM environment is started by the user as described in Section 2.5.3. The end result of this is that there is a lamd process on every node in the LAM environment. The user invokes Paradyn, giving it the `mpirun` command to start the MPI program. Paradyn forks the mpirun process, substituting the executable argument given by the user for a Paradyn generated script, represented by *pdd-scr* in the top diagram of Figure 15. The script contains commands to start a Paradyn daemon. The arguments to the Paradyn daemon startup command inform the Paradyn daemon of the MPICH program that the user specified. The mpirun process instructs the LAM daemons to start the Paradyn script on the nodes. This series of events is depicted in the top diagram of Figure 15. The dark solid arrows in this Figure represent communication over sockets. The lighter arrows represent a parent-child relationship.

The Paradyn scripts then execute Paradyn daemons, instructing them to start the MPI processes that the user specified. The Paradyn daemons establish communication sockets with the Paradyn frontend. Then, the Paradyn daemons fork the MPI processes. This series of events is shown in the bottom diagram of Figure 15. The darker solid lines depict communication sockets. The light arrows represent parent-child relationships between processes.



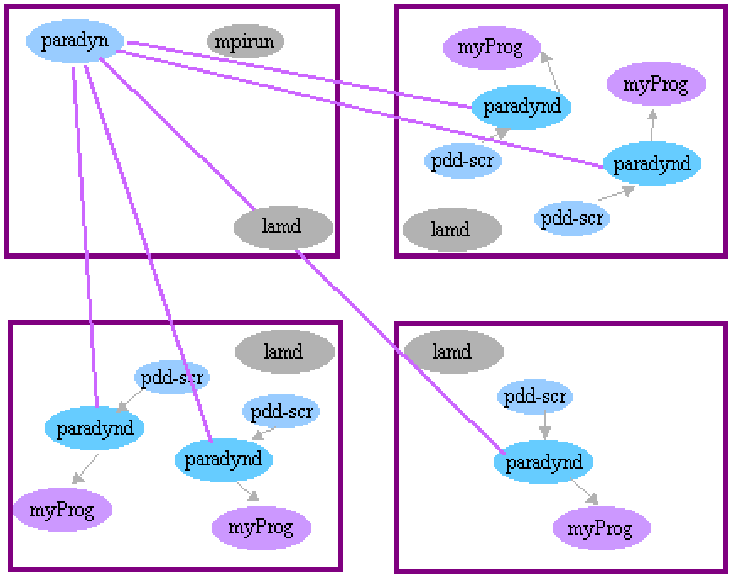
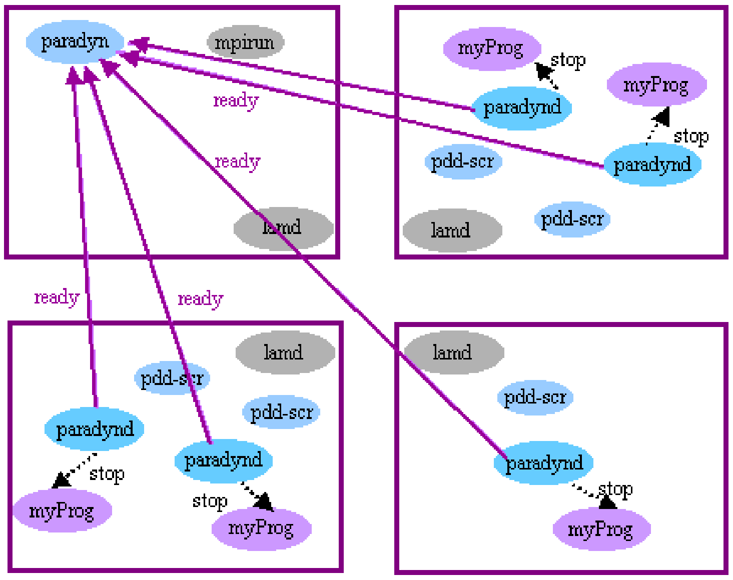


Figure 15: Paradyn/LAM/MPI Starting the MPI Processes
The top diagram shows the initial steps for starting a LAM/MPI program with Paradyn. The user first sets up the LAM session, then invokes Paradyn, giving it the `mpirun` command. Paradyn tells mpirun to start a Paradyn generated script instead. Because of this, the LAM daemons launch this script. The bottom diagram shows that the Paradyn generated scripts execute the Paradyn daemons. Then, each of these daemons starts an MPI process.



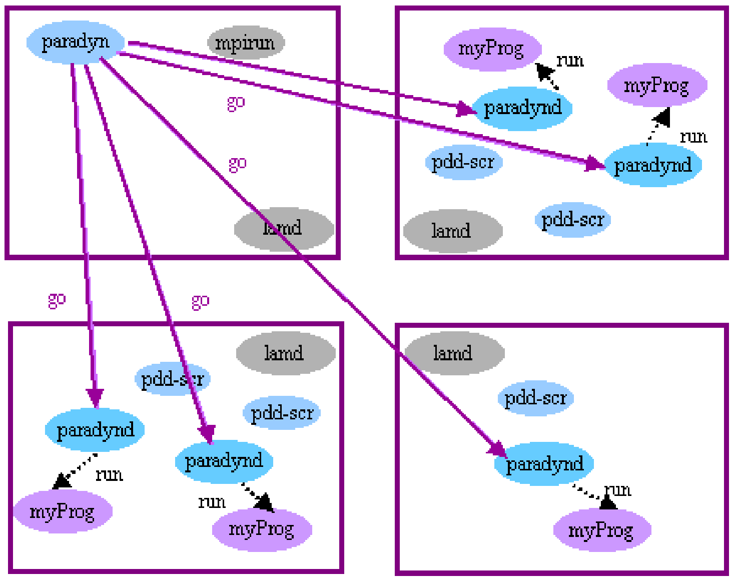


Figure 16: Paradyn/LAM/MPI Intitialization Complete
The top diagram shows that the Paradyn daemons stop the MPI processes in `main`. Then they report back to the Paradyn frontend that they are ready. The Paradyn user interface enables the “Run” button. The bottom diagram shows that after the user hits the “Run” button, the Paradyn frontend instructs the Paradyn daemons to continue the MPI processes.

A communication pipe is established between the MPI processes and the paradyn daemons. The MPI processes are told to stop. At this point, the Paradyn daemons communicate to the Paradyn frontend that they are ready. The dotted arrows between the Paradyn daemons and the MPI processes indicate Paradyn control of the MPI processes. After receiving the message that the daemons are ready, the Paradyn frontend enables the "Run" button user interface. The top diagram of Figure 16 shows these events.

When the user hits the "Run" button, this causes the Paradyn frontend to tell the Paradyn daemons to continue the MPI processes. The MPI processes establish themselves with the LAM daemons in `MPI_Init`. When initialization is complete, the program returns from `MPI_Init` and begins to execute the user's code, as shown in the bottom diagram of Figure 16.

4 Related Work

There are several options available to MPI programmers when faced with the task of optimizing the performance of their programs. In general, there are two types of applications to help: debuggers and performance tools. Debuggers allow the programmer to capture the state of their program at a specific time, while performance tools usually provide measurements of performance aspects of the program execution.

Some debuggers designed specifically for message-passing are TotalView [ELLC03], MQM (Message Queue Manager) [PTC03], and Panorama [MB93]. These products provide the capabilities of examining message queues at particular points in time, as well as stepping through sections of code. While these are useful for determining performance problems, it can be tedious to find the information in this manner, mainly due to the large number of tasks.

Most parallel performance tools are of the post-mortem viewers of trace data. Generally, a static visualization of the data is provided to help the programmer more easily understand the performance of the program. Tools of the post-mortem variety include Jumpshot [ZLGW99], Vampir [NAW+96], ParaGraph [HF93],and AIMS [Yan94]. Post-mortem viewers do not provide the flexibility of a tool like Paradyn that uses dynamic instrumentation. The decision on what to measure must be made before the program starts and cannot be changed during execution as it can with Paradyn. This limits optimizations for the amount of performance data that must be collected, and also is not as convenient for the user. The user has to re-run the program to alter what performance data is being collected. Jumpshot and Vampir are viewers of static post-mortem data. Some post-mortem tools also provide an animation of trace data. Examples are ParaGraph and AIMS. Because these tools use trace data, they are able to gather very detailed information about program execution. However, collecting such detailed data increases the possibility of generating unmanageably large trace files.

Another post-mortem performance viewer is mpiP, developed at Lawrence Livermore National Laboratory [VM01]. It is a profiling tool as opposed to a tracing tool, and reportedly collects a relatively constant amount of performance data, regardless of the program's execution time. Thus, it is more scalable to long-running programs than are tracing tools. The tool is able to present a statistical analysis of which MPI functions are using most of the program's time, and breaks them down by callsite and MPI rank. However, unlike Paradyn, mpiP does not present a program callgraph, but only identifies the parent function of the MPI calls. It does not present information on how the performance data may change over time like Paradyn does, but gives a statistical analysis of the program over the entire execution.

Performance toolkits aim to be more than just simple performance tools, and provide a variety of tools in a kit to help the user understand the performance of the program. Examples are TAU [MBM94] and Pablo [RAN+93]. TAU is a post-mortem analyzer that also supports dynamic instrumentation.

However, the level of support for dynamic instrumentation is not nearly at the same level as it is in Paradyn. The user is not able to direct TAU to insert instrumentation for specific metrics during runtime. TAU lacks the automatic diagnosis features of Paradyn. Pablo is also a kit of tools to be used for performance diagnosis of programs. While it provides features such as source instrumentation on the level of loops and basic blocks, it does not have much support for the Linux platform.

A post-mortem analysis tool that also provides performance monitoring of a running program is XMPI, which is distributed by LAM/MPI as an environment for running, debugging, and visualizing MPI programs in the LAM environment [LTJ03]. The user is not required to complete any instrumentation steps, such as recompiling or linking; the MPI program must simply be started by XMPI. The execution trace can be viewed at runtime or post-mortem. This tool is still in the early stages of development. A beta version of XMPI only supports LAM/MPI 6.5.9. Also, because it is a tracing tool means that the potential for large trace file problems exists.

There are other tools in addition to Paradyn that provide automated analysis of the performance data, to relieve the programmer from having to process the program execution data manually. Examples are Kappa-Pi [EM98], Peridot [WM01], KOJAK [WM00], and Prophesy[TWS03]. Both Kappa-Pi and Peridot are designed to measure the performance of message-passing programs. Each has a scheme similar to Paradyn's for searching for performance bottlenecks. However, both are tracing tools. The downside of this is the potential for problems with very large trace files, such as those we encountered using the MPE library. Neither of these tools has been released yet. KOJAK is a project from the Research Centre Juelich whose goal is a generic automatic performance analysis environment for parallel programs [WM00]. This product has been released and supports MPI on the Linux platform. However, KOJAK does not provide the performance analysis at runtime; the user must wait until the program execution is completed for KOJAK to begin its analysis of the trace data. Prophesy is an automated performance tool that allows the user to utilize performance data from multiple executions in the computation of the program performance model. The performance models generated by Prophesy can be used to predict program performance on different platforms. Prophesy does not give runtime performance analysis like Paradyn, but analyzes the programs post-mortem.

We found only three tools that support MPI-2 features of MPI. Vampir supports MPI-I/O. It provides trace information of the MPI-I/O operations and statistics such as operation count, bytes read/written, and transmission rate. However, Vampir is a post-mortem viewer of performance data, and as such does not allow the flexibility of runtime performance viewing. It also does not provide any automated performance diagnosis. The Totalview debugger supports the naming of communicator objects, so it can display user-defined names for communicators in the user interface. Pablo supports the MPI-I/O features of

MPI-2. However, Pablo utilizes source code instrumentation, so the user cannot change what performance data is collected at runtime as can be done with Paradyn. Also, Pablo does not include support for the Linux platform. We believe that an implementation of our changes for MPI-2 in Paradyn would be the first performance tool of its kind to support MPI-2.

5 Alterations Made to Paradyn For MPI-1

We made several alterations to Paradyn in order to achieve support for MPI-1 applications. Section 5.1 describes the changes that were common to both MPI implementations used in this project. Section 5.2 discusses changes for MPICH. Section 5.3 outlines the alterations we made to support LAM/MPI.

5.1 Alterations Common to Both MPI Implementations

We added three environment variables for this additional support. The first is PARADYN_SHARED_FILESYS. If this variable is set to 'false', then support for a non-shared filesystem is enabled. The next environment setting, PARADYN_MPI, determines which implementation of MPI is being used. It can be set to either 'LAM' or 'MPICH'. The last variable is PARADYN_MACHINEFILE and it is set to the full path location of a listing of the machines to be used for the MPI program. The format of the file is MPI implementation dependent. If 'LAM' is specified for PARADYN_MPI, then the file should conform to the machinefile format specified by LAM/MPI, and likewise for MPICH. This variable is not required if the user chooses to give the machinefile on the command line with MPICH. However, LAM/MPI does not allow a machine listing to be given on the command line, so the variable must be defined when using LAM/MPI.

5.2 Addition of Support for MPICH

We made alterations to Paradyn for MPICH. First, we changed Paradyn to support a non-shared filesystem. Then, we altered the metric definitions file for Paradyn to enable complete measurement of MPICH performance. We also spent considerable time diagnosing a problem that Paradyn 4.0 had with the MPICH ch_p4 device on our system. We do not include discussion of the changes to Paradyn for the ch_p4 device in this document as they are not directly relevant to this thesis. This section first describes the changes to Paradyn for support of a non-shared filesystem with the ch_p4mpd device and then outlines the metrics definitions changes that we made for MPICH.

In order to support a non-shared filesystem with Paradyn, we need to ensure that the Paradyn generated script, as described in Section 3.2, exists on each node that will have an MPI process running on it. We determine which machines need to have the file, and then copy the file out to the correct working directory on those nodes. To discover the nodes that need the file, we inspect the command line arguments to `mpirun`. The MPICH ch_p4mpd device has a relatively simple set of command line arguments to specify where to start the MPI processes.

`-np <n>`: number of processes to start
`-g <group_size>`: start group_size processes per mpd daemon
`-m <machinefile>`: a listing of the machines to be used
`-1`: do not start first process locally
`-wdir <directory>`: specifies directory for program

Without our changes, Paradyn only supported the `-np` argument. We added code to process the `-m` and `-wdir` arguments. To discover the nodes that need a copy of the Paradyn generated file, we begin by parsing either the command line supplied machinefile or, if that is not given, by looking at the file referenced by PARADYN_MACHINEFILE. Next, we inspect the value given to the `-np` argument (`n`) to find out how many processes were going to be run. Then, Paradyn copies its generated script out to the first `n` machines into the directory specified by the user. The user is allowed to override the directory specified to Paradyn in the process definition with the `-wdir` argument to `mpirun`. In that case, Paradyn copies the generated script file to the directory specified to `mpirun`.

After we completed the alterations to Paradyn for support of a non-shared filesystem, we discovered that we were unable to gather data for MPI performance metrics for MPICH C/C++ programs. We found the cause of this to be the way that the MPICH implementation chose to support the MPI Profiling Interface. The MPI Specification requires that every MPI routine be callable by an alternative name for profiling purposes. The Forum declared that each MPI routine also be accessible with a PMPI prefix. For example, `MPI_Send` must also be callable by the name `PMPI_Send`. The purpose of this is to provide a mechanism by which users can write profiling wrapper routines for the MPI functions. By default, the MPICH implementation uses weak symbols to support this requirement. The use of weak symbols means that a program is able to override an external identifier already defined in a library; the linker will not complain that there is more than one definition of an external symbol. The MPICH implementation uses a directive to tell the compiler that, for example, `PMPI_Send` is a weak symbol for `MPI_Send`. When the user calls `MPI_Send` in their application, it resolves to the definition for `PMPI_Send`. However, when the user links in the MPI profiling library, that library has a definition for `MPI_Send`. In this case, when the user calls `MPI_Send`, it resolves to the strong symbol for `MPI_Send` in the profiling library. The `MPI_Send` in the profiling library is a wrapper that does some performance measurement and then calls `PMPI_Send`. The designers of the MPICH implementation do this to reduce the size of the MPICH library. Otherwise, two copies of the library need to be compiled, one for each callable name. A user can override MPICH's default behavior and make two copies of the library by giving the `--disable-weak-symbols` flag to `configure` during compilation.

When MPICH is installed using the default configuration, the symbols for the MPI routines in the binary image of an MPICH program resolve to their PMPI counterparts. The MPI metrics definitions in Paradyn 4.0 did not account for this completely. The metric definitions included the profiling function names for Fortran programs, but not for programs written in C/C++. This limited the performance data we could gather for MPICH programs written in C/C++ on our system. To overcome this, we added the C family forms of the PMPI function names to Paradyn's metric definitions file. For this task, the challenge was in finding the source of the problem, whereas the actual addition of the C family forms of the functions to the metrics definitions was trivial.

5.3 Considerations for LAM/MPI

When embarking upon this project, LAM/MPI was in version 6.5.9 and Paradyn was in version 3.0. Paradyn 3.0 was unable to start and instrument LAM/MPI programs. Because of this, considerable time was spent understanding both programs to see how they could be made to cooperate with each other. We never fully determined why the pair would not work together. One reason could be that in LAM/MPI 6.5.9, when the MPI processes were started by the Paradyn daemons, the LAM processes were not able to properly install themselves with the LAM daemons. In any case, we did design two scenarios for Paradyn to start LAM/MPI applications. Both designs involved having the Paradyn daemons attach to the already running MPI programs.

However, at about the same time, both software groups released new versions, LAM/MPI 7.0 and Paradyn 4.0. A simple test run showed that whatever problems prevented the compatibility of the two previous versions no longer existed. What remained was to accommodate LAM/MPI's more extensive set of command line arguments to `mpirun` and to verify that Paradyn was correctly measuring the metrics of LAM/MPI programs.

LAM/MPI has a comparatively robust and flexible set of arguments to `mpirun` that allow the user to specify where the MPI processes should be started. The machines and processors in the system are defined in a startup file that is given to `lamboot`. The nodes are indexed in the order they are listed in the machinefile. LAM/MPI allows the user to specify how many processors each machine has in the machinefile. They can do this by putting an explicit `cpu=x` next to the machine name in the machinefile, where `x` is a number representing processor count, or by listing the machine's name multiple times, once for each processor in the machine. Each processor in the system is given an also index, in order of listing in the machinefile.

There are four different ways to specify the number of MPI processes to be started:

1. By direct CPU count: For direct CPU count, the command line argument `-np n` argument simply denotes that n processes be started on the first `n` processors.
2. By node specification: For node specification, there are two options. The user can give the argument `N` to `mpirun`, which means to run one copy of the process on each node in the LAM session. The user can also designate a subset of the nodes using a LAM/MPI specific notation of the form `nR[,R]*`, where `R` denotes a range of nodes within the defined number of nodes, [0, num_nodes). For example, the user could specify `n0-2,4`, which would start an MPI process on nodes 0,1,2, and 4.
3. By processor specification: For processor specification, there are two options. The command line argument `C` tells LAM/MPI to start one MPI process on every processor in the LAM session. The user can also indicate a subset of processors by using a notation like the one for selecting nodes. The specification is of the form `cR[,R]*`, where this time, `R` denotes a range of processors within the defined number of CPU's [0, num_cpus). It is also possible for the user to give a mixture of node and processor specifications on the command line.
4. By application schema: An application schema is text file in which users can indicate where MPI processes should be started. An application schema allows even

more flexibility for the user. It provides support for heterogeneous systems and multiple binaries in the MPI program

We altered Paradyn to support the first three of these. We chose them because they are the most commonly used for running LAM/MPI programs.

We determine which machines will need a copy of the Paradyn generated script by parsing the command line arguments and mapping them to the specified nodes, according to the list referenced by PARADYN_MACHINEFILE. Then, Paradyn copies the script to those nodes. It should be noted that this mechanism will not work if the user opts to `lamgrow` and/or `lamshrink` the LAM session without also changing the file pointed to by PARADYN_MACHINEFILE.

No other changes are required for Paradyn to support the MPI-1 features of LAM/MPI. We performed several tests which show Paradyn is instrumenting and measuring the performance of LAM/MPI programs. The tests and their results are given in Chapter 7.

6 Additions to Paradyn for Support of MPI-2 Features

This chapter discusses items of interest for parallel performance tool support of MPI-2. Section 6.1 discusses which MPI-2 features are most important for consideration by performance tool developers. Section 6.2 describes our proposed changes to Paradyn for support of MPI-2, including new metrics and changes to both the Performance Consultant and Paradyn's Where Axis.

6.1 Discussion of MPI-2 Features Important for Performance Tool Developers

The most important new functionalities of MPI-2 that are of interest to performance tool developers are:

- dynamic process creation,
- RMA,
- MPI-I/O,
- thread support,
- the ability to name MPI objects, and
- language mixing.

The first four of these features are likely to have performance impacts on MPI programs, potentially positive and negative. The last three features are important in that they may effect the internal structure of performance tools used for MPI programs. The following paragraphs discuss each of these features in turn and point out the topics of interest to performance tool developers.

Measuring the performance of dynamic process creation is important because these operations could represent serious performance bottlenecks if used incorrectly. First of all, a spawning operation includes the time to start the child processes, which is non-trivial in itself. Secondly, the operation is collective over two communicators, those of the parent group of processes and the child group of processes. The potential synchronization required for this operation could be time consuming. We believe that MPI programmers will want to know the specific performance costs to their programs from these operations.

RMA provides a mechanism by which MPI programmers can improve the communication performance of some programs. However, the RMA interface is quite flexible, so it is possible the programmer could use a suboptimal combination of the functions provided. Also, the fact that the interface contains collective operations means that synchronization bottlenecks can occur. MPI programmers who use this feature will be interested in optimizing the communication performance of their programs.

File I/O has traditionally been a performance bottleneck for programs. MPI programmers can improve performance by utilizing the parallel file I/O operations included in MPI-2. The MPI-I/O interface is extensive, allowing the programmer to find the best combination of file operations for the program. In addition, there are many options for the Info argument for this feature. These flexibilities increase the chances that a less than optimal combination could be chosen. Programmers will desire performance measurement for MPI-I/O to help find the best combinations of file operations and access settings.

Features that require consideration from the perspective of performance tool internal structure are: thread support, the naming of MPI objects, and language mixing. The addition of thread support means that performance tools for MPI programs should support multi-threaded applications. The ability to name MPI objects is of importance, because the performance tool can display the user defined names for MPI objects in the user interface. This will facilitate user's interpretation of the performance data. Language mixing could have an effect on how the programs are instrumented, especially for those that do automated source-level instrumentation. Performance tools will need to support programs with source files written in different languages.

6.2 Design for MPI-2 Feature Performance Measurement in Paradyn

In this section, we outline our proposed changes to Paradyn for support of MPI-2 features. Section 6.2.1 lists and describes the metric changes we propose. Section 6.2.2 shows our changes for Paradyn's Hypothesis Hierarchy. Last, in Section 6.2.3 we give our changes to Paradyn's Where Axis.

6.2.1 Metric Changes

We propose new metrics to Paradyn to enable the performance measurement of some of the more important MPI-2 features. We designed metrics for dynamic process creation, RMA, and MPI-I/O. Table 3 shows the new metrics for dynamic process creation. Tables 4-8 show the metrics proposed to measure the performance of RMA. Tables 9-13 list the metrics for MPI-I/O.

Table 3: Dynamic Process Creation Metrics

Metric	Description	MPI Functions
Operation Counts		
spawn_count	A count of the number of times per unit time a spawning operation occurs	`MPI_Comm_spawn`, `MPI_Comm_spawn_multiple`
Synchronization Time		
spawn_time	Inclusive wall clock time spent in spawning operations	`MPI_Comm_spawn`, `MPI_Comm_spawn_multiple`

We created the metric spawn_count for counting spawning operations per unit time, because applications that spawn processes are likely to use a runtime calculation for determining when and/or how many processes will be started. The user may wish to gather performance data with this metric to determine when and how many processes are being started in the application.

The time required for spawning processes is likely to be significant. For this reason, we created the metric spawn_time to measure the wall clock time spent in spawning operations. A spawning operation is collective over two communicators and thus may incur synchronization overhead, which could include the time for all processes to be started. The tool user may wish to collect performance data pertaining to the time spent in spawning operations.

Table 4: RMA Metrics for Operation Counts

Metric	Description	MPI Functions
rma_put_ops	A count of the number of Put operations per unit time. Aggregation is total Put operations.	`MPI_Put`
rma_get_ops	A count of the number of Get operations per unit time. Aggregation is total Get operations.	`MPI_Get`
rma_acc_ops	A count of the number of Accumulate operations per unit time. Aggregation is total Accumulate operations.	`MPI_Accumulate`
rma_ops	A count of the number of Put, Get, and Accumulate operations per unit time. Aggregation is total RMA operations.	`MPI_Put, MPI_Get, MPI_Accumulate`

We created metrics in Table 4 for counting RMA operations so that users could collect performance data about the number of RMA operations that occur in a unit of time. There are individual metrics for counting each of the RMA data transfer routines, and one metric, rma_ops, that counts all of the data transfer operations.

Table 5 shows metrics that count the bytes of data transferred per unit time as a result of RMA operations. There are metrics for counting the bytes due to each of the RMA data transfer operations individually. Also, there is a general byte-counting metric, rma_bytes, that represents of all the data transfer operations.

Table 5: RMA Metrics for Bytes Transferred

Metric	Description	MPI Functions
rma_put_bytes	Number of bytes put per unit time. Aggregation is total bytes put.	`MPI_Put`
rma_get_bytes	Number of bytes gotten per unit time. Aggregation is total bytes gotten.	`MPI_Get`
rma_acc_bytes	Number of bytes accumulated in the target process. Aggregation is total bytes accumulated.	`MPI_Accumulate`
rma_bytes	Sum of RMA byte count metrics. Aggregation is total RMA bytes.	`MPI_Put, MPI_Get, MPI_Accumulate`

The next set of metrics are for the measurement of synchronization time due to RMA operations. Although RMA is designed to reduce the synchronization overhead of data transfer operations in MPI, there will still be some synchronization time, the amount and distribution of which is largely dependent upon the MPI implementation.

Table 6 shows the metrics we designed for RMA active target synchronization, at_rma_sync_wait and at_rma_sync_wait_incl. These metrics represent the wall clock time spent in the MPI functions listed in Table 6. The data collected with the inclusive metric, at_rm_sync_wait_incl, includes not only the time spent in these MPI functions but also the time spent in any routines called by those functions.

We selected these functions for active target synchronization waiting time based on the possibility that they could block, waiting on a state change of another process.

MPI_Win_fence could incur synchronization waiting time as it is a collective call. Also, the MPI-2 Standard states that it will usually act as a barrier routine, which means that the synchronization overhead could be particularly high. The function MPI_Win_start could cause synchronization waiting time as it is allowed to block until matching MPI_Win_post calls have been executed on each process in the target group. In fact, any of the routines, MPI_Win_start, MPI_Win_complete, MPI_Put, MPI_Get, or MPI_Accumulate are allowed to block until the corresponding MPI_Win_post has been issued on the target processes. Thus, any of them could contribute to synchronization waiting time. However, the data transfer routines, MPI_Put, MPI_Get and MPI_Accumulate are not included in the active target metrics even though they could contribute to synchronization time. They are included with the general RMA metrics found in Table 8. The reason for this is that it is impossible to distinguish between a data transfer routine being used in active target synchronization versus passive target synchronization just by looking at the function state itself. The MPI-2 Standard says that the function MPI_Win_wait will block until all outstanding MPI_Win_complete calls have been issued, and as such could add to the synchronization waiting time, so it is incorporated into the active target metrics.

Table 6: RMA Metrics for Active Target Synchronization

Metric	Description	MPI Functions
at_rma_sync_wait	Wall clock time spent in active target RMA synchronization routines during time interval. Aggregation is total active target synchronization time.	`MPI_Win_fence, MPI_Win_start, MPI_Win_complete, MPI_Win_wait`
at_rma_sync_wait_incl	Inclusive wall clock time spent in active target RMA synchronization routines during time interval. Aggregation is total inclusive active target synchronization time.	

Table 7: RMA Metrics for Passive Target Synchronization

Metric	Description	MPI Functions
pt_rma_sync_wait	Wall clock time spent in passive target RMA synchronization routines during time interval. Aggregation is total passive target synchronization time.	`MPI_Win_lock, MPI_Win_unlock`
pt_rma_sync_wait_incl	Inclusive wall clock time spent in active target RMA synchronization routines during time interval. Aggregation is total inclusive passive target synchronization time.	

The passive target RMA metrics, pt_rma_sync_wait and pt_rma_sync_wait_incl, are shown in Table 7. The passive target metrics give the wall clock time spent in the passive target RMA routines shown in Table 7 per unit time. The metric pt_rma_sync_wait_incl not only includes the time spent in the passive target routines, but also the wall clock time spent in any functions called by those routines. The functions `MPI_Win_lock, MPI_Win_unlock, MPI_Put, MPI_Get`, or `MPI_Accumulate` could all incur synchronization waiting time. However, the data transfer routines are not included in the passive target metric. They are included in the general RMA synchronization metrics found in Table 8, because the data transfer routines could be used in both passive target and active target synchronization. The MPI-2 Standard requires that `MPI_Win_unlock` not return until the data transfer is complete at both the origin and target. The Standard also says that `MPI_Win_lock` or the data transfer routine could block until the lock is acquired at the target. For these reasons, these functions could both contribute to passive target synchronization waiting time.

The metrics for overall RMA synchronization wall clock time are shown in Table 8. For the most part, the functions included for these metrics are the passive target and active target synchronization routines. Also included are `MPI_Win_create` and `MPI_Win_free`. `MPI_Win_create` is collective and thus carries the possibility of synchronization overhead. The MPI-2 Standard states that `MPI_Win_free` requires a barrier synchronization; thus it will incur synchronization waiting time. The metric rma_sync_wait_incl not only includes the wall clock time spent in the functions in Table 8, but the time spent in any routine called by those functions. Also, the data transfer routines are included in the general RMA metric as they could contribute to either passive target or active target synchronization.

Table 8: RMA Metrics for Overall Synchronization

Metric	Description	MPI Functions
rma_sync_wait	Wall clock time spent in RMA synchronization routines during time interval. Aggregation is total synchronization time.	`MPI_Win_fence`, `MPI_Win_create`, `MPI_Win_free`, `MPI_Win_start`, `MPI_Win_complete`, `MPI_Win_wait`, `MPI_Win_lock`, `MPI_Win_unlock`, `MPI_Put`, `MPI_Get`, `MPI_Accumulate`
rma_sync_wait_incl	Inclusive wall clock time spent in RMA synchronization routines during time interval. Aggregation is total inclusive synchronization time.	

The next set of metrics we created are for the performance measurement of MPI-I/O. These new metrics can be seen in Tables 9, 10, 11, 12, and 13. They were developed to keep track of the number of I/O operations, count the bytes transferred, and measure wall clock time of the operations.

The metrics in Table 9 are for keeping count of the number of MPI-I/O operations that occur per unit time. The metric par_read_ops counts the number of parallel read operations, while par_write_ops counts the write operations. Last, par_io_ops keeps track of all I/O read, write, and seek operations.

Table 9: MPI-I/O Metrics for Operation Counts

Metric	Description	MPI Functions
par_read_ops	A count of the number of parallel read operations per unit time. Aggregation is total read operations.	`MPI_File_read(_at)(_all), MPI_File_iread(_at), MPI_File_read_shared, MPI_File_read_ordered, MPI_File_read(_at)_all_begin, MPI_File_read_ordered_begin`
par_write_ops	A count of the number of parallel write operations per unit time. Aggregation is total write operations.	`MPI_File_write(_at)(_all), MPI_File_iwrite(_at), MPI_File_write_shared, MPI_File_write_ordered, MPI_File_write(_at)_all_begin, MPI_File_write_ordered_begin`
par_io_ops	A count of the number of parallel read and write operations per unit time. Aggregation is total read and write operations.	`MPI_File_read(_at)(_all), MPI_File_iread(_at), MPI_File_read_shared, MPI_File_read_ordered, MPI_File_read(_at)_all_begin, MPI_File_read_ordered_begin, MPI_File_write(_at)(_all), MPI_File_iwrite(_at), MPI_File_write_shared, MPI_File_write_ordered, MPI_File_write(_at)_all_begin, MPI_File_write_ordered_begin, MPI_File_seek, MPI_File_seek_shared`

Table 10: MPI-I/O Metrics for Bytes Transferred

Metric	Description	MPI Functions
par_read_bytes	Number of bytes read per unit time. Aggregation is total bytes read.	`MPI_File_read(_at)(_all)`, `MPI_File_iread(_at)`, `MPI_File_read_shared`, `MPI_File_read_ordered`, `MPI_File_read(_at)_all_begin`, `MPI_File_read_ordered_begin`
par_write_bytes	Number of bytes written per unit time. Aggregation is total bytes written.	`MPI_File_write(_at)(_all)`, `MPI_File_iwrite(_at)`, `MPI_File_write_shared`, `MPI_File_write_ordered`, `MPI_File_write(_at)_all_begin`, `MPI_File_write_ordered_begin`
par_io_bytes	Number of bytes read and written per unit time. Aggregation is total bytes read and written.	`MPI_File_read(_at)(_all)`, `MPI_File_iread(_at)`, `MPI_File_read_shared`, `MPI_File_read_ordered`, `MPI_File_read(_at)_all_begin`, `MPI_File_read_ordered_begin`, `MPI_File_write(_at)(_all)`, `MPI_File_iwrite(_at)`, `MPI_File_write_shared`, `MPI_File_write_ordered`, `MPI_File_write(_at)_all_begin`, `MPI_File_write_ordered_begin`

Table 10 shows the metrics designed for measuring the number of bytes transferred to and from a file. These metrics keep track of the bytes transferred for both collective and non-collective MPI-I/O read and write operations. The metric par_read_bytes counts the number of bytes read from files, while par_write_bytes counts those written to files. The general metric, par_io_bytes, counts the bytes that were read from or written to files with MPI-I/O.

The next set of metrics, shown in Table 11, are for the measurement of synchronization time due to MPI-I/O collective operations. The first four metrics are for the measurement of collective read and write synchronization times. The metrics cc_par_read_wait and cc_par_write_wait represent the wall clock time spent in

Table 11: MPI-I/O Metrics for Collective Operations Synchronization

Metric	Description	MPI Functions
cc_par_read_wait	Wall clock time spent in collective read routines during time interval. Aggregation is total collective read time.	`MPI_File_read(_at)_all`, `MPI_File_read_ordered`, `MPI_File_read(_at)_all_begin`, `MPI_File_read(_at)_all_end`, `MPI_File_read_ordered_begin`, `MPI_File_read_ordered_end`
cc_par_read_wait_incl	Inclusive wall clock time spent in collective read routines during time interval. Aggregation is total inclusive collective read time.	
cc_par_write_wait	Wall clock time spent in collective write routines during time interval. Aggregation is total collective write time.	`MPI_File_write(_at)_all`, `MPI_File_write_ordered`, `MPI_File_write(_at)_all_begin`, `MPI_File_write(_at)_all_end`, `MPI_File_write_ordered_begin`, `MPI_File_write_ordered_end`
cc_par_write_wait_incl	Inclusive wall clock time spent in collective write routines during time interval. Aggregation is total inclusive collective write time.	

Table 11: MPI-I/O Metrics for Collective Operations Synchronization

Metric	Description	MPI Functions
cc_par_seek_wait	Wall clock time spent in collective seek routines during time interval. Aggregation is total collective seek time.	`MPI_File_seek_shared`
cc_par_seek_wait_incl	Inclusive wall clock time spent in collective seek routines during time interval. Aggregation is total inclusive collective seek time.	`MPI_File_seek_shared`
cc_par_io_sync_wait	Wall clock time spent in all collective parallel file I/O routines during time interval. Aggregation is total parallel I/O time.	`MPI_File_read(_at)_all`, `MPI_File_read_ordered`, `MPI_File_read(_at)_all_begin`, `MPI_File_read(_at)_all_end`, `MPI_File_read_ordered_begin`, `MPI_File_read_ordered_end`, `MPI_File_write(_at)_all`, `MPI_File_write_ordered`, `MPI_File_write(_at)_all_begin`, `MPI_File_write(_at)_all_end`, `MPI_File_write_ordered_begin`, `MPI_File_write_ordered_end`, `MPI_File_seek_shared`, `MPI_File_open`, `MPI_File_close`, `MPI_File_set_size`, `MPI_File_preallocate`, `MPI_File_set_info`, `MPI_File_set_view`
cc_par_io_sync_incl	Inclusive wall clock time spent in all collective parallel file I/O routines during time interval. Aggregation is total inclusive parallel I/O time.	

MPI-I/O collective read and write functions. The metrics cc_par_read_wait_incl and cc_par_write_wait_incl record not only the wall clock time spent in the MPI-I/O collective read and write functions, but also the time spent in routines called by those functions. Next are the metrics for measuring collective seek time, cc_par_seek_wait and cc_par_seek_wait_incl. These metrics will give the user the amount of time spent in seek operations using a shared file pointer, with cc_par_seek_wait_incl including the time spent in routines called by `MPI_File_seek_shared`. The last two metrics in Table 11 are general metrics for measuring the time spent in all collective MPI-I/O routines, both excluding and including routines called by the functions. Because the MPI-I/O functions in Table 11 are all collective, the possibility of there being synchronization time due to the coordination of processes exists, as well as synchronization time due to resource contention.

Table 12 shows metrics for non-collective MPI-I/O operations. The first two

Table 12: MPI-I/O Metrics for Non-Collective Operations Synchronization

Metric	Description	MPI Functions
nc_par_read_wait	Wall clock time spent in non-collective read routines during time interval. Aggregation is total read time.	`MPI_File_read(_at)`, `MPI_File_read_shared`
nc_par_read_wait_incl	Inclusive wall clock time spent in non-collective read routines during time interval. Aggregation is total inclusive read time.	`MPI_File_read(_at)`, `MPI_File_read_shared`

Table 12: MPI-I/O Metrics for Non-Collective Operations Synchronization

Metric	Description	MPI Functions
nc_par_write_wait	Wall clock time spent in non-collective write routines during time interval. Aggregation is total write time.	`MPI_File_write(_at)`, `MPI_File_write_shared`
nc_par_write_wait_incl	Inclusive wall clock time spent in non-collective write routines during time interval. Aggregation is total inclusive write time.	`MPI_File_write(_at)`, `MPI_File_write_shared`
nc_par_io_sync_wait	Wall clock time spent in non-collective read and write routines during time interval. Aggregation is total and write time.	`MPI_File_read(_at)`, `MPI_File_read_shared`, `MPI_File_write(_at)`, `MPI_File_write_shared`
nc_par_io_sync_incl	Inclusive wall clock time spent in non-collective read and write routines during time interval. Aggregation is total inclusive read and write time.	

metrics, nc_par_read_wait and nc_par_read_wait_incl, measure the exclusive and inclusive wall clock time of the non-collective read operations, respectively. The metrics nc_par_write_wait and nc_par_write_wait_incl keep track of the wall clock time spent in non-collective write operations. The last two metrics in this category are for the general measurement of non-collective MPI-I/O synchronization. Only the non-collective read and write functions were included for this metric, instead of every single non-collective MPI-I/O function, as we believe only these functions will be responsible for non-collective synchronization waiting time.

We did not include the non-blocking file access operations in the non-collective read and write metrics. We feel that non-blocking MPI-I/O reads and writes require special consideration, because they are not complete until a corresponding positive `MPI_Test` or an `MPI_Wait` function returns. The non-blocking message-passing and I/O operations each have a ‘request’ object parameter. This object is then used in subsequent calls to `MPI_Test/MPI_Wait` as a way to identify the non-blocking request. Currently, Paradyn counts the `MPI_Wait` function as synchronization time due to message passing. This definition will have to be altered to account for the different uses of `MPI_Wait`. One way to handle this would be for Paradyn to detect the non-blocking calls and store the request objects organized by type of call: read, write, send, or receive. When an `MPI_Wait` call is encountered, Paradyn could find the matching request object and adjust the corresponding synchronization metric accordingly.

The last set of metrics are for general MPI-I/O synchronization, and include the collective and non-collective file operations. The first two metrics, par_io_read_sync and par_io_read_sync_incl, measure the wall clock time spent in parallel read operations, with par_io_sync_incl measuring the time spent in functions called by the read operations, as well as in the read operations themselves. The second metrics are

Table 13: MPI-I/O Metrics for Overall Synchronization

Metric	Description	MPI Functions
par_io_read_sync	Wall clock time spent in read routines during time interval. Aggregation is total read time.	`MPI_File_read(_at)(_all), MPI_File_iread(_at), MPI_File_read_shared, MPI_File_read_ordered, MPI_File_read(_at)_all_begin, MPI_File_read_ordered_begin`
par_io_read_sync_incl	Inclusive wall clock time spent in read routines during time interval. Aggregation is total inclusive read time.	
par_io_write_sync	Wall clock time spent in write routines during time interval. Aggregation is total write time.	`MPI_File_write(_at)(_all), MPI_File_iwrite(_at), MPI_File_write_shared, MPI_File_write_ordered, MPI_File_write(_at)_all_begin, MPI_File_write_ordered_begin`
par_io_write_sync_incl	Inclusive wall clock time spent in write routines during time interval. Aggregation is total inclusive write time.	

Table 13: MPI-I/O Metrics for Overall Synchronization

Metric	Description	MPI Functions
par_io_sync	Wall clock time spent in I/O synchronization routines during time interval. Aggregation is total I/O synchronization time.	`MPI_File_read(_at)(_all)`, `MPI_File_iread(_at)`, `MPI_File_read_shared`, `MPI_File_read_ordered`, `MPI_File_read(_at)_all_begin`, `MPI_File_read_ordered_begin`, `MPI_File_write(_at)(_all)`, `MPI_File_iwrite(_at)`, `MPI_File_write_shared`, `MPI_File_write_ordered`, `MPI_File_write(_at)_all_begin`, `MPI_File_write_ordered_begin`, `MPI_File_seek_shared` `MPI_File_open`, `MPI_File_close`, `MPI_File_set_size`, `MPI_File_preallocate`, `MPI_File_set_info`, `MPI_File_set_view`
par_io_sync_incl	Inclusive wall clock time spent in I/O synchronization routines during time interval. Aggregation is total inclusive I/O synchronization time.	

for write operations. As with the read metrics, the suffix _incl denotes that the metric is an inclusive measurement of functions called by the write operations. Lastly, there are exclusive and inclusive metrics for general parallel I/O synchronization. Included are all the collective and non-collective MPI-I/O functions which may cause synchronization overhead.

6.2.2 Hypothesis Hierarchy Changes

We propose three changes to Paradyn's Hypothesis Hierarchy, or Why Axis, for support of MPI-2 features. These changes will enable the Performance Consultant to automate performance analysis of the more important MPI-2 features.

1. Change top level hypothesis for ExcessiveSyncWaitingTime to include the synchronization metrics for spawning and RMA operations.
2. Change top level hypothesis for ExcessiveIOBlockingTime to include the metrics for MPI-I/O.
3. Add new hypotheses for Parallel and Non-Parallel I/O to the ExcessiveIOBlockingTime hypothesis hierarchy. The Parallel hypothesis compares the MPI-I/O synchronization metrics against a threshold. The Non-Parallel hypothesis will compare non-parallel file access metrics against a threshold.

Figures 17 and 18 illustrate the changes. Figure 17 shows Paradyn's Hypothesis Hierarchy as it exists currently. Figure 18 displays the hierarchy with our proposed

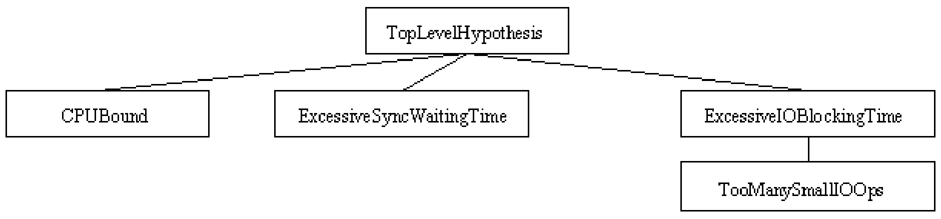


Figure 17: Current Paradyn Hypothesis Hierarchy
This figure shows the Hypothesis Hierarchy, or Why Axis, for the current version of Paradyn.

changes. We refine the ExcessiveIOBlockingTime hypothesis into two hypotheses for Parallel and Non-parallel file I/O. The new metrics for MPI-I/O are included into the top level hypothesis, ExcessiveIOBlockingTime. There are two new hypotheses under Parallel I/O. The first is TooManySmallIOOps, which will compare the MPI-I/O metrics for operation count and byte count against thresholds. The second is ExcessiveIO-

SyncTime, which will compare the MPI-I/O synchronization metrics against a threshold. We add the synchronization metrics for RMA and spawning operations to the ExcessiveSyncWaitingTime hypothesis.

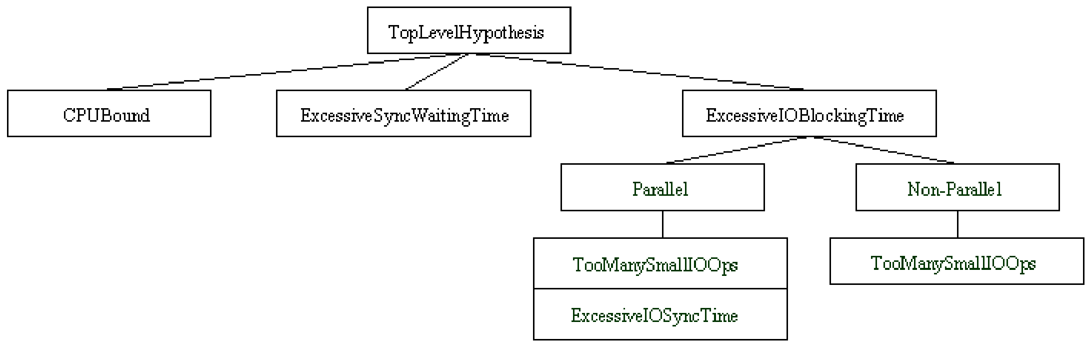


Figure 18: Changes to Paradyn's Hypothesis Hierarchy for MPI-2 Support
This figure shows our proposed changes to Paradyn's Hypothesis Hierarchy for MPI-2 support. The metrics for RMA and spawning synchronization time have been incorporated into the ExcessiveSyncWaitTime hypothesis. The metrics for MPI-I/O have been added to the ExcessiveIOBlockingTime hypothesis, which accounts for both Parallel and Non-Parallel I/O.

6.2.3 Where Axis Changes

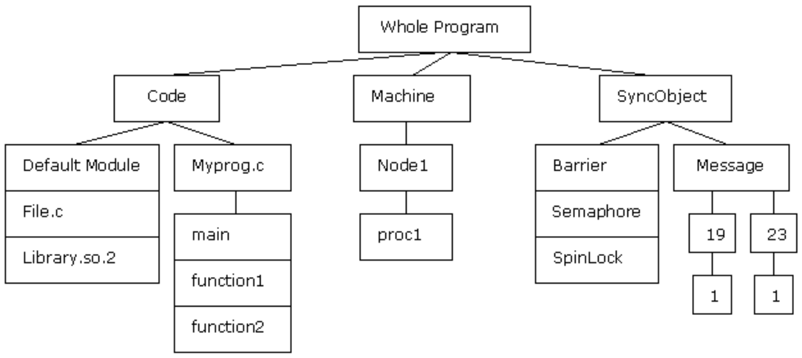
We designed five changes to Paradyn's Where Axis to support the new MPI-2 features.

1. Display user-defined names for MPI objects in the Paradyn interface.
2. Add communicator to the top level of the Where Axis, Comm, to enable more flexible focus selection. Also, remove communicator level from /SyncObject/Message.
3. Include intercommunicators and intracommunicators in the Comm hierarchy. Detect and incorporate newly created communicators from `MPI_Comm` operations.
4. Incorporate newly spawned processes into the Machine Hierarchy.

5. Add RMA Windows and File Handles to the /SyncObject resource hierarchy.

In Figure 19, we show mock-ups of Paradyn's Where Axis for a hypothetical MPI-2 program that spawns three child processes. The top diagram in Figure 19 shows the Where Axis as it is in the current version of Paradyn. In the bottom diagram of Figure 19, we show the Where Axis with our proposed changes to support MPI-2. One difference is that three newly spawned processes have been incorporated into the /Machine hierarchy. Another is that RMA Windows and File Handles have been added to the / SyncObject resource hierarchy. The last changes are that the communicators have been moved from the /SyncObject/Message hierarchy to their own Comm resource hierarchy and are displayed with their user-defined names.

A problem that requires more work to solve is how to represent MPI-I/O files in the Paradyn user interface. An obvious solution is to use the filename to represent the file in the /SyncObject hierarch of the Where Axis. However, the MPI-2 Standard explicitly states that an MPI-I/O filename is implementation independent and isn't necessarily just a filename at all. It could be a string with a hostname, a filesystem specification, a filename, and a username and password. While it will probably suffice in most cases to use the filename in the user interface for the file object, it may be a problem in the future. This may be more of an issue in heterogeneous systems.



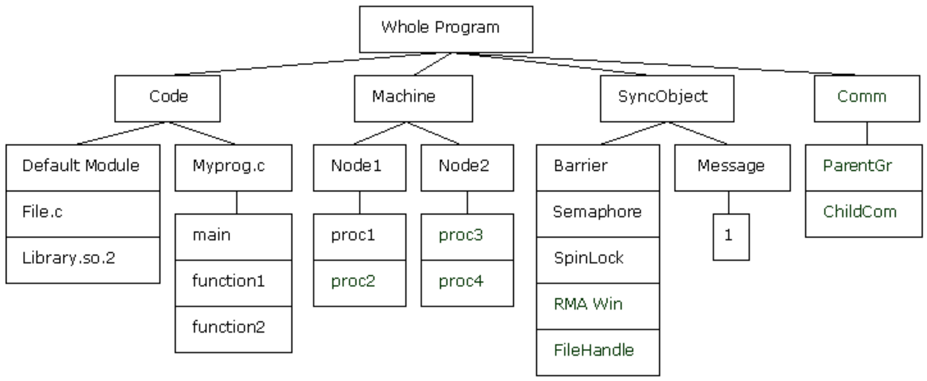


Figure 19: Changes to Paradyn Resource Hierarchy for MPI-2

The top figure shows the Resource Hierarchy for an MPI-2 application that would be generated using the current version of Paradyn. The program starts out with one process that then generates three child processes. The current version of Paradyn only detects the parent process and none of the spawned child processes. The bottom figure shows a Resource Hierarchy that incorporates our proposed changes for the same application as in the top figure. Here we highlight three key differences: three additional processes resulting from `MPI_Comm_spawn` now appear in the Machine hierarchy; RMA Windows and FileHandles are part of the SyncObject hierarchy; and Communicators are part of a new resource hierarchy instead of the /SyncObject/Message hierarchy. The separate Communicator hierarchy gives the programmer more flexibility in choosing a metric-focus pair. This allows metrics to be collected for individual Communicators, which helps the programmer easily select a group of processes that may have a particular performance issue.

7 Results and Discussion

This chapter outlines the testing phase of this project. We show that the enhanced Paradyn correctly measures the performance of all but one of the programs we tested. Section 7.1 discusses the rationale for our test plan. Section 7.2 describes the results of tests on the Grindstone Test Suite for Parallel Tools. In Section 7.3, we give the test results for a toy MPI program, ssTwod. Section 7.4 concludes this chapter by summarizing our findings.

7.1 Discussion of Testing Design

When it came time to determine whether Paradyn was correctly instrumenting and gathering the performance metrics for the MPI-1 features of LAM/MPI, we faced a challenge. There is no common file format for performance tool output to allow for direct comparison between them. Also, performance tools don't always record the same metrics or even the same types of metrics. For instance, most of the popular performance tools for message-passing programming are tracing tools, which record events in a program in a sequential manner, so that later, the order and timing of events can be reconstructed. Paradyn is for the most part a profiling tool. A profiling tool is one that records the amounts of time that a program spends in certain states. The data from these two types of tools is not directly comparable in a traditional sense. However, we did find that by looking at the overall trends produced by the tools we chose, that there was a basis for comparison.

We decided to use the MPE profiling libraries along with the Jumpshot-3 log file viewer from MPICH, which we were able to get to work with both MPICH and LAM [ZLGW99]. We also used the gprof profiling tool from GNU.

There are several different log file formats that the MPE libraries can produce. The most recent and efficient of these is SLOG. SLOG files are much smaller and require less overhead for the performance tool than its predecessor format, CLOG. The trouble we had with this tool is that we could not view the SLOG files that were generated from the MPE libraries. When we ran `logviewer` or `slog_print`, we very often received segmentation faults. We found that if we generated CLOG files and then converted them to SLOG after the run was finished, with the `clog2slog` command, that we got results more consistently. The drawback to this method was that the CLOG files that were generated were quite large, and we often had to shorten the run times of the programs to stay within the 2 GB file size limit on Linux. Also, sometimes the conversion from CLOG to SLOG file formats was not successful (i.e. segmentation fault and core dump) and we were not able to get the comparison from this other tool.

Another difficulty with the testing phase of this project is that no comprehensive test suite for automated performance tools exists. This lack of a test bed for automated performance tools has been noted by the APART Group and they are currently developing such a suite [MT02]. We did find the Grindstone Test Suite for Parallel Performance Tools [HS96]. This test suite was written for PVM, but we adapted it for use with MPI. The developers of Grindstone considered it to be a starting point for a more comprehensive set of tests for parallel performance tools. We found the Grindstone programs to be quite helpful in determining the correctness of Paradyn's performance analysis of MPI programs.

Our test plan for demonstrating that Paradyn correctly measures the performance of the MPI-1 features of LAM/MPI programs has three parts:

- Compare Paradyn's results against expected values for programs with known behavior
- Compare Paradyn's results against those of another tool: gprof or MPE
- Compare Paradyn's analysis of LAM/MPI programs against Paradyn's analysis of the same programs run under MPICH

For the tests we used LAM/MPI 7.0 with the sysv RPI and MPICH 1.2.5 with the ch_p4mpd device.

7.2 Grindstone Test Suite Discussion and Results

The programs in the Grindstone Test Suite can be broken up into two categories: communication bottlenecks and computational bottlenecks. Table 14 lists the programs that have communication bottlenecks and describes their characteristics. Table 15 shows the computational bottleneck programs and gives their behavioral descriptions.

Table 14: The Grindstone Communication Bottleneck Program Characteristics

Test Program	Characteristics
Communication Bottlenecks	
Small-messages	This program sends many small messages between several processes. The process with rank 0 acts as the server and the other processes act as clients. The clients send many small messages to the server. The server does not reply.
Big-message	This program sends very large messages between two processes. The bottleneck is the overhead associated with setting up and sending a very large message. The communication bandwidth limits the speed at which the messages are passed.
Wrong-way	This program simulates the problem where one process expects to receive messages in a certain order, but another process sends them in a different order than is expected.
Intensive-server	This program simulates an overloaded server. Again, the process with rank 0 acts as the server and the other processes are the clients. Each of the clients repeatedly sends a message to the server and then waits for a reply. Upon receiving a message, the server wastes time before replying, simulating a server that is overloaded with client requests.
Random-barrier	This program is like the intensive-server program except that no single process is the bottleneck. On each iteration through a loop a random process is chosen to waste time while the other processes wait in `MPI_Barrier`.

Table 15: The Grindstone Computational Bottleneck Program Characteristics

Computational Bottlenecks	
Diffuse-procedure	This program demonstrates a bottleneck that is distributed over the processes in the MPI application. The `bottleneckProcedure` consumes ~50% of the time for the application. Each of the processes in the application take turns "being the bottleneck" while the others execute `irrelevantProcedure`s and then wait in `MPI_Barrier`.
System-time	This program spends most of its time executing in system calls.
Hot-procedure	This program has a bottleneck in a single procedure, called `bottleneckProcedure` that uses most of the program's time. There are also several `irrelevantProcedure`s that use hardly any of the program's time.

We give a short synopsis of the test results on the Grindstone Test Suite in Table 16. Each program is listed along with a 'pass' or 'fail' and a summary of our findings during testing. More detailed test results are given in the subsections to follow and are labelled by program name. Note, that in most cases, it was necessary to increase the number of iterations of the program to allow adequate time for Paradyn to complete its diagnosis. In each section detailing the test results for a particular program, the parameters used for running the tests, such as number of iterations and message size, are given. The table clearly shows that Paradyn is able to find the synchronization and computation bottlenecks in LAM/MPI programs. The exception is the program system-time. Paradyn does not have metrics for measuring system time and thus did not find bottlenecks in the program. Discussion of this is outside of the scope of this work.

Table 16: Grindstone Test Suite Results

Test Program	Paradyn's Result	Details
Small-messages	Pass	Paradyn identified the bottleneck as being too much message passing and showed the clients spending too much time in `MPI_Send`.
Big-message	Pass	Paradyn showed that the bulk of the program's time was spent sending and receiving messages. It also was able to give a good count of the number of bytes sent and received.
Wrong-way	Pass	Paradyn identified that the program was spending too much time in send and receive operations.
Random-barrier	Pass	Paradyn found that the program was spending much time in `MPI_Barrier` because processes were late getting to the barrier. It also showed that the program had a computational bottleneck.
Intensive-server	Pass	Paradyn identified that the clients were waiting in `MPI_Send` because the server was too busy to do the matching receive on time.
Diffuse-procedure	Pass	Paradyn correctly showed that the program was spending too much time in `MPI_Barrier`. Paradyn did not find a computational bottleneck. We justify this in Section 7.2.7.
System-time	Fail	Paradyn showed all top level hypotheses as false. Paradyn does not have metrics specifically for system time.
Hot-procedure	Pass	Paradyn correctly found that the each process was CPUBound in the function `bottleneckProcedure`.

7.2.1 Explanation of Diagrams and Symbols

We created diagrams that allow us to compare the Performance Consultant's output for LAM/MPI and MPICH relatively easily. We condensed the information given by the Performance Consultant to achieve this. In the diagrams, only hypotheses that tested true for either MPICH or LAM/MPI are shown. The staggered ovals represent the findings for MPICH and LAM/MPI, with MPICH in the foreground. If only one oval is present for a hypothesis, this indicates that the result was identical for both MPICH and LAM/MPI. A † in a diagram of this chapter symbolizes that a mapping has been made between what is shown in the Performance Consultant window and what is shown in the diagram. A common mapping is for MPI function names for LAM/MPI and MPICH. For instance, the Performance Consultant window shows `MPI_Send` for LAM/MPI and `PMPI_Send` for MPICH. We use the `MPI_` prefix in the diagrams when a mapping is necessary. In Section 5.2, we discuss the reason for the Performance Consultant showing a `PMPI_` prefix for function names with MPICH instead of the `MPI_` prefix as it does for LAM/MPI. Another mapping takes machine names shown in the Performance Consultant window and renames them "NodeX", where X is an integer representing the order the machine is listed in the machinefile used by the MPI implementation. We also mapped process identifiers to "Process{X}", where here X represents the process's MPI rank. The last mappings are for MPI communicators and MPI message tags. They are named "CommunicatorX" and "MsgTagX." The X in the communicator name gives a mapping between a communicator in MPICH and in LAM/MPI, which have different representations for the same source-code object. The message tag mapping is primarily used to identify the node as an MPI message tag.

7.2.2 Small-Messages

For the program small-messages, the following parameters were used: 10,000,000 iterations, 4 byte message size, 6 processes, 2 each on three nodes. The program run under LAM/MPI took approximately 515 seconds. Figure 20 shows the condensed form of the output from the Performance Consultant for LAM/MPI and MPICH. We see that the Performance Consultant found that for both LAM/MPI and MPICH ExcessiveSyncWaitingTime is true, and drilled down into the function `Gsend_message`, and even further to find `MPI_Send`. This is what we would expect to see for this program given that the clients all send messages to the server process. For LAM/MPI, the Performance Consultant was able to discover the communicator and message tag on which the communication was taking place. For MPICH, the program is found to have ExcessiveIOBlockingTime. This may be because the MPICH ch_p4mpd device does not currently have SMP support. Instead of using shared memory operations to optimize communication between processes on the same machine, it uses traditional socket communication. The `send` and `recv` functions are included in Paradyn's I/O metric definitions, so ExcessiveIOBlockingTime tests true.

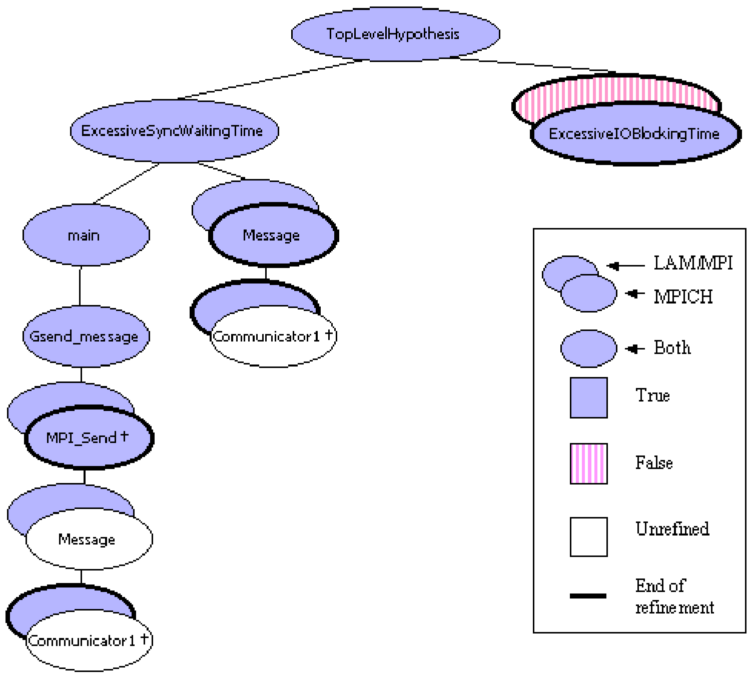
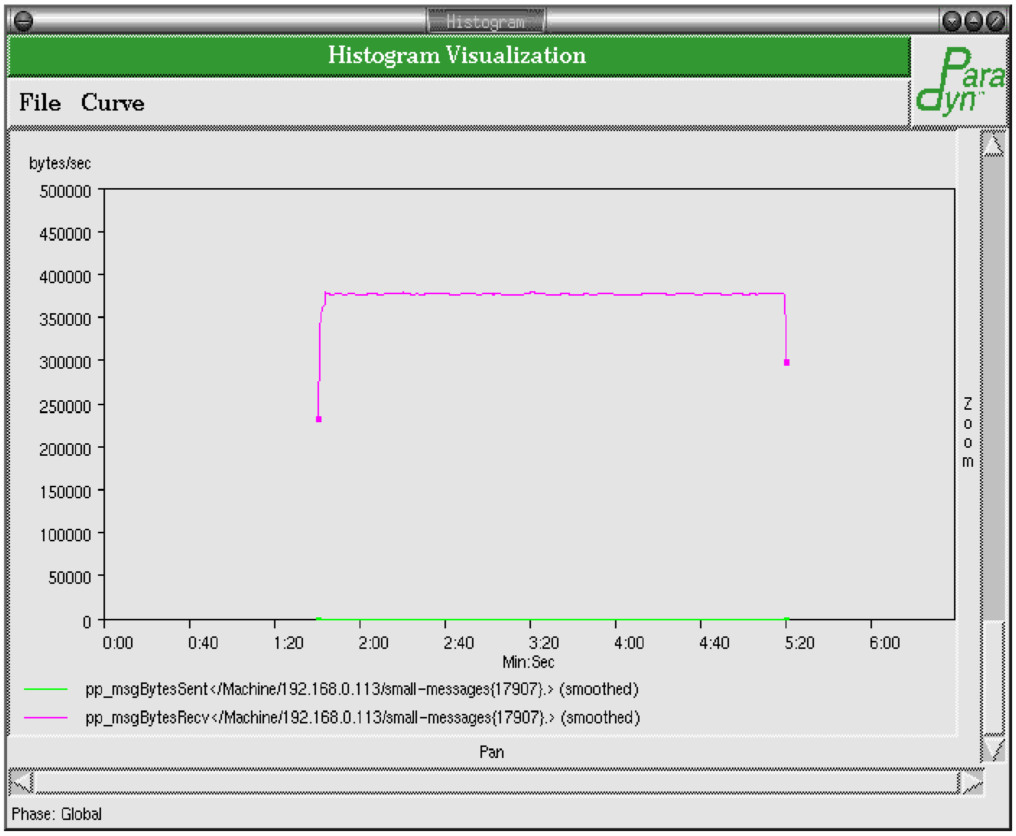


Figure 20: Paradyn PC Output for Small-Messages
This figure shows a condensed form of the Performance Consultant output for small-messages. It compares the output for LAM/MPI and MPICH. In it we see Paradyn determined that ExcessiveSyncWaitingTime is true for both MPI implementations and drilled down through the function `Gsend_message` to `MPI_Send`. For LAM/MPI, it also identified the communicator that the processes are using for the message-passing. For MPICH, the Performance Consultant found that ExcessiveIOBlockingTime is true. This is a result of the inner workings of MPICH's communication routines, which make heavy use of `read` and `write` system commands to pass messages.

To further ensure that Paradyn was correctly working with this program, we counted message bytes that were passed between the processes. By inspecting the program itself and its per process output, we know that each client process sent 10,000,000 messages at 4 bytes each, for a total of 40,000,000 bytes, and that the server process received 50,000,000 messages, for a total of 200,000,000 bytes.

Figure 21 is a screenshot of the histogram that Paradyn generated showing the byte counts for the server process and one client process for LAM/MPI. We exported the data that Paradyn gathered while making the histogram and calculated the number of bytes that were sent and received throughout the course of the program. Our calculations on the data showed that the average bytes/second of messages received by the



pp_msgBytesSent</Machine/192.168.0.113/small-messages{17907}.> (smoothed)
pp_msgBytesRecv</Machine/192.168.0.113/small-messages{17907}.> (smoothed)

Figure 21: Paradyn Histogram Small-Message with LAM/MPI, Server Process Message Bytes Sent and Received
This is a histogram from Paradyn showing the message bytes sent and received for the server process. We see that the server did not send any messages, but received many. The average bytes/second of messages received by the server was 386,910.809. Multiplying this by the number of seconds the program ran gives 386,910.809 * 515 = 199,259,066 total bytes received at the server. Note: The colors in this screenshot were altered for printing purposes.

server was 386,910.809 and that the average number of bytes/second sent by the client was 77,526.34. Multiplying these by the number of seconds the program ran, gives 199,259,066 total bytes received at the server, and 39,925,890 total bytes sent by one client. These numbers are slightly lower than the known values, but this is to be expected. Paradyn does not insert instrumentation into a program until runtime. It is understandable that some messages may have been passed before Paradyn was able to insert the instrumentation.

7.2.3 Big-Message

The next set of tests were done with the program big-message. The parameters we used for this program were: 1000 iterations, 100,000 byte message size, and 2 processes, one per node. The results we gathered for this program were consistent with the program's behavioral description. In Figure 22, we show the condensed Paradyn Consultant output for big-message with LAM/MPI and MPICH. The Performance Consultant had identical findings for both MPI implementations. We see that ExcessiveSyncWaitingTime is true and that the Performance Consultant drilled through both `Gsend_message` and `Grecv_message` to the MPI functions `MPI_Send` and `MPI_Recv`. It also determined the communicator on which the excessive communication was taking place.

In addition, we measured the message byte count for big-message. By inspecting the program source, we know that each process sent and received 1000 messages. They received 400,000,000 bytes total and sent 400,000,000 bytes total. From the per process output we know that the program ran for approximately 68.6 seconds. In Figure 23, we show the histogram from Paradyn of point-to-point message bytes sent and received for one of the processes. We exported the data that Paradyn collected to create the histogram. Then, we calculated the average bytes sent per second to be 5,800,820.4

and the average bytes received per second to be 5,800,482.79 for that process.

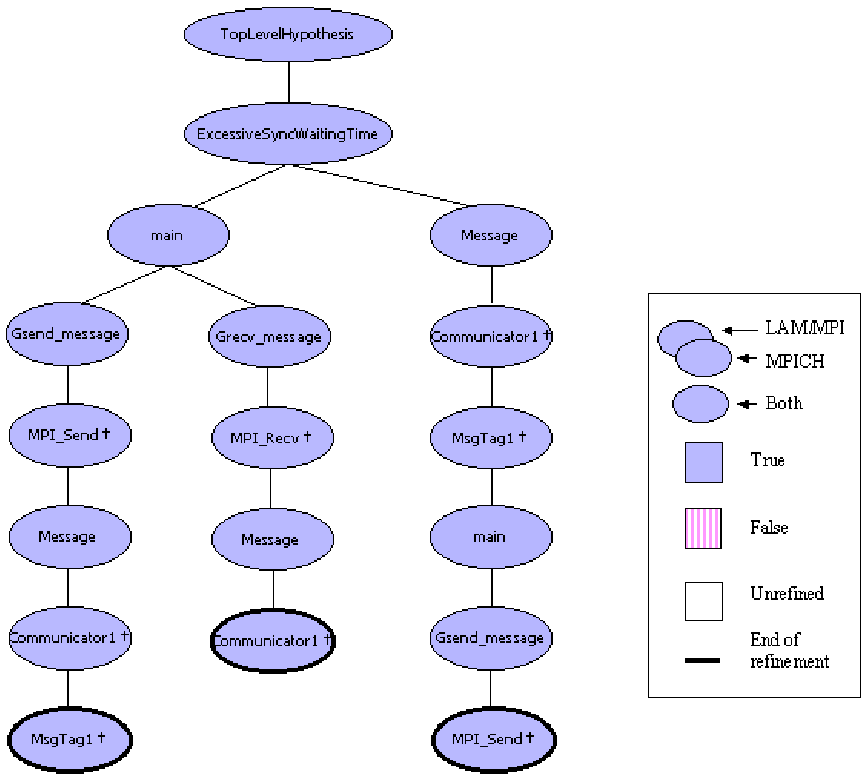


Figure 22: Paradyn PC Output for Big-Messages
Here were show the condensed Performance Consultant output for big-message with LAM/MPI and MPICH. We see that ExcessiveSyncWaitingTime is true for both implementations and that the PC has drilled down through the functions `Gsend_message` and `Grecv_message` to `MPI_Send` and `MPI_Recv`. It also found the communicator associated with the communication bottleneck.

Multiplying these by the number of seconds the program ran, gives 397,936,279.44 total bytes sent and 397,913,119.394 total bytes received. These figures are slightly lower than the 400,000,000 reported by the processes. This is because of the delay before Paradyn inserts the instrumentation to count the bytes at runtime.

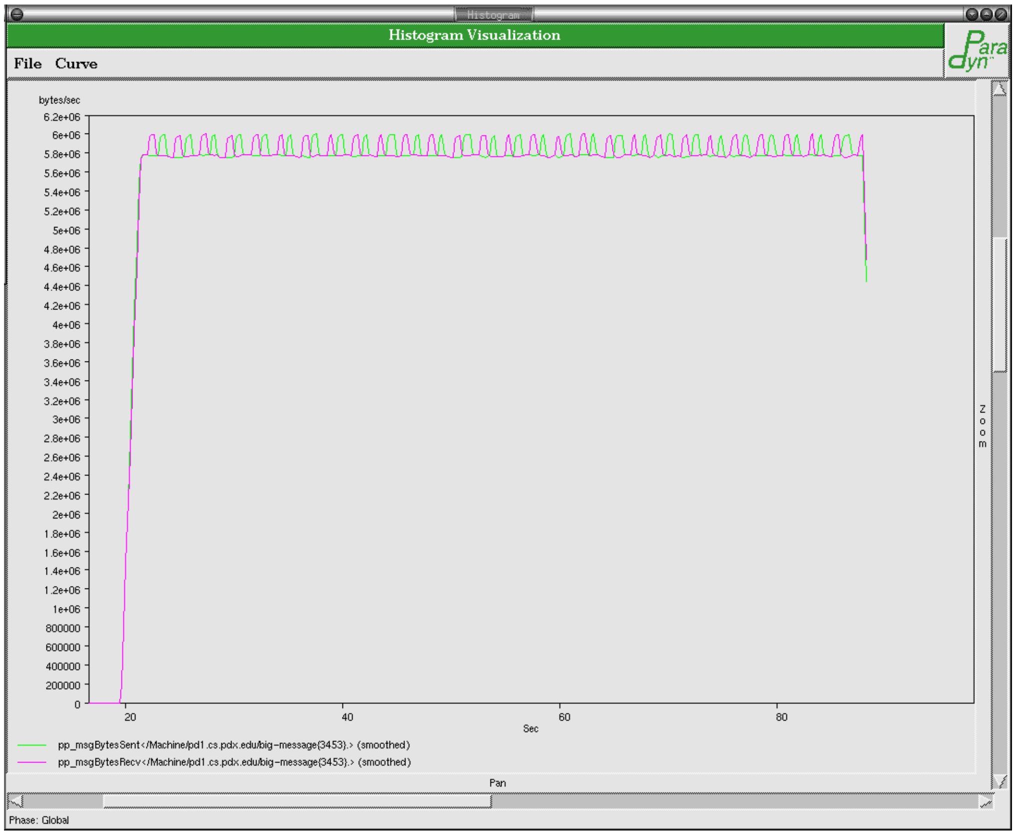


Figure 23: Paradyn Histogram Big-Message with LAM/MPI, Message Bytes Sent and Received

This figure shows the histogram that Paradyn generated for point-to-point message bytes sent and received for one process with LAM/MPI. We calculated the average bytes sent per second to be 5,800,820.4 and the average bytes received per second to be 5,800,482.79. Multiplying these by the number of seconds the program ran, 68.6, gives 397,936,279.44 total bytes sent and 397,913,119.394 total bytes received. Note: The colors in this screenshot were altered for printing purposes.

7.2.4 Wrong-Way

The next program we used for testing was wrong-way. The parameters we used were: 18,000 iterations and 1000 messages. Again, we see that Paradyn was able to find the bottlenecks. In Figure 24, we show the condensed Performance Consultant output for wrong-way. We see that ExcessiveSyncWaitingTime is true and that `Gsend_message` and `Grecv_message` are the bottlenecks for both LAM/MPI and

MPICH. Also, for both MPI implementations, the Performance Consultant finds message-passing to be consuming excessive synchronization time. For MPICH, the Performance Consultant drilled down through `Gsend_message` and `Grecv_message` to find `PMPI_Send` and `PMPI_Recv` as synchronization bottlenecks.

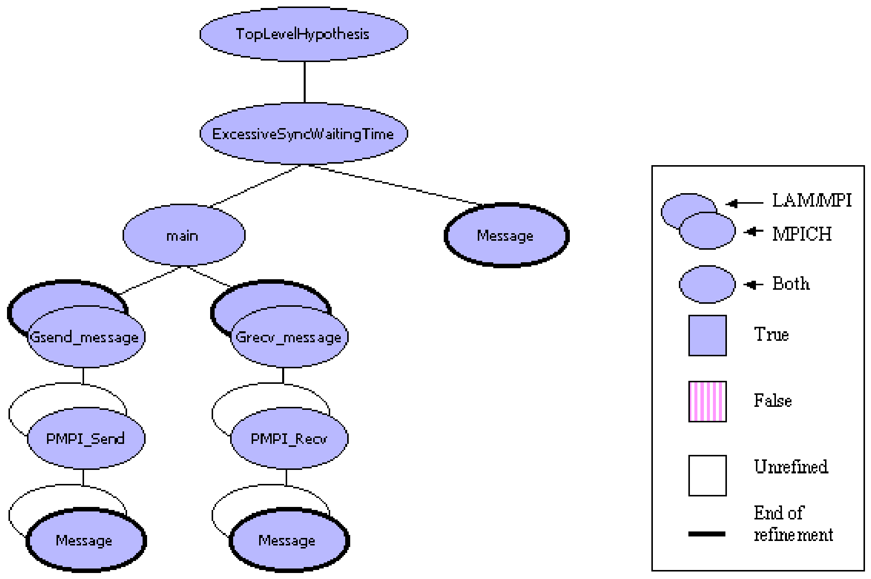


Figure 24: Paradyn PC Output for Wrong-Way
Here we see the Performance Consultant has discovered that ExcessiveSyncWaitingTime is true and that the functions `send_message` and `recv_message` are the bottlenecks for both MPICH and LAM/MPI. It further drilled down to find `MPI_Send` and `MPI_Recv`.

We also used Paradyn to measure the number of bytes that were sent between the processes for wrong-way with LAM/MPI. We see from looking at the process output and from inspecting the program source that 18,000,000 messages were sent and 18,000,000 messages received. Since 4 bytes were sent in each message, this gives a total of 72,000,000 bytes sent and received. The process output also shows that the wall clock time for the program run was approximately 74.6 seconds. Figure 25 shows the histogram that Paradyn generated to display the bytes sent and received for LAM/

MPI. We exported the data that Paradyn collected and calculated that process 0 sent an average of 956,779.2 bytes per second, and that process 1 received 944,582.7 bytes per second. Multiplying these by the number of seconds that the program ran gives

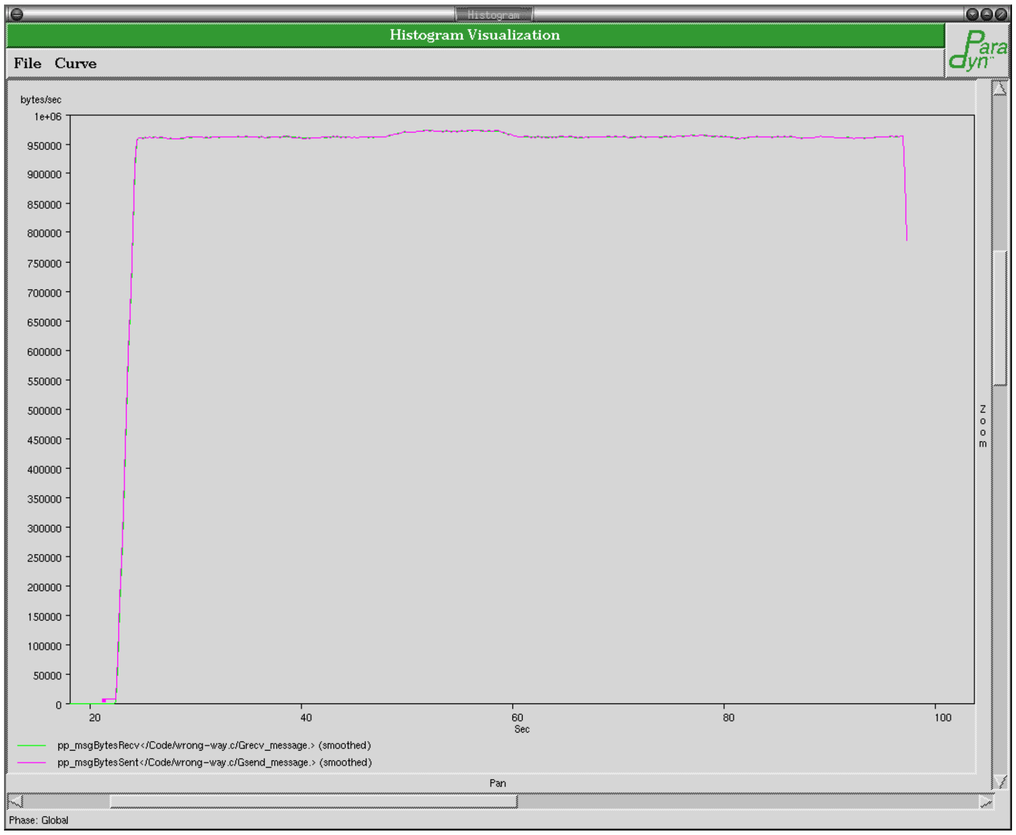


Figure 25: Paradyn Histogram Wrong-Way with LAM/MPI, Message Bytes Sent and Received
This is a histogram showing the bytes sent by process 0 and the bytes received by process 1. We performed calculations on the data that Paradyn generated and found that process 0 averaged sending 956,779.2 bytes per second and that process 1 received 944,582.7 bytes per second. Multiplying these by the number of seconds that the program ran, 74.6, gives 71,375,728 bytes sent and 70,465,869 bytes received. Note: The colors in this screenshot were altered for printing purposes.

71,375,728 bytes sent and 70,465,869 bytes received. Again, these numbers are slightly lower than the actual values. This is attributed to the delay before Paradyn was able to insert the instrumentation to gather these measurements at runtime.

7.2.5 Random-Barrier

The next program we used to verify Paradyn's measurements was random-barrier. The parameters we used for the program runs were: 800 iterations, TIMETOWASTE = 5, and six processes, two each on three nodes. Paradyn was able to correctly identify the bottlenecks for this program. Figure 26 shows the condensed Performance Consultant's analysis of the program with LAM/MPI and MPICH. The Performance Consultant found that ExcessiveSyncWaitingTime is true and drilled down to find

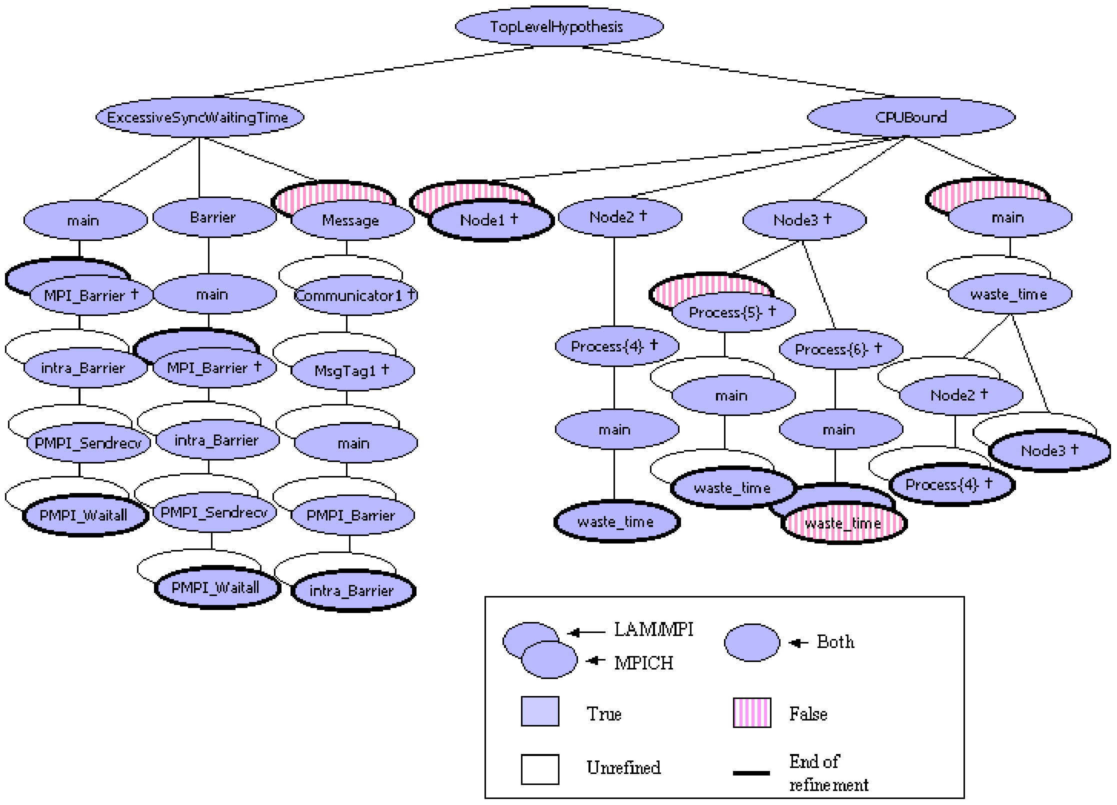


Figure 26: Paradyn PC Output for Random-Barrier
This is the condensed Performance Consultant output for random-barrier. It shows that Paradyn finds too much time is being spent in `MPI_Barrier`, and that the program is CPU-bound. This agrees with the program's behavioral description. The Performance Consultant is able to pinpoint the function `waste_time` as the computation bottleneck.

`MPI_Barrier` as the bottleneck. For MPICH, it drilled down to expose some of the inner workings of the MPICH implementation, showing that `PMPI_Barrier` is implemented as a collective communication operation with `PMPI_Sendrecv`. Also for MPICH, the Performance Consultant was able to identify the communicator and message tag on which the excessive message passing was taking place. For both implementations, the program was found to be CPU bound, and discovered to be so in the function `waste_time`. Due to the random nature of this program, not every process was found to be CPU bound in `waste_time`. This is because that a particular process may not be designated by the program to be the "time waster" at the point when the Performance Consultant was measuring the CPU usage of that process.

We also used Paradyn to generate histograms of the synchronization time spent in these programs. Figure 27 is from runs with LAM/MPI and MPICH. The figure in the back left is for MPICH, while the one for LAM/MPI is in the foreground. Each show that the time spent in synchronization is approximately equal across all the processes in the MPI program. We exported the data that Paradyn collected and found that for the LAM/MPI run, the average inclusive synchronization wait time was 61%, while the same measurement for the MPICH run was 62%.

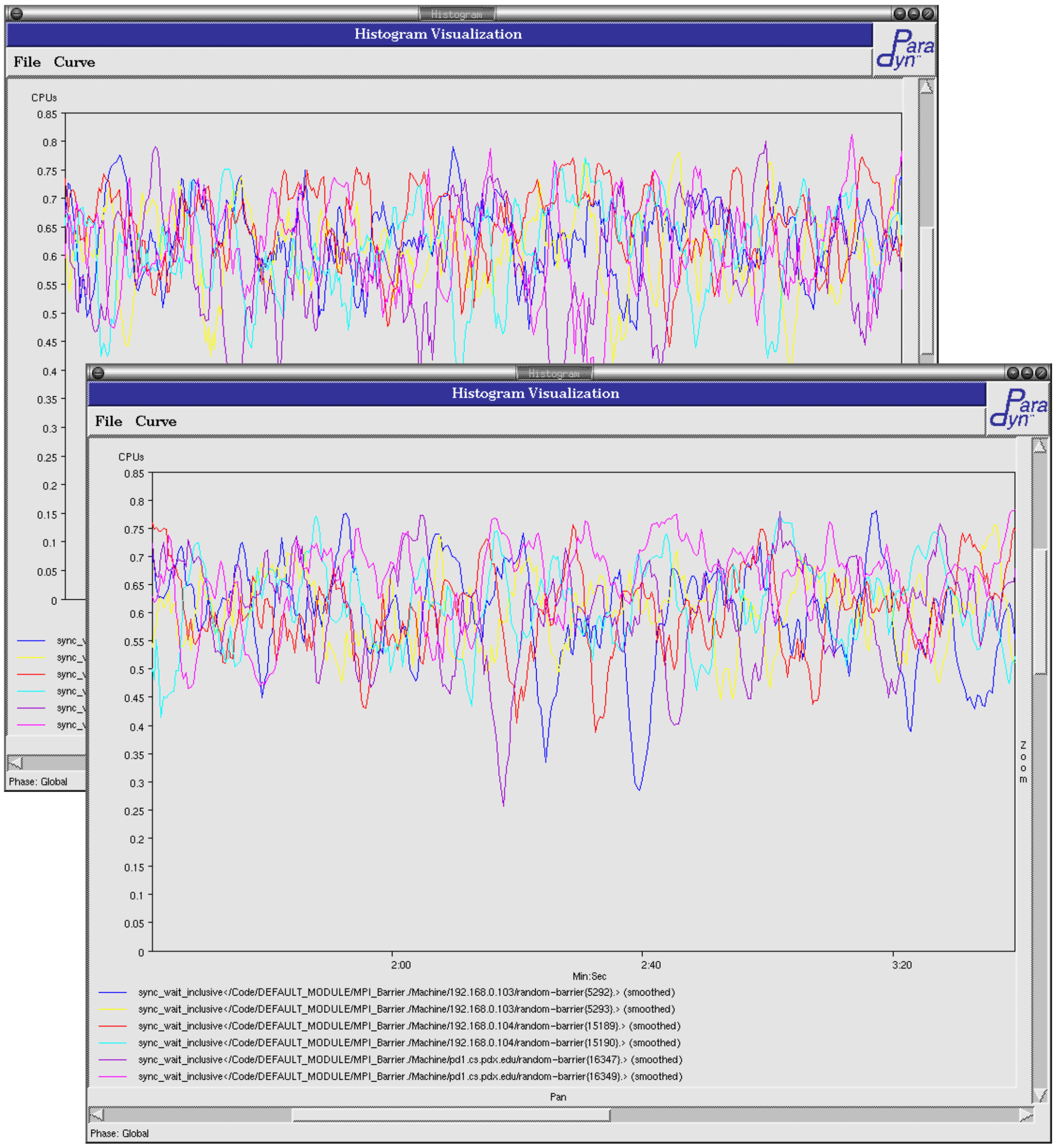

Histogram Visualization
File Curve
CPUs
Min:Sec
2:00
2:40
3:20
sync_wait_inclusive</Code/DEFAULT_MODULE/MPI_Barrier./Machine/192.168.0.103/random-barrier{5292}.> (smoothed)
sync_wait_inclusive</Code/DEFAULT_MODULE/MPI_Barrier./Machine/192.168.0.103/random-barrier{5293}.> (smoothed)
sync_wait_inclusive</Code/DEFAULT_MODULE/MPI_Barrier./Machine/192.168.0.104/random-barrier{15189}.> (smoothed)
sync_wait_inclusive</Code/DEFAULT_MODULE/MPI_Barrier./Machine/192.168.0.104/random-barrier{15190}.> (smoothed)
sync_wait_inclusive</Code/DEFAULT_MODULE/MPI_Barrier./Machine/pd1.cs.pdx.edu/random-barrier{16347}.> (smoothed)
sync_wait_inclusive</Code/DEFAULT_MODULE/MPI_Barrier./Machine/pd1.cs.pdx.edu/random-barrier{16349}.> (smoothed)
Zoom
Pan
Phase: Global

Figure 27: Paradyn Histograms Random-Barrier, Inclusive Synchronization Time

These are histograms generated by Paradyn showing the sync_wait_inclusive metric for all six processes in the programs for LAM/MPI and MPICH. The histogram for MPICH is in the back left, while the one for LAM/MPI is in the foreground. They both show that the programs are spending a significant portion of time in synchronization operations and that the time is spread out over all processes in the programs. The average sync_wait_inclusive time over all processes for LAM/MPI is 61%, and 62% for MPICH.

We ran another test to verify the amount of synchronization time that was spent in this program. We used the MPE libraries to generate a log of the events that occurred in the program. Figure 28 shows the Statistical Preview window from Jumpshot-3. Because of file size limitations, we had to shorten the run time of the program to be able to produce a usable log file. For this run we used 80 iterations, TIMETOWASTE = 5, and four processes, two each on two nodes. The figure shows that of the four processes in the MPI program approximately three of them were executing in `MPI_Barrier` at any given time. This agrees with Paradyn's findings and with the program's behavioral description.

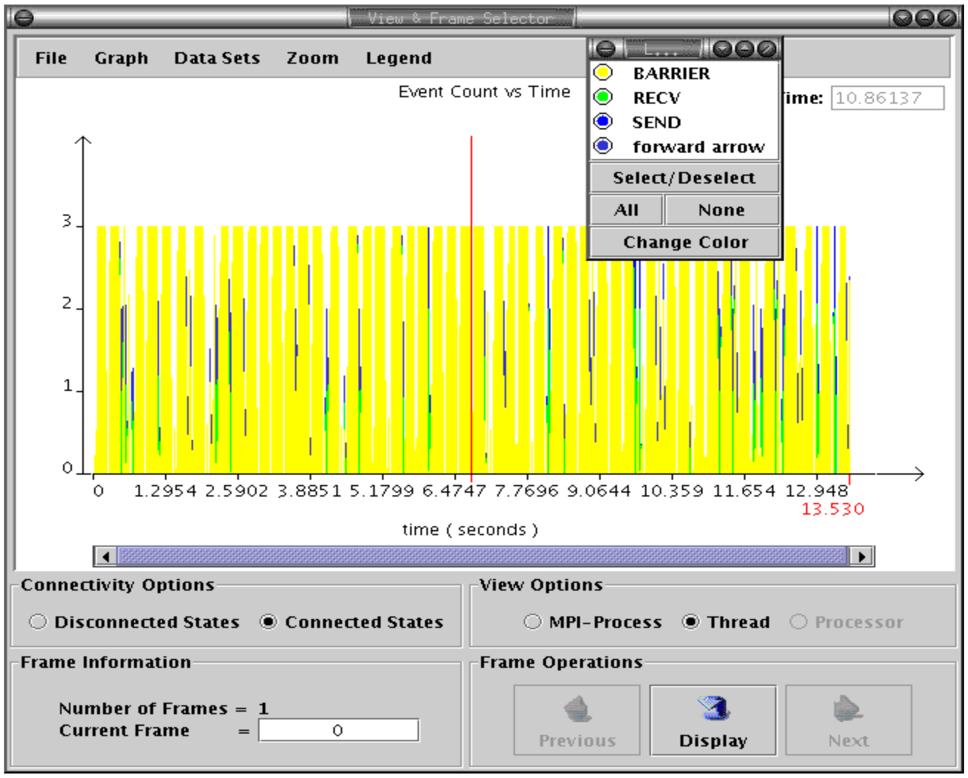


Figure 28: Jumpshot-3 Statistical Preview for Random-Barrier with LAM/MPI

This is a screen shot of the statistical preview window in Jumpshot-3 for random-barrier when compiled with the MPE libraries. This figure shows that of the four processes in the MPI program approximately three of them were executing in `MPI_Barrier` at any given time.

7.2.6 Intensive-Server

The next program we used for testing was intensive-server. The parameters we used for the runs were: 10,000 iterations, TIMETOWASTE = 1, and 6 processes, two each on three nodes. Paradyn was able to find the bottleneck in this program. Figure 29 shows the condensed form of the Performance Consultant's findings for LAM/MPI and MPICH. We see that ExcessiveSyncWaitingTime is true and that the Perfor-

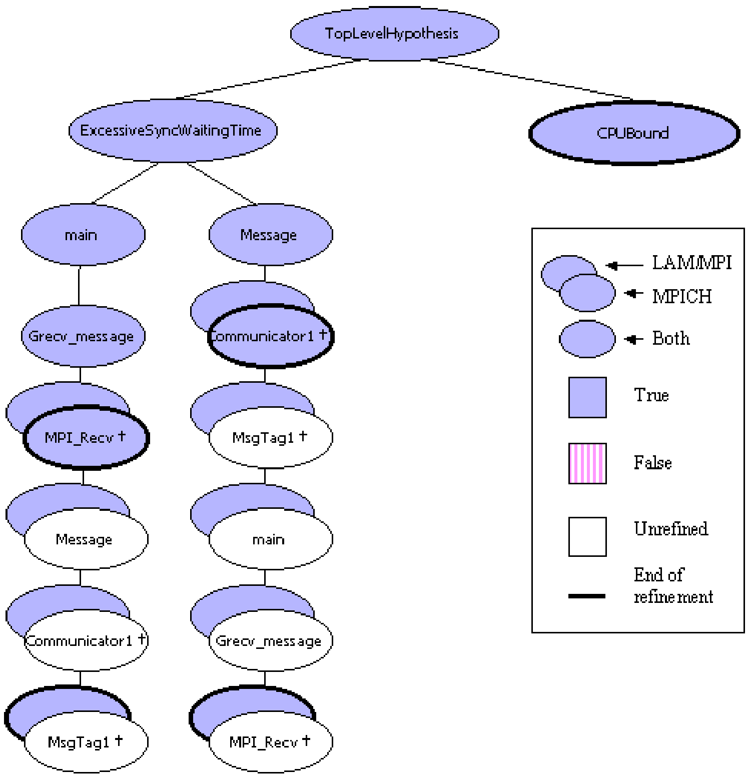


Figure 29: Paradyn PC Output for Intensive-Server
This shows the condensed form of the Performance Consultant's output for the intensive-server program run with LAM/MPI and MPICH. For both implementations, we see that the hypothesis ExcessiveSyncWaitingTime is true and that the PC drilled down through `Grecv_message` to discover `MPI_Recv` as the bottleneck. It was also able to determine the communicator for the bottleneck. For LAM/MPI, further refinement was possible and the message tag on which the communication was taking place was found. For both, the Performance Consultant showed that CPUBound was also true, but did not refine the hypothesis further.

mance Consultant drilled down through `Grecv_message` to show `MPI_Recv` as the bottleneck. It was also able to determine the communicator upon which the excessive communication was taking place. For LAM/MPI, the Performance Consultant found the message tag on which the communication occurred. For both MPI implementations, the hypothesis CPUBound was also found to be true, although the root of the bottleneck was not discovered.

Figure 30 shows histograms generated by Paradyn when measuring inclusive synchronization waiting time for a run of intensive-server with LAM/MPI. The top left diagram shows a client process using nearly all of its time in synchronization in the function `Grecv_message,` which is represented by the red line in that diagram. It also shows that virtually none of its time is spent in synchronization in the function `Gsend_message`, shown by the blue-green line in the diagram. This is what we expect, because the intensive-server program is set up to mimic clients waiting for response from an overloaded server. The diagram in the bottom left is synchronization time for the server process. Here we see that the server process is not spending overly much time in synchronization, which is what we would predict, given the program’s behavioral description.

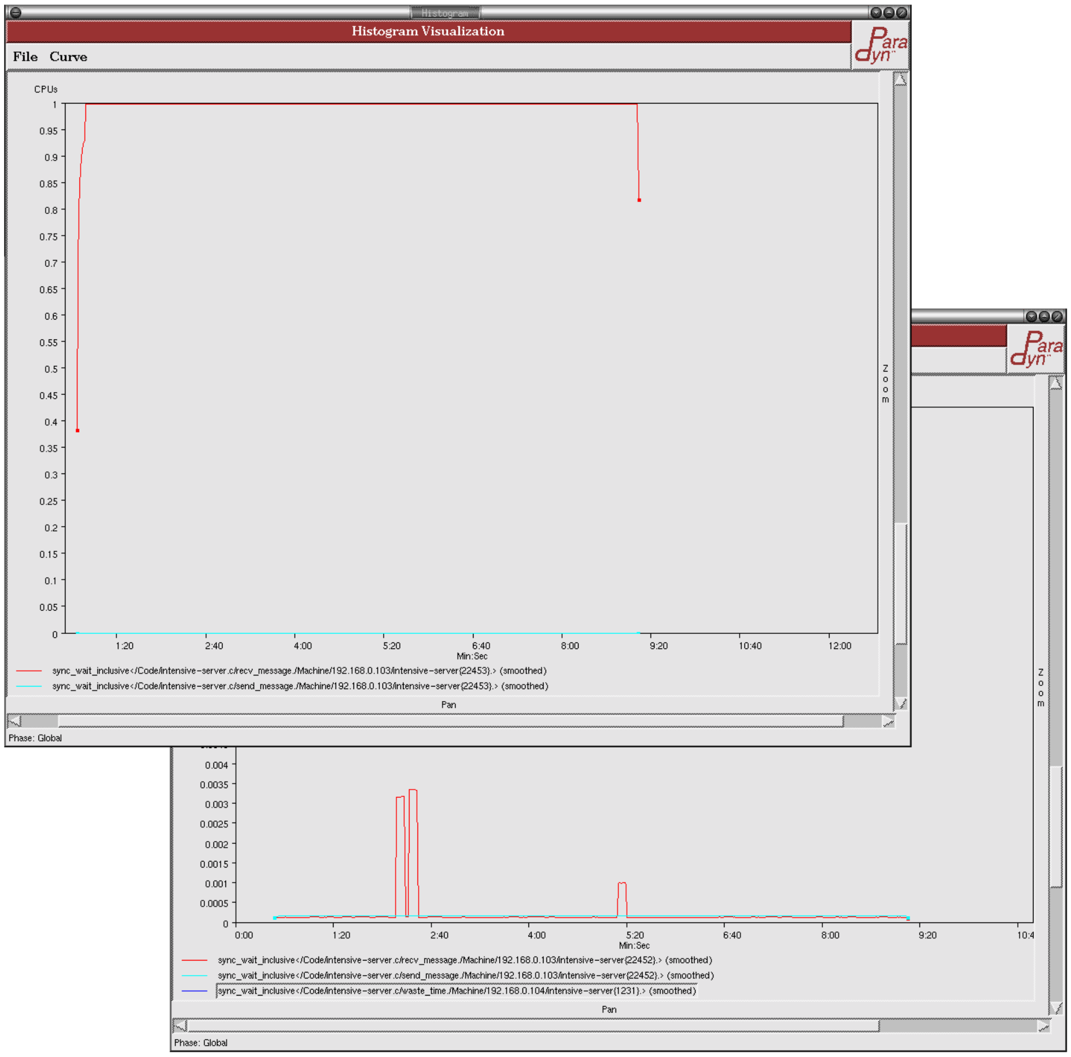


Figure 30: Paradyn Histograms Intensive-Server with LAM/MPI, Inclusive Synchronization Time for a Client Process and Server Process

These are histograms generated by Paradyn showing the inclusive synchronization waiting time for a client process and server process in the intensive-server program with LAM/MPI. The top left diagram shows that the clients are spending nearly all of their time in `Grecv_message`, represented by the red line, and hardly any time in `Gsend_message`, shown in the blue-green line. Calculations on the data collected by Paradyn tell that an average of 0.997976 of the CPU time for a client process was spent in `Grecv_message`. In contrast, on average, only 0.000027 of a client's CPU time was spent in `Gsend_message`. The diagram in the bottom right shows the synchronization time for the server process. We see that the server does not spend much time in `Gsend_message` or `Grecv_message`. The average inclusive synchronization waiting times were 0.000249 and 0.000181 for `Grecv_message` and `Gsend_message`, respectively.Note: The colors in this screenshot were altered for printing purposes.

Figures 31 and 32 further uphold Paradyn's findings. They are Jumpshot-3 output for intensive-server run with LAM/MPI and linked with the MPE libraries. We shortened these runs to avoid any log file size problems. The parameters were: 10 iterations, TIMETOWASTE = 1, and three processes, one each on three nodes. Figure 31 shows the Statistical Preview window for this program run. From it, we can see that of the three processes in the program, at any given time, approximately two of them are in `MPI_Recv`.

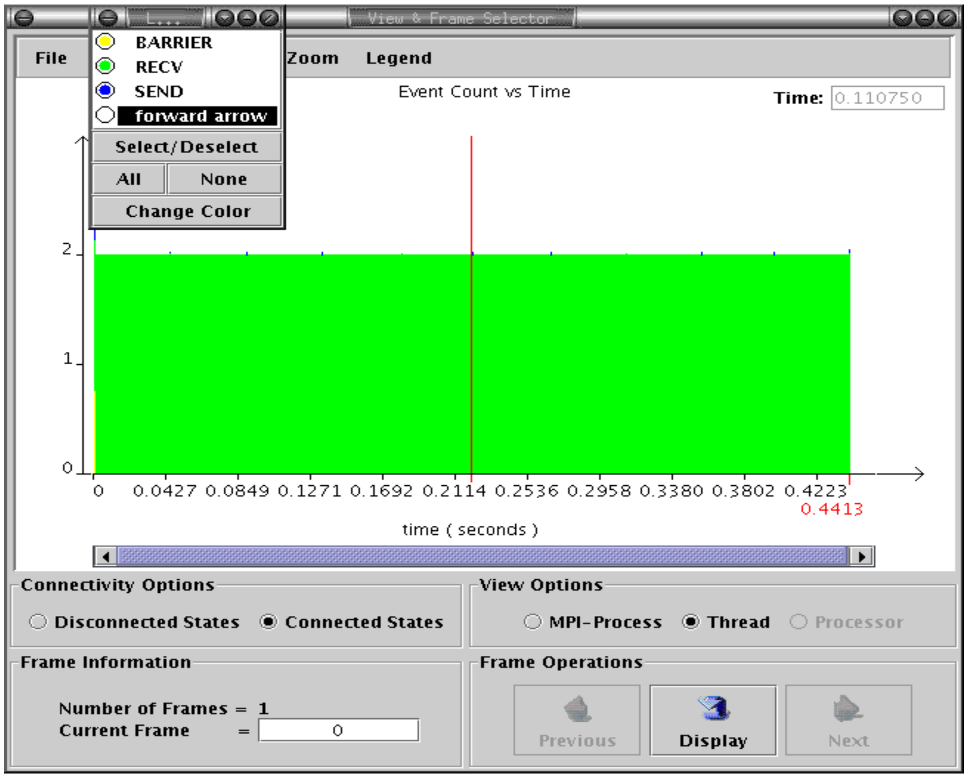
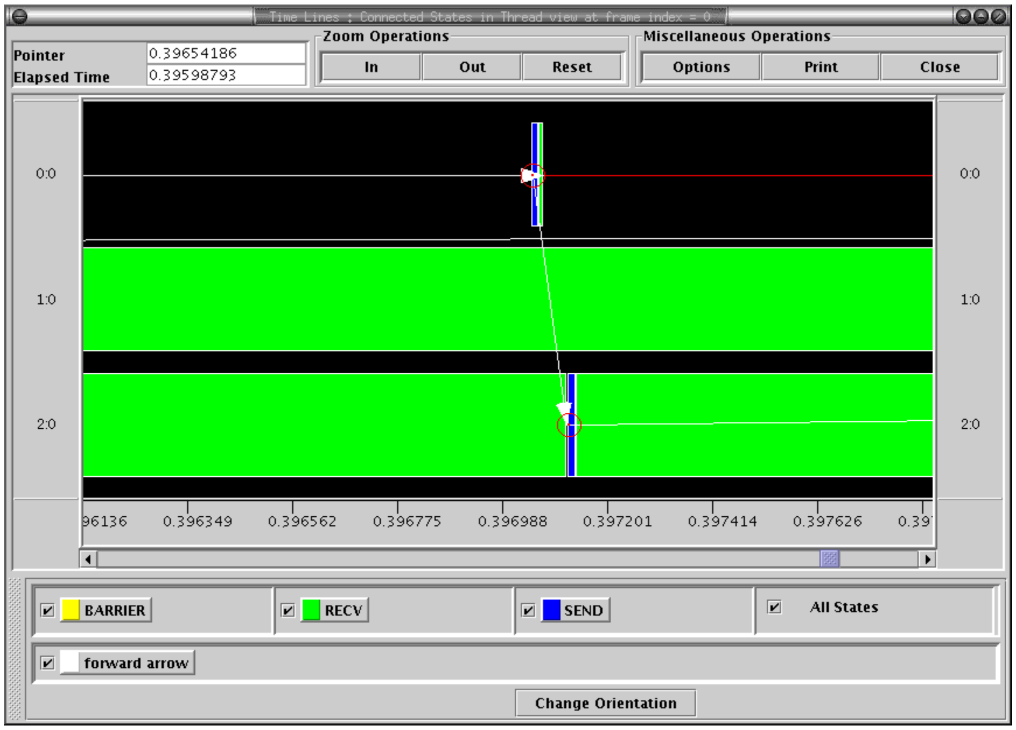


Figure 31: Jumpshot-3 Statistical Preview for Intensive-Server with LAM/MPI

This is a screenshot of the Statistical Preview window generated by Jumpshot-3 for the intensive-server program run with LAM/MPI and linked with the MPE libraries. From it, we can see that of the three processes in the MPI program, at any given time, approximately two of them were executing in `MPI_Recv`.

Figure 32 is a small portion of Jumpshot-3's Time Lines Window that illustrates that the server process, process 0, is not spending much time in communication operations, but that the clients, processes 1 and 2, are spending a large portion of their time in `MPI_Recv` and hardly any in `MPI_Send`.



Pointer 0.39654186
Elapsed Time 0.39598793

96136 0.396349 0.396562 0.396775 0.396988 0.397201 0.397414 0.397626 0.39

Figure 32: Jumpshot-3 Time Lines Window for Intensive-Server with LAM/MPI
This figure is the Time Lines Window from Jumpshot-3 for the intensive-server program run with LAM/MPI and linked with the MPE libraries. It gives further evidence of the behavior of this program. We have used the zoom feature of the program to make details of the communication in the program visible. It shows that the server process, Process 0, spends hardly any time in synchronization operations, while the client processes, processes 1 and 2, are spending most of their time in `MPI_Recv`.

7.2.7 Diffuse-Procedure

The next program studied was diffuse-procedure. The parameters we used for this run were: 2000 iterations and 4 processes, two each on two nodes. Figure 33 shows the condensed form of the Performance Consultant's analysis of the program run with LAM/MPI and MPICH. For both implementations, the Performance Consultant found the hypothesis ExcessiveSyncWaitingTime to be true and drilled down to find `MPI_Barrier` as the bottleneck. With the threshold for CPU usage set to 0.2, it found that the program was CPU bound, and found the bottleneck to be in the function `bottleneckProcedure`. For MPICH, the Performance Consultant showed that `MPI_Barrier` is implemented as a collective communication, with `PMPI_Sendrecv`.

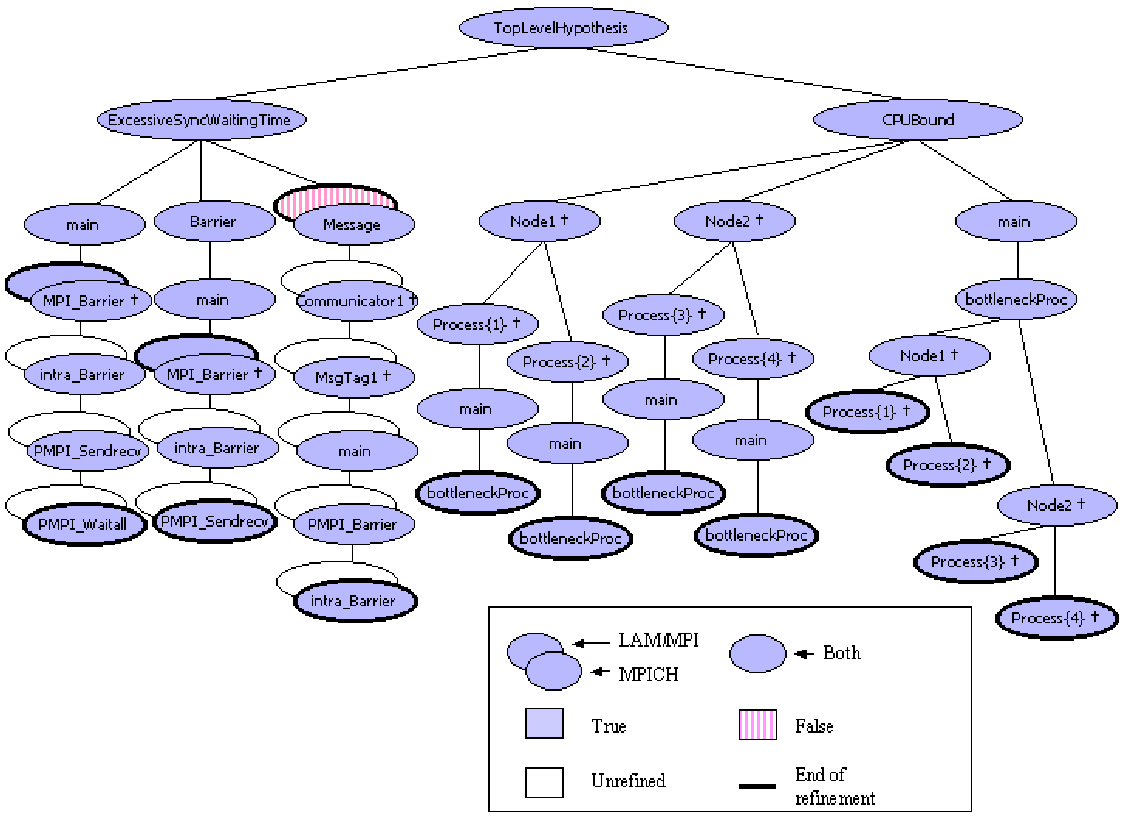


Figure 33: Paradyn PC Output for Diffuse-Procedure
This figure shows the condensed form of the Performance Consultant's findings for the diffuse-procedure program run with LAM/MPI and MPICH. For both, we see that ExcessiveSyncWaitingTime is true and that the bottleneck is `MPI_Barrier`. It also shows that the program is CPU bound in the function `bottleneckProcedure`.

We set the threshold for CPU usage to 0.2 because if we did not, the Performance Consultant did not find a computational bottleneck. Figure 34 shows a histogram of the CPU inclusive time for three procedures across the whole application. The three procedures are `bottleneckProcedure`, `irrelevantProcedure0`, and `irrelevantProcedure1`. The histogram shows that approximately 1 CPU's worth

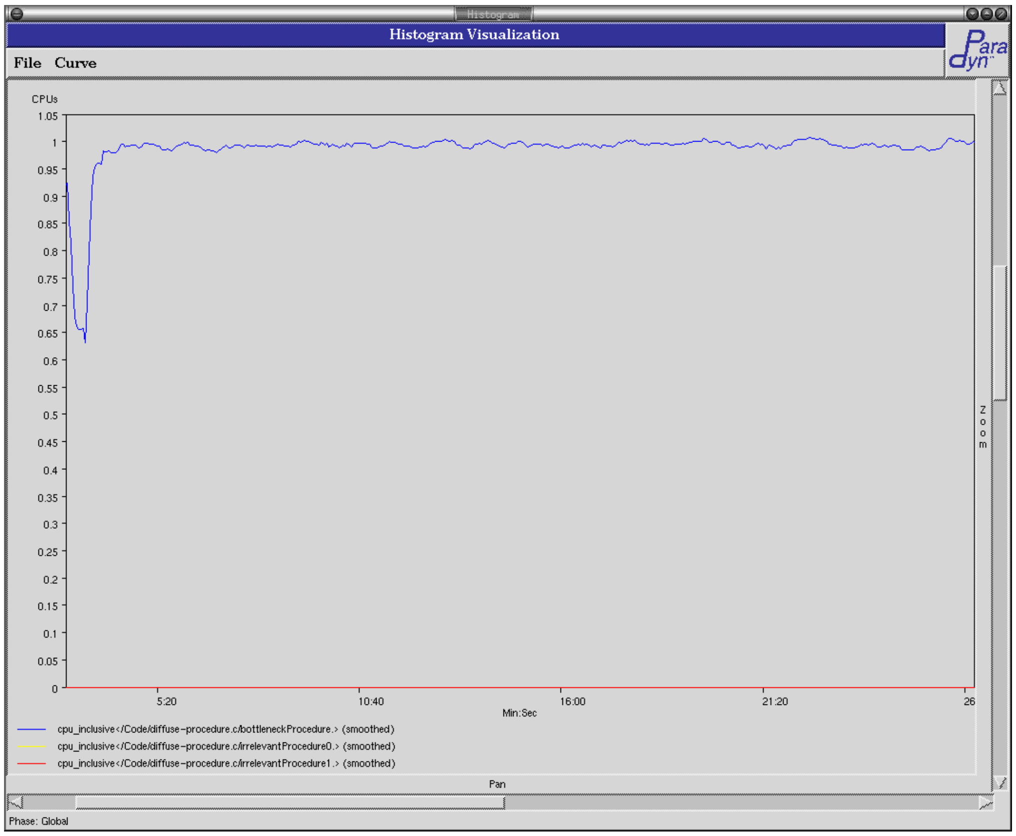


Figure 34: Paradyn Histogram Diffuse-Procedure with LAM/MPI, CPU Inclusive for Three Procedures
This is a Paradyn generated histogram showing CPU inclusive time for three procedures across the whole MPI program. From it we can see that the program is spending more of its time in the `bottleneckProcedure` and hardly any time in the `irrelevantProcedure`s.

of the program's time is spent in `bottleneckProcedure`. If we divide 1 by the number of processes in the application, 4, we get 0.25. This means that only about 25% of a process's time is spent in this function. That is why the Performance Consultant did not consider it to be a computational bottleneck until we set the threshold to be 0.2.

The creators of the Grindstone Test Suite described the program by saying that the `bottleneckProcedure` used 50% of the program's time when using four processes. We found that if we ran the program with only two processes that the Performance Consultant found the `bottleneckProcedure` to be CPU bound without changing the CPU usage threshold. In this case, the procedure was using ~50% of the program's time.

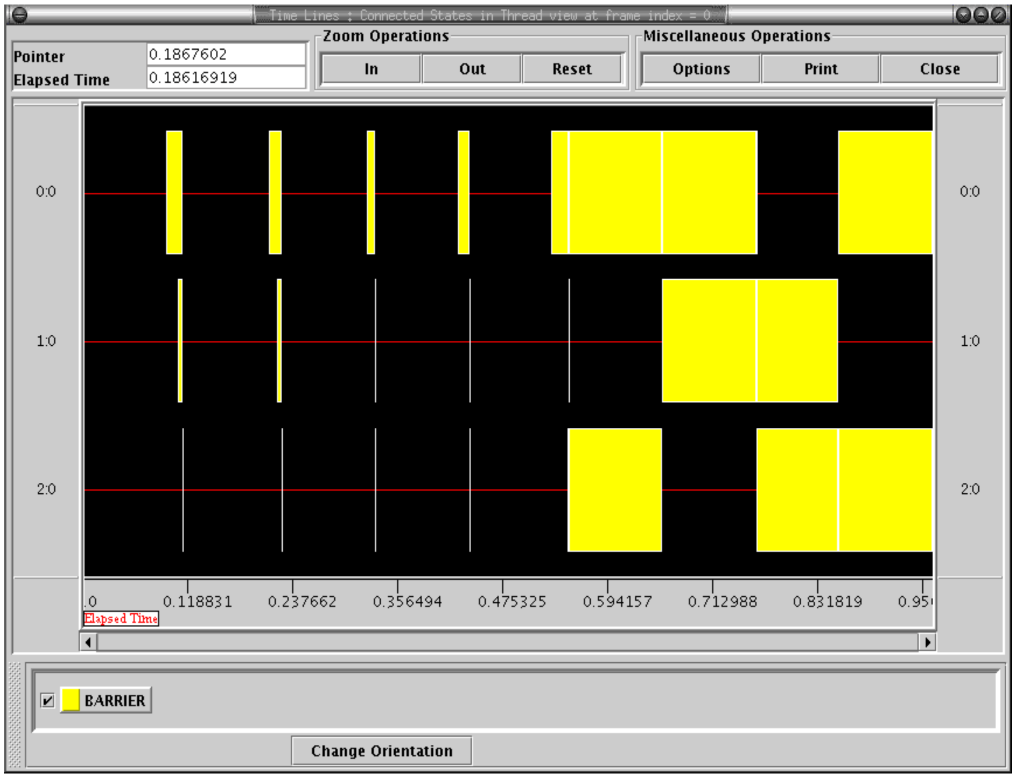


Figure 35: Jumpshot-3 Time Lines Window for Diffuse-Procedure with LAM/MPI
This is a screenshot of the Time Lines Window generated by Jumpshot-3 for the diffuse-procedure program run with LAM/MPI and linked with the MPE libraries. This shows that overall, each of the processes in the application are spending approximately the same amount of time in MPI_Barrier, even though at a specific point in the program the distribution might not be balanced.

The last test for this program, in Figure 35, shows the Time Line window from Jumpshot-3 for a 10 iteration 3 process run of diffuse-procedure. We had to change the parameters for this run because the trace files got too large. Here Paradyn's synchronization findings are confirmed. The program is indeed spending much time in `MPI_Barrier`.

7.2.8 System-Time

The next program used for testing was system-time. The parameters used for the program run was: 10,000 iterations and four processes, two each on two nodes. Paradyn did not pass this test, because Paradyn does not have metrics for measuring the system time of a program. The Performance Consultant found that all top-level hypothesis tested false. The findings for system-time with MPICH are exactly the same as those for LAM/MPI.

7.2.9 Hot-Procedure

The last program in the Grindstone Test Suite we used was hot-procedure. The parameters we used for this program were: 1,000,000 iterations and four processes, two each on two nodes. Figure 36 shows the condensed form of the Performance Consultant's findings for this program for LAM/MPI and MPICH. Both were found to have excessive CPU usage in the function `bottleneckProcedure`.

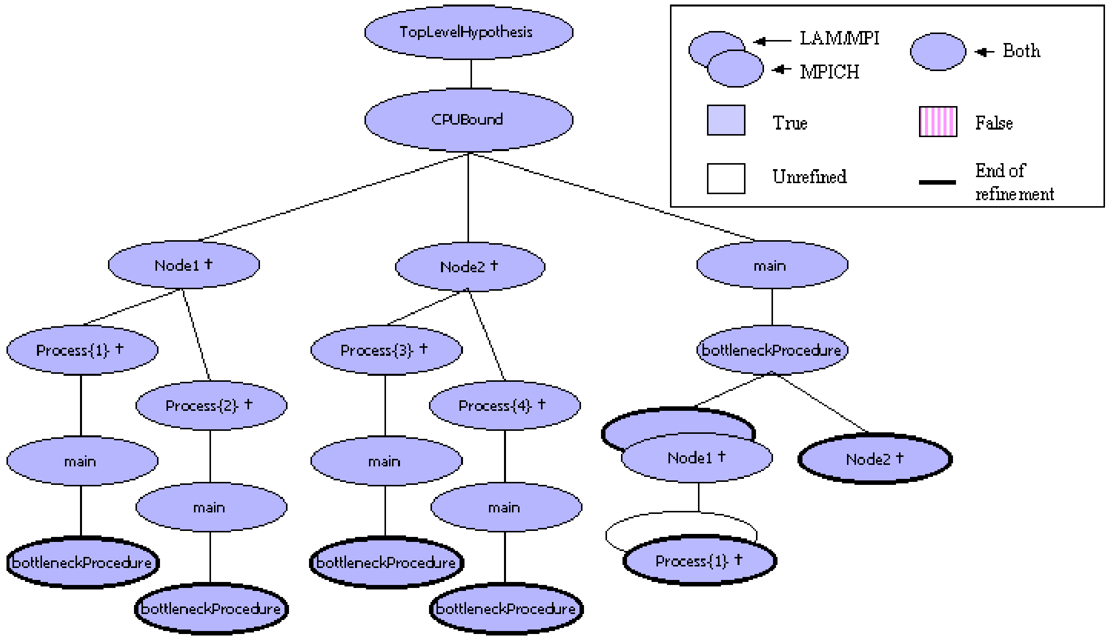


Figure 36: Paradyn PC Output for Hot-Procedure
This is the condensed Performance Consultant output for the hot-procedure program with LAM/MPI and MPICH. For both, the hypothesis CPUBound tested true and the Performance Consultant drilled down to find the source of the bottleneck, `bottleneckProcedure`.

As proof that Paradyn is correctly measuring the CPU time for the functions in this program, Figure 37 shows a portion of the output from gprof generated by a non-MPI version of the hot-procedure program on Linux: It shows that all of the `irrelevantProcedures` indeed take up none of the program's time and that the computational bottleneck is in `bottleneckProcedure`.

```
time    seconds    seconds     calls  us/call  us/call  name
100.00      46.19     46.19      1000 46190.00 46190.00  bottleneckProcedure
0.00     46.19     0.00      1000      0.00      0.00  irrelevantProcedure0
0.00     46.19     0.00      1000      0.00      0.00  irrelevantProcedure1
0.00     46.19     0.00      1000      0.00      0.00  irrelevantProcedure10
0.00     46.19     0.00      1000      0.00      0.00  irrelevantProcedure11
0.00     46.19     0.00      1000      0.00      0.00  irrelevantProcedure12
```

Figure 37: Gprof Analysis of Hot-Procedure
This is gprof output for the hot-procedure program. It shows that the `bottleneckProcedure` is indeed a computational bottleneck.

7.3 A Toy Program Test: ssTwod

For our final test of Paradyn we use a toy program developed in Using MPI: Portable Parallel Programming with the Message-Passing Interface [GLS99]. The book discusses the program as an example for performance tuning message-passing. It is known to have a communication bottleneck in the function `exchng2`, as that function is the focus of the optimization lesson in the book. In Figure 38, we show the con-

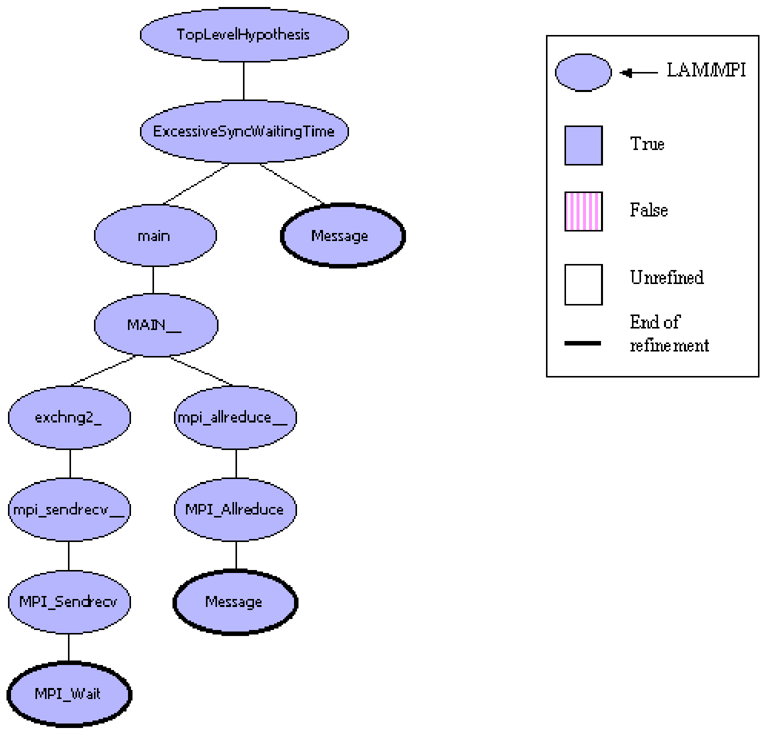


Figure 38: Paradyn PC Output for ssTwod with LAM/MPI
This figure shows the Performance Consultant's findings for the ssTwod program. It found that ExcessiveSyncWaitingTime is true and drilled down to find `MPI_Sendrecv` and `MPI_Allreduce` to be bottlenecks.

densed Performance Consultant's findings for this program. Paradyn is able to find the bottlenecks in this program. It found ExcessiveSyncWaitingTime to be true and drilled down through the function `exchng2` to find `MPI_Sendrecv` to be a bottleneck. It also found a synchronization bottleneck in `MPI_Allreduce`.

7.4 Conclusions

Our testing shows that Paradyn correctly instruments and measures the performance of the MPI-1 features of LAM/MPI for the majority of programs. We verified Paradyn's results for LAM/MPI applications by using test programs with known behavior. We compared what we expected to see, given the program's description, with what Paradyn generated. We also compared the results that Paradyn generated for MPICH programs against what was generated for LAM/MPI programs. Last, we used other performance tools and compared their results with those that Paradyn gave.

We showed that Paradyn found all synchronization bottlenecks. This result is important because synchronization time is a primary concern in message-passing programming. Paradyn also found the computational bottlenecks. The exception was the program system-time. Paradyn does not have metrics for system time measurement, and thus did not find bottlenecks in the program. Discussion of this is outside of the scope of this work.

8 Conclusions and Future Work

The goal of this work was to increase the level of parallel performance tool support for MPI programmers on Linux clusters. We achieved this by implementing support for LAM/MPI into the Paradyn Parallel Performance Tool. In addition, we made alterations to Paradyn to support clusters of workstations with non-shared filesystems. We also verified that Paradyn correctly measures the performance of LAM/MPI programs on Linux clusters. Another contribution of this work was that we outlined and investigated items of interest in MPI-2 for parallel performance tool developers. Furthermore, we designed hypotheses and metrics for MPI-2 features for Paradyn. Then, we explored ways in which changes to the presentation of MPI performance data in Paradyn could expedite the programmer's analysis of the data. We are working with the Paradyn group to incorporate the changes made in this project into the next release of Paradyn.

Our work exposes several key avenues for future work. One of these is to implement support for the MPI-2 features into Paradyn. Another is to implement support for multiple binaries into Paradyn so users could measure the performance of MPMD MPI programs. An additional improvement to Paradyn would be to implement support for running programs on heterogeneous systems with LAM/MPI. Yet another project is to change Paradyn so that only one paradynd daemon needs to run on a node. Currently, one Paradyn daemon runs for each MPI process with LAM/MPI and MPICH ch_p4mpd. This is not strictly necessary and adds to the overhead of the performance tool. It also adds to the perturbation of the studied program as these additional daemons compete with it for resources. Another project of interest is to design a test suite, similar in nature to the Grindstone Test Suite, for testing the performance analysis of MPI-2 functionality by parallel tools.

References

[ANL03] Argonne National Laboratory MPI Home Page. Available from: http://www-unix.mcs.anl.gov/mpi/. December 2003.

[ASQ99] Brandon Allgood, Joachim Stadel, and Tom Quinn. "MPICH and LAM Performance on Astrolab." Available from: http://www-hpcc.astro.washington.edu/faculty/trq/brandon/perform.html. 1999.

[BDV94] Gregory Burns, Raja Daoud, and James Vaigl. "LAM: An open cluster environment for MPI." Proceedings of Supercomputing Symposium '94. John W. Ross, editor. University of Toronto. pp 379-386. 1994.

[BF00] Scott Baden and Stephen Fink. "A Programming Methodology for Dual-tier Multicomputers." IEEE Transactions on Software Engineering 26(3):212-26. March 2000.

[BG02] Gordon Bell and Jim Gray. "What's Next in High-Performance Computing?" Communications of the ACM. 45(2):91-95. February 2002.

[BGG03] Ryan Braby, Jim Garlick, and Robin Goldstone. "Achieving Order through CHAOS: the LLNL HPC Linux Cluster Experience." Cluster World. San Jose, CA. June 23-26, 2003. Available at: http://www.llnl.gov/linux/ucrl-jc-153559.pdf.

[BKS+00] Milind Bhandarkar, L.V. Kal, Eric de Sturler, and Jay Hoeflinger. "Object-Based Adaptive Load Balancing for MPI Programs." PPL Technical report 00-03. University of Illinois at Urbana-Champaign. September 2000.

[BL94] Ralph Butler and Ewing Lusk. "User's guide to the p4 parallel programming system." Technical Report ANL92 /17. Argonne National Laboratory. Mathematics and Computer Science Division. October 1992. Available at: http://www-fp.mcs.anl.gov/~lusk/p4/p4-paper/paper.html. .

[Burn99] Gregory Burns. "Trollius: Early American Transputer Software." Parallelogram. Issue 13. 1999.

[CHH+94] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. "Portable programming with the PARMACS message passing library." Parallel Computing. 20(4):615-632. 1994.

[CLMR99] Phillip Carns, Walter Ligon III, Scott McMillan, and Robert Ross. "An Evaluation of Message Passing Implementations on Beowulf Workstations." Proceedings of the 1999 Extreme Linux Workshop. June 1999.

[Dann02] Jim Danneskiold. “Linux Networx to build Linux supercomputer for Los Alamos.” News and Public Affairs. News Release. Los Alamos National Laboratory. Available at: http://www.lanl.gov/worldview/news/releases/archive/02-106.shtml. September 23, 2002.

[EM98] Antonio Espinosa and Toms Margalef. “Automatic Performance Evaluation of Parallel Programs.” IEEE Proc. of the 6th Euromicro Workshop on Parallel and Distributed Processing. IEEE Computer Society Press. January 1998.

[ELLC03] Etnus LLC. “TotalView User’s Guide.” Document version 6.2. Available on the web at http://www.etnus.com. June 2003.

[FK94] Jon Flower and Adam Kolawa. “Express is not just a message passing system: Current and future directions in Express.” Parallel Computing. 20(4):597-614. 1994.

[FKN+02] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. “The Physiology of the Grid --- An Open Grid Services Architecture for Distributed Systems Integration.” Technical report. Argonne National Laboratory. 2002.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, and Vaidy Sunderam. PVM: Parallel Virtual Machine A Users’ Guide and Tutorial for Networked Parallel Computing. MIT Press. 1994.

[GD02] Jim Garlick and Chris Dunlap. “Linux Project Report.” UCRL-ID-150021. Lawrence Livermore National Laboratory. Available at: http://www.llnl.gov/linux/ucrl-id-150021.pdf. August 18, 2002.

[GHD00] William George, John Hagedorn, and Judith Devaney. “IMPI: Making MPI Interoperable.” with appendix I by IMPI Steering Committee. “IMPI: Interoperable MessagePassing Interface.” Protocol Version 0.0. January, 2000. http://impi.nist.gov/IMPI/. Journal of Research of the National Institute of Standards and Technology. MayJune 2000.

[GHL+98] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. MPI- The Complete Reference: Volume 2, The MPI Extensions. MIT Press. 1998.

[GLD+96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. “A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard.” Parallel Computing. North-Holland. vol. 22. pp. 789-828. 1996.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. Second Edition. MIT Press. 1999.

[GLT99] William Gropp, Ewing Lusk, and Rajeev Thakur. Using MPI-2: Advanced Features of the Message-Passing Interface. MIT Press. 1999.

[GS93] William Gropp and Barry Smith. "Users Manual for the Chameleon Parallel Programming Tools." Technical Report ANL93 /23. Argonne National Laboratory. 1993.

[HF93] Michael T. Heath and Jennifer E. Finger. "ParaGraph: A Tool for Visualizing Performance of Parallel Programs." Technical Report Oak Ridge National Lab. 1993.

[Kear03] Brian Kearns. "A Performance Study of LAM and MPICH on an SMP Cluster." Master's Thesis in Computer Science. Portland State University. 2003.

[HS96] Jeffrey Hollingsworth and Michael Steel. "Grindstone: A Test Suite for Parallel Performance Tools." University of Maryland Computer Science Technical Report. CS-TR-3703. October, 1996.

[HX98] Kai Hwang and Zhiwei Xu. Scalable Parallel Computing. McGraw-Hill. 1998.

[LANL02] Los Alamos National Laboratory Public Affairs Office. "High Performance Computing For National Security." News and Public Affairs. Los Alamos National Laboratory. Available at: http://www.lanl.gov/worldview/news/pdf/HighPerf_Computing.pdf. May 2002.

[LLNL02] "Meeting Enduring National Needs. Livermore's New Unclassified Supercomputer in Top Five." Lawrence Livermore National Laboratory 2002 Annual Report. Lawrence Livermore National Laboratory. Available at: http://www.llnl.gov/annual02/pdfs/national.pdf. 2002.

[LT00] LAM Team. "Porting the LAM-MPI 6.3 Communication Layer." Available from: http://www.lam-mpi.org/download/. Filename: lam_rip.ps. March 8, 2000.

[LTJ03] Lam Team. "LAM / MPI Parallel Computing: XMPI - A Run/Debug GUI for MPI." Available on the web at http://www.lam-mpi.org/software/xmpi. June 2003.

[LTA03] LAM Team. “The History of LAM/MPI.” http://www.lam-mpi.org/about/overview/history.php. August 2003.

[Malo02] Staci Maloof. “World’s Most Powerful Linux-based Supercomputer.” DOE Science News. Office of Science. U.S. Department of Energy. Available at: http://www.science.doe.gov/Science_News/feature_articles_2002/April/PNNL_supercomputer/PNNL-Supercomputer.htm. April 22, 2002.

[May01] John May. Parallel I/O for High Performance Computing. Academic Press. 2001.

[MB93] John May and Francine Berman. “Panorama: A Portable, Extensible Parallel Debugger.” ACM/ONR Workshop on Parallel and Distributed Debugging. ACM SIGPLAN 28(12):96-106. December 1993.

[MBM94] Bernd Mohr, Darryl Brown, and Allen D. Malony. “TAU: A portable parallel program analysis environment for pC++.” In Proceedings of CONPAR 94 - VAPP VI. University of Linz, Austria. September 1994.

[MCC+95] Barton Miller, Mark Callaghan, Jonathan Cargille, Jeffrey Hollingsworth, Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam and Tia Newhall. “The Paradyn Performance Tools.” IEEE Computer 28(11):37-46. November 1995.

[MPI03] Message Passing Interface Forum. http://www.mpi-forum.org/. September 2003.

[MPIC03] MPICH - A Portable MPI Implementation. http://www-unix.mcs.anl.gov/mpi/mpich/. October 2003.

[MT02] Bernd Mohr and Jesper Traff. “Intitial Design of a Test Suite for Autmatic Performance Analysis Tools.” APART Technical Report. FZJ-ZAM-IB-2002-13. October 2002. Available at: http://www.fz-juelich.de/apart.

[NAW+96] Wolfgang E. Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. “VAMPIR: Visualization and analysis of MPI resources.” Supercomputer, 12(1):69-80. January 1996.

[Nevi96] Nick Nevin. “The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster.” Ohio Supercomputing Center. Technical Report OSC-TR-1196-4. Columbus, OH. 1996.

[NSB03] National Science Board. “Science and Engineering Infrastructure for the 21st Century: The Role of the National Science Foundation.” NSF Report NSB-02-190. February 2003.

[OF00] Hong Ong and Paul Farrell. “Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network.” Proceedings of the 4th Annual Linux Showcase & Conference. Atlanta, GA. October 10-14, 2000.

[PCL+02] William Putman, Jiundar Chern, Shian-Jiann Lin, William Sawyer, and Bo-Wen Shen. “Modeling the Earth’s Atmosphere.” Presented at SC2002. NASA Goddard Space Flight Center. November 18-21, 2002.

[PG03] The Paradyn Group. “Paradyn Parallel Performance Tools User’s Guide: Release 4.0.” Available at http://www.cs.wisc.edu/paradyn/manuals.html. October 2003.

[PTC03] The Parallel Tools Consortium. PTools MQM working group home page. Available on the web at http://www.ptools.org/projects/mqm/. June 2003.

[RAD+98] Daniel Reed, Ruth Aydt, Luiz DeRose, Celso Mendes, Randy Ribler, Eric Shaffer, Huseyin Simitci, Jeffrey Vetter, Daniel Wells, Shannon Whitmore, and Ying Zhang. “Performance Analysis of Parallel Systems: Approaches and Open Problems.” Proceedings of the Joint Symposium on Parallel Processing (JSPP), pp. 239-256. June 1998.

[RAN+93] Daniel Reed, Ruth Aydt, Roger Noe, Philip Roth, Keith Shields, Bradley Schwartz, and Luis Tavera. “Scalable performance analysis: The Pablo performance analysis environment.” In Proceedings of the Scalable Parallel Libraries Conference, A. Skjellum, Ed. IEEE Computer Society. pp. 104-113. 1993.

[Saph97] William Saphir. “A Survey of MPI IMplementations.” Lawrence Berkeley National Laboratory. University of California. Berkeley, CA. Nov. 6, 1997. Available from: http://www-library.lbl.gov/docs/LBNL/410/25/PDF/LBNL-41025.pdf.

[Schw01] David Schwoegler. “Fact Sheet: ASCI White Dedication.” News Release. Lawrence Livermore National Laboratory. Available at: http://www.llnl.gov/llnl/06news/NewsReleases/2001/NR-01-08-03b.html. August 15, 2001.

[Seit01] Charles Seitz. “Recent Advances in Cluster Networks.” Keynote address presented at IEEE Cluster 2001 Conference. Newport Beach, CA. October 8-11, 2001.

[SLG+00] Jeffrey Squyers, Andrew Lumsdaine, William George, John Hagedorn, and Judith Devaney. “The Interoperable Message Passing Interface (IMPI) Extensions to LAM/MPI.” In Proceedings, MPI Developer's Conference. Cornell, NY. 2000.

[SOH+99] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. MPI - The Complete Reference. Volume 1, The MPI Core. 2nd Ed. MIT Press. 1999.

[SSD+94] Anthony Skjellum, Steven Smith, Nathan Doss, Alvin Leung, and Manfred Morari. The Design and Evolution of Zipcode. Parallel Computing 20(4):565-596. 1994.

[SSB+99] Thomas Sterling, John Salmon, Donald Becker, and Daniel Savarese. How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters. MIT Press. 1999.

[TWS03] Valerie Taylor, Xingfu Wu, and Rick Stevens. “Prophesy: An Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications.” ACM SIGMETRICS Performance Evaluation Review 30(4):13-18. March 2003.

[Thak00] Rajeev Thakur. “MPICH on Clusters: Future Directions.” Technical Paper presented at Linux Supercluster Users Conference. September 11-15, 2000. Argonne National Laboratory. Available at www.linuxclustersinstitute.org/ Linux-HPC-Revolution/ Archive/PDF00/Thakur.pdf.

[Top503] Top 500 Supercomputer Sites. Available at: http://www.top500.org. September 1, 2003.

[Vett02] Jeffrey Vetter. “Dynamical Statistical Profiling of Communication Activity in Distributed Applications.” ACM SIGMETRICS 2002. June 15-19, 2002. Marina Del Rey, CA. Performance Evaluation Review 30(1):240-249. June 2002.

[VM01] Jeffrey Vetter and Michael McCracken. "Statistical Scalability Analysis of Communication Operations in Distributed Applications." Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP). 2001.

[VM02] Jeffrey Vetter and Frank Mueller. “Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures.” Proceedings of the International Parallel and Distributed Processing Symposium. 2002.

[WM00] Felix Wolf and Bernd Mohr. “Automatic Performance Analysis of MPI Applications Based on Event Traces.” Proceedings of European Conference on Parallel Computing (Euro-Par). Munich, Germany. August 2000.

[WM01] Felix Wolf and Bernd Mohr. “Automatic Performance Analysis of SMP Cluster Applications.” Forschungszentrum Jülich, ZAM. Technical Report IB-2001-05. 2001.

[Yan94] Jerry C. Yan. “Performance Tuning with AIMS -- An Automated Instrumentation and Monitoring System for Multicomputers.” Proceedings of the 27th Hawaii International Conference on System Sciences. Hawaii. January 1994.

[ZG03] Mary Zosel and John Gyllenhaal. Private communication. Lawrence Livermore National Laboratory. October 2003.

[ZLGW99] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. “Toward scalable performance visualization with Jumpshot.” High Performance Computing Applications. 13(2):277-288. Fall 1999.