Performance Tool Support for MPI-2 on Linux[1]

Kathryn Mohror and Karen L. Karavanic
Portland State University
{kathryn, karavan}@cs.pdx.edu

Abstract

Programmers of message-passing codes for clusters of workstations 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.

This paper reports on our recent efforts developing a performance tool for MPI applications on Linux clusters. Our target MPI implementations were LAM/MPI and MPICH2, both of which support portions of the MPI-2 Standard. We started with an existing performance tool and added support for non-shared file systems, MPI-2 one-sided communications, dynamic process creation, and MPI Object naming. We present results using the enhanced version of the tool to examine the performance of several applications. We describe a new performance tool benchmark suite we have developed, PPerfMark, and present results for the benchmark using the enhanced tool.

1 Introduction

Developing efficient parallel programs for medium to large scale Linux clusters is a challenging task. A primary difficulty is dividing the work and data needed to compute a solution into distinct processes that can be run on separate computers. This division needs to take into account not only computational correctness, but also needs to minimize communication overhead. Communication between processes on separate nodes of a cluster is costly because the nodes do not share memory. As the number of cluster nodes increases, it is becomes increasingly difficult for the software engineer to maintain a clear understanding of the state of the application at any given moment during execution.

Improving this development process 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 more expensive specialized supercomputers, as well as for production use. Second, 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. Third, the problems being solved using Linux clusters are important and include: DNA analysis, drug design, global weather prediction, nuclear simulations, and molecular modeling.

We are addressing this need by strengthening the parallel performance tool base for MPI programmers on Linux clusters. Parallel applications for Linux clusters are commonly developed using a message-passing library such as MPI [28]. MPI defines an interface; there are many commercial and free implementations available. We have extended an existing parallel performance tool, Paradyn [18] to support performance analysis of applications developed with either of two freely-available MPI implementations: MPICH and LAM [12, 4]. Our goal is availability of a single performance tool that supports all MPI features, including the newer MPI-2 Standard features, for the Linux platform. In this paper, we present results for all MPI-1 and selected MPI-2 features.

Major efforts have been undertaken to develop supercomputing-caliber Linux clusters by the Department of Energy and the National Science Foundation (NSF). In June of 2003, the MCR Linux cluster at Lawrence Livermore National Laboratory (LLNL), running with commodity processors, was ranked as the third fastest system in the world with a peak computing speed of 11 teraflops [30]. The TeraGrid is an NSF-funded project to build the world's first supercomputing grid. The primary computational units of this system are Linux clusters [29]. Reasons reported for the increased use of Linux clusters include: the low price/performance ratio when using commodity or near-commodity parts; the open-source nature of the operating system; and the overall increased availability and manageability of the system as compared to proprietary systems [9, 3].

The increased use of Linux clusters for parallel applications has led to an urgent need for improved software support tools. The National Science Foundation finds the need for support software for users to be urgent and recom-

[1] This work was supported in part by UC/LLNL contract B524838.

mends that more work be done to develop software tools for supercomputing platforms [22]. The researchers in the Linux Project at LLNL report that there is a need for system software to support Linux clusters [9]. 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 [2]. The need for software tools is especially great for Linux clusters because the nodes of a Linux cluster are distinct complete computers, and, in general, are not designed for the purpose of clustering. Because of this, there is a need for software to make the nodes of a cluster act as one computing machine.

The Message Passing Interface (MPI) emerged as a standard in 1994 as MPI-1 and was widely accepted by the scientific programming community. 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, LAM 7.0.4 has support for most of MPI-2 and the current beta release of MPICH2 supports a substantial portion. There is little performance tool support for these new features, likely because the MPI implementations had not yet provided for them until recently, so there wasn't much demand. Interest in performance tuning MPI-2 features will likely increase now that the freely-available MPI implementations 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 reported 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 [24].

Our goal is to strengthen the parallel performance tool base for MPI programmers on Linux clusters. To achieve this, we chose to increase the level of support for MPI in an existing parallel performance tool, Paradyn [18]. Paradyn is a run time profiling tool developed at the University of Wisconsin that utilizes dynamic instrumentation to insert and delete measurement instructions at run time [15].

We chose Paradyn instead of creating our own new tool because the source code for Paradyn is freely available and is well-documented, its mechanism for creating new metrics and resource constraints is extensible, and it already supported MPICH on Linux clusters with shared filesystems. Paradyn's Performance Consultant module automatically searches for performance bottlenecks in user programs. This is ideal for long-running programs, because it eliminates the need for manually searching through execution trace visualizations for potential performance problems. Paradyn is convenient for the application programmer, because the application does not need to be recompiled in order for performance measurement to be possible. Its use of dynamic instrumentation can dramatically decrease the amount of data that must be collected over the course of the program, as the decision on what to instrument can be made dynamically. Performance measurement instructions only need to be inserted in code sections where a performance problem is suspected.

In the following section we summarize related research. Section 3 contains a discussion of issues for performance tool developers. Section 4 describes the changes we made to Paradyn for the new functionality. Section 5 presents results from a variety of tests we conducted to verify the enhanced tool's performance measurements. We conclude and discuss future work in Section 6.

2 Related Work

Parallel performance tools fall into two main categories: post-mortem analysis tools and dynamic instrumentation tools. Post-mortem tools that measure MPI applications include: Jumpshot, Vampir, SeeWithin/Pro, Paraver, mpiP, Pablo, TAU, KOJAK, Prophesy, and PE Benchmarker [34, 21, 5, 25, 32, 26, 17, 33, 31, 23]. All of these tools allow the user to analyze detailed performance data of MPI programs at the end of the program's execution. Feedback is given to the user through either a visualization of the execution trace (Jumpshot, Vampir, Paraver), post-mortem performance analysis (SeeWithin/Pro, mpiP, KOJAK, Prophesy), or both (Pablo, TAU, PE Benchmarker). In general, these tools suffer from a scalability limit caused by the tool's generation of unmanageably large trace files. An exception is mpiP, which uses profiling information to perform its analysis of the MPI program. Dynamic instrumentation is the insertion and removal of instrumentation instructions during program execution [15]. This greatly reduces the total amount of data collected during each measured application run. Dynamic instrumentation tools include DPCL, TAU, Paradyn, SIGMA, DynaProf, Autopilot, KOJAK, and PE Benchmarker [8, 17, 18, 7, 20, 27, 33, 23]. Of the available performance tools, Jumpshot, KOJAK, Vampir, Paraver, SeeWithin/Pro, mpiP, Pablo, Paradyn, and TAU can be used to measure MPI applications on Linux clusters.

We found several tools that support MPI-2 features of MPI. TAU supports performance measurement of mixed-language programs, which means that a user could measure the performance of a program with source files written in a combination of C, C++, and Fortran 90. Vampir supports MPI-I/O and 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 run time performance viewing. It also does not provide any automated performance diagnosis. Pablo supports the MPI-I/O features of MPI-2. Pablo utilizes source code instrumentation, so the user cannot change what performance data is collected at run time as can be done with Paradyn. Also, Pablo MPI-I/O support has not been tested on the Linux platform. Prophesy has some MPI-2 support. Prophesy's goal is performance prediction of applications on different systems using performance data from multiple program runs. This contrasts with that of Paradyn: automated performance diagnosis at run time. We believe that a full implementation of our enhanced version of Paradyn will greatly improve the performance tool support available for MPI applications on Linux clusters.

3 Issues for MPI-2 Performance Tool Developers

The MPI-2 Standard introduces a number of new features. The most important new functionalities of MPI-2 that are of interest to performance tool developers are: dynamic process creation, remote memory access (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. We discuss each of these features in turn and point out the topics of interest to performance tool developers.

Dynamic process creation is an important feature for two reasons: spawn operations could represent significant performance bottlenecks; and tools cannot determine the number of application processes until run time. An MPI spawn operation includes process creation overhead. Also, the operation is collective over two sets of processes, the parent group of processes and the child group of processes, so there is a potential synchronization overhead. We believe that MPI programmers will want to know the specific performance costs to their programs from these operations. Performance tools need to be able to detect the newly spawned processes at run time in order to measure them with the rest of the application.

RMA 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 run time. 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 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 synchronization. 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. The use of RMA 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. Examples of RMA synchronization methods are shown in Figure 1. 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. MPI-I/O does not refer to terminal I/O (stdout, stdin, stderr), but to file access. The interface allows MPI processes to access shared files collectively or individually. 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.

Additional 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 must support multi-threaded applications. The ability to name MPI objects is of importance, because the performance tool should display the user defined names for MPI objects in the user interface.

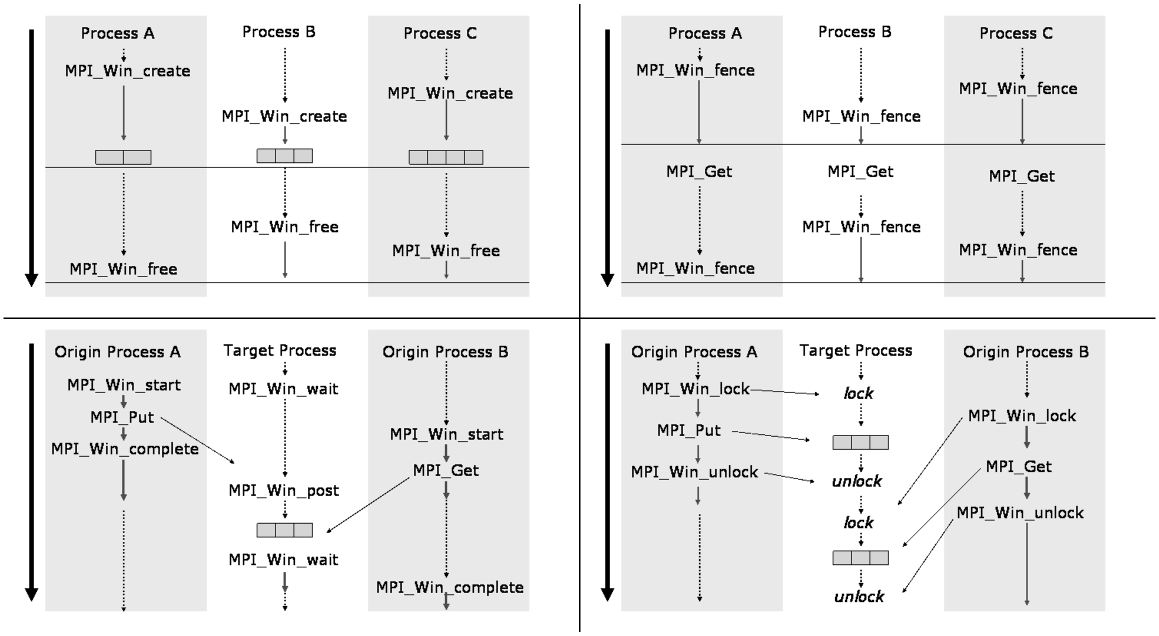


Figure 1: RMA Synchronization

This figure shows different remote memory access synchronization patterns. The top left figure shows three processes that are collectively creating a new RMA window. It illustrates the synchronization overhead that could occur if a process were late in executing `MPI_Win_create`. The top right figure shows the simplest form of RMA active target synchronization, using `MPI_Win_fence`. We see that if Process B is late executing the fence, then processes A and C may incur synchronization waiting time as a result. The diagram in the bottom left shows the second form of active target synchronization. Although the MPI Standard is flexible in its specification of which of these routines will be blocking, synchronization overhead can occur. The bottom right diagram shows RMA passive target synchronization. Synchronization waiting time can occur because a `MPI_Win_unlock` is not allowed to return until all of its data transfers have completed at both the origin and the target.

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.

4 Implementation

In this section, we describe the implementation of our changes to Paradyn. First we describe our starting point, the functionality already included in Paradyn at the start of our efforts. Next, we detail changes made to enhance support of MPI-1 level features. Finally, we describe the implementation for MPI-2 features. We show the results of our enhanced version in Section 5.

We started our project with an unmodified copy of Paradyn version 4.0 from the Paradyn Group download site (http://www.paradyn.org). This version already included support for MPICH 1.2.4 ch_p4 on Linux clusters. Paradyn is an automated parallel performance tool developed at the University of Wisconsin. Paradyn is a profiling tool that employs dynamic instrumentation to insert performance measurement instructions into programs at run time. 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 consists of a front end process to collect and visualize data and search for performance bottlenecks; and daemons that run on each machine node, inserting and deleting instrumentation into the application processes, and collecting and forwarding performance data. A Paradyn daemon is assigned one or more processes. A Paradyn user can either request data for specific metrics, or can run an automated search for performance problems using the Performance Consultant module. Either approach results in new instrumentation being inserted into the application, specified by metric-focus pairs, where the metric specifies what to measure, and the focus specifies what parts of the application (which processes, or functions, or synchronization objects) to include in the measurement. Resources are the building blocks for foci. Examples of resources include functions, processes, machine nodes, and message tags.

Application resources are organized into a tree structure called the Resource Hierarchy. The root of the tree is the Whole Program. The next level in the hierarchy consists of three general categories: Code, Machine, and SyncObject. Beneath these are more specific subcategories. For example, the children of SyncObject are types of synchronization objects such as Message and Barrier. A particular resource is uniquely identified by traversing the tree from the root to the particular resource. As an example, an MPI communicator 'X' would be identified by /SyncObject/Message/X, because it is a synchronization object associated with message passing.

Users are able to modify and extend Paradyn's functionality through its Paradyn Configuration Language (PCL) and Metric Description Language (MDL). The PCL allows users to modify Paradyn's default behavior by specifying daemon and process attributes, defining visualizations of performance data, and changing the value of default constants within Paradyn. In order to define new metrics, MDL, a sub-language of PCL, is used. The MDL can also define resource constraints, which are used to restrict the metric to a particular resource.

4.1 Changes to Support MPI-1 Features

Our first goal was to correctly measure both LAM and MPICH MPI-1 applications on our Linux cluster. We added support for non-shared filesystems, LAM and some additional support for MPICH. For support of non-shared file systems, we altered the daemon definition by the addition an optional attribute that specifies the MPI implementation (i.e. LAM or MPICH). We also eliminated an intermediate step in the way Paradyn starts MPI processes on Linux. Originally, Paradyn generated a custom script that would be executed by `mpirun`. This script would then start the Paradyn daemons, which would start the actual MPI processes. There was overhead in parsing machine files and determining from user arguments to `mpirun` where the MPI processes would be started. To avoid this, we eliminated the Paradyn generated script from the steps. This required adding new command line arguments to the Paradyn daemon and changing the default way that the Paradyn daemon handled the MPI process's file descriptors. Results of our enhanced version are presented in Section 5.

4.1.1 MPICH Specific Changes

To support non-shared filesystems and the newer MPICH version 1.2.5 ch_p4mpd, we added code to process the `-m` and `-wdir` arguments to `mpirun`. These arguments specify a machine file and working directory, respectively.

We changed Paradyn's metric definition file to take into account MPICH's profiling interface. The MPI Specification requires that every MPI routine 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`. A user can override MPICH's default behavior and make two copies of the library by giving the `--disable-weak-symbols` argument 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++.

4.1.2 LAM Specific Changes to Support MPI-1

LAM 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 machine file.

We implemented support for three commonly used 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 des-

ignate a subset of the nodes using a LAM 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 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.

4.2 Support for MPI-2 Features

Our goal is complete performance tool support of MPI-2 features on the Linux platform. We have made strides to achieving that goal by implementing support for RMA, dynamic process creation, and MPI object naming into Paradyn.

4.2.1 One-sided Communication

We added a new type of synchronization object to Paradyn's Resource Hierarchy for RMA Windows, /SyncObject/Window. RMA windows are created at run time, so we added code to Paradyn to dynamically instrument the function `MPI_Win_create` to detect new windows. Upon detection of the newly created RMA window, it is added as a new resource in Paradyn's Resource Hierarchy under /SyncObject/Window. We insert the instrumentation into the MPI program at the function return of `MPI_Win_create`. At this point, we can collect the identifier given to the RMA window by the MPI implementation. We give the new RMA window a slightly more complex identifier in Paradyn to ensure its uniqueness in the Resource Hierarchy. The reason for this is that the MPI implementation may choose to reuse a window identifier after a previously existing window using that same identifier has been deallocated with `MPI_Win_free`. We represent an RMA window in the Resource Hierarchy by 'N-M', where N is the identifier given to the window by the MPI implementation, and M is a number that makes the pair unique. We also detect when an RMA window is freed with`MPI_Win_free`. The `MPI_Win_free` function is instrumented to mark the RMA window as retired in the Resource Hierarchy.

In addition, we defined metric definitions for RMA specific metrics (See Table 1). The metrics were created to measure performance specifically related to MPI one-sided communications. We also created a resource constraint for the metrics. The constraint allows the user to extract performance information related to just one RMA window, by choosing it as a focus in the Resource Hierarchy. We introduce four types of metrics for performance measurement of RMA operations: active target synchronization metrics, passive target synchronization metrics, general synchronization metrics, and data transfer metrics. Examples of metric definitions and the resource constraint are shown in Figure 2.

We selected the 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 MPI-2 Standard says that the function `MPI_Win_wait` will block until all outstanding `MPI_Win_complete` calls have been issued, and as a result could add to the synchronization waiting time, so it is incorporated into the active target metrics. The function `MPI_Win_start` could cause synchronization waiting time, because 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 1. 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 arguments.

The passive target metrics give the wall clock time spent in the passive target RMA routines shown in column 3 per unit time. 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 passive target metrics. They are included instead in the general RMA synchronization metrics, because the data transfer routines can 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

Metric	Description	MPI Functions
rma_put_ops	A count of the number of Put operations per unit time.	`MPI_Put`
rma_get_ops	A count of the number of Get operations per unit time.	`MPI_Get`
rma_acc_ops	A count of the number of Accumulate operations per unit time.	`MPI_Accumulate`

rma_ops	A count of the number of Put, Get, and Accumulate operations per unit time.	`MPI_Put MPI_Get`
		`MPI_Accumulate`

rma_put_bytes	Number of bytes put per unit time.	`MPI_Put`
rma_get_bytes	Number of bytes gotten per unit time.	`MPI_Get`
rma_acc_bytes	Number of bytes accumulated in the target process.	`MPI_Accumulate`

rma_bytes	Sum of RMA byte count metrics.	`MPI_Put MPI_Get`
		`MPI_Accumulate`

at_rma_sync_wait	Wall clock time spent in active target RMA synchronization routines during time interval.	`MPI_Win_fence`
		`MPI_Win_start`
		`MPI_Win_complete`
		`MPI_Win_wait`

pt_rma_sync_wait	Wall clock time spent in passive target RMA synchronization routines during time interval.	`MPI_Win_lock` `MPI_Win_unlock`

rma_sync_wait	Wall clock time spent in RMA synchronization routines during time interval.	`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_ops	A count of the number of RMA synchronization operations per unit time.	

Table 1: RMA Metrics

This table shows the metric definitions that we created for performance measurement of MPI-2 programs with Paradyn. With these the user can get a count of the number of RMA operations, bytes transferred by those operations, and the synchronization overhead incurred as a result of RMA operations.

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 general RMA synchronization wall clock time include the passive target and active target synchronization routines, `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. Also, the data transfer routines are included in the general RMA metric as they could contribute to either passive target or active target synchronization.

The data transfer metrics measure RMA operation counts and the number of bytes transferred by them. The metric definitions for the data transfer metrics include the routines `MPI_Put, MPI_Get`, and `MPI_Accumulate.`

4.2.2 Dynamic Process Creation

The major difficulty in supporting `MPI_Comm_spawn` is knowing where the MPI implementation starts the new processes. The MPI Forum chose to define `MPI_Comm_spawn` in such a way that it does not define a platform-independent interface to a process manager. Instead, they allow the user to provide hints about where to start the processes via reserved keys in the `info` argument. However, an MPI implementation is free to ignore the keys if it chooses. An implementation may also define its own `info` keys, and use them to direct where the new processes will be started. For instance, LAM defines the `lam_spawn_file` key to be the name of a file that contains a LAM application schema, which has all the information that is needed for the LAM daemon to know where to start the new processes. If the user does not provide any hints to the spawn call, then the spawned processes will be started in some MPI implementation dependent way.

For these reasons, there is no implementation independent way to determine where the new processes have started based on the arguments to the `MPI_Comm_spawn` call itself. We designed two methods to implement support for MPI spawn operations, *intercept* and *attach*.

The *intercept* method intercepts the call to `MPI_Comm_spawn` and replaces the user-provided `command` argument with "paradynd." We also replace the `argv` argument with values for the Paradyn daemon, which give it information about how to contact the Paradyn front end and what MPI process to start. The other arguments are left intact. This way the MPI implementation will start the Paradyn daemons according to its policies. The Paradyn daemons then

```
metric mpi_rma_put_ops {
  name              "rma_put_ops";
  units             ops;
  aggregateOperator sum;
  style             EventCounter;
  flavor            { mpi };
  unitsType         unnormalized;

  constraint moduleConstraint;
  constraint procedureConstraint;
  constraint mpi_WindowConstraint;

  base is counter {
   foreach func in mpi_put {
    append preInsn func.entry constrained
(* mpi_rma_put_ops++; *)
    }
  }
}
```

```
metric mpi_rma_syncWait {
  name              "rma_sync_wait";
  units             CPUs;
  aggregateOperator sum;
  style             EventCounter;
  flavor            { mpi };
  unitsType         normalized;
  constraint procedureConstraint;
  constraint moduleConstraint;
  constraint mpi_syncObjConstraint;
  constraint mpi_WindowConstraint;
  base is wallTimer {
    foreach func in mpi_rma_sync {
      append  preInsn func.entry  constrained
      (* startWallTimer(mpi_rma_syncWait); *)
      prepend preInsn func.return constrained
       (* stopWallTimer( mpi_rma_syncWait); *)
    }
    foreach func in mpi_all_calls { }
  }
}
```

```
  metric mpi_rma_put_bytes {
  name              "rma_put_bytes";
  units             bytes;
  aggregateOperator sum;
  style             EventCounter;
  flavor            { mpi };

  constraint moduleConstraint;
  constraint procedureConstraint;
  constraint mpi_WindowConstraint;

  counter bytes;
  counter count;

  base is counter {
    foreach func in mpi_put{
      append preInsn func.entry constrained
        (* MPI_Type_size($arg[2], &bytes);
           count = $arg[1];
           mpi_rma_put_bytes += bytes * count;
          *)
    }
  }
}
```

```
constraint mpi_WindowConstraint /SyncObject/Window
is counter {
  foreach func in mpi_get{
    prepend preInsn func.entry
     (* if (DYNINSTTWindow_FindUniqueId($arg[7]) ==
$constraint[0])
                 mpi_WindowConstraint = 1;
       *)
    append preInsn func.return
      (* mpi_WindowConstraint = 0; *)
  }
  foreach func in mpi_put{
    prepend preInsn func.entry
     (* if (DYNINSTTWindow_FindUniqueId($arg[7]) ==
$constraint[0])
             mpi_WindowConstraint = 1;
       *)
    append preInsn func.return
      (* mpi_WindowConstraint = 0; *)
  }.....
  // includes rest of MPI_Win routines
  }
```

Figure 2: RMA Metric Definition and Constraint Examples

This figure shows examples of metric definitions and a resource constraint that we added to Paradyn for the performance measurement of MPI-2 programs. On the top left is the metric definition for rma_put_ops. It specifies that the counter mpi_rma_put_ops should be incremented each time a call to MPI_Put is executed. The top right box shows the metric definition for rma_sync_wait. It starts a timer at the beginning of each RMA synchronization routine and stops it at the routine's exit, measuring the wall clock time spent in RMA synchronization. On the bottom left is the metric definition for rma_put_bytes. It specifies that at the function entry ofMPI_Put, a call to MPI_Type_size will be made with the MPI_Type argument to MPI_Put The size of that type is returned in the variable bytes. The number of bytes transferred by MPI_Put is bytes multiplied by the count argument to MPI_Put. On the bottom right is the resource constraint for RMA windows. It specifies that for each MPI_Win function, at function entry, the MPI_Win argument will be examined with DYNINSTWindow_FindUniqueId to see if Paradyn recognizes that window as a resource. If so, a flag is set to specify that the particular resource is active. In the interest of space, only the entries forMPI_Put and MPI_Get are shown. The constraint in its entirety contains entries for all MPI_Win functions.

start the MPI processes that the user specified. While this approach is simple, it has the drawback of adding overhead to the spawning operation. If the user wanted to measure the performance cost of spawning operations, this method would inflate the measured values. It also starts a new Paradyn daemon for each new process, which is not strictly necessary. We implemented the intercept method by making a library that contains wrapper functions for MPI_Init and MPI_Comm_spawn, using the MPI profiling interface. The wrapper for MPI_Init gathers the information necessary for the startup of the Paradyn daemon. The MPI_Comm_spawn wrapper generates new command and argv arguments for the Paradyn daemon and then calls PMPI_Comm_spawn with those arguments. Paradyn detects the newly spawned processes and incorporates them into the Resource Hierarchy.

The *attach* method lets the call to MPI_Comm_spawn proceed without interference, discovers where the newly spawned processes were created, and then attaches a Paradyn daemon to the new processes. This method has the advantage of adding less overhead to the spawning operation compared to the intercept method. However, it has higher implementation complexity because we now need to determine where the new processes were started. One way to support the attach method in Paradyn would be to use the MPI Debugging Interface [6] to get information about the new processes. The purpose of the interface is to allow debuggers to get detailed information about MPI application objects, such as communicators and message queues through the interface. Paradyn could insert instrumentation at the function exit of the call to `MPI_Comm_spawn` to query the interface for the new process information. The interface has a global variable `MPIR_proctable` that holds information about every process in the application. It also has a function to query for new processes that have been added to the application by spawning operations. Unfortunately, as of this writing, neither LAM nor MPICH2 support the dynamic process creation parts of the debugging interface.

We consider our existing support for dynamic process creation in Paradyn to be a first effort. We feel that a better solution would be to detect the spawn operation using dynamic instrumentation, then attach to the newly created processes using information such as that provided by the MPI Debugging Interface, should it become available.

4.2.3 Naming of MPI Objects

The MPI-2 Standard allows a programmer to give user-friendly names to RMA windows, MPI communicators, and MPI datatypes. We implemented support for the naming of RMA windows and MPI communicators into Paradyn.

When a new resource, such as a process, communicator, or RMA window is discovered by Paradyn at run time, a new Paradyn resource object is created for it and a display representation of the resource is shown in the user interface. To support MPI-2 object naming, we implemented a way for the Paradyn daemon to communicate updated information about resources to the front end. This updated information could include a user-defined name for a resource or a notification that a resource has been deallocated. In the case that the new information is a user-defined name, the daemon sends an update report to the front end for that resource. The front end then updates the display of the resource hierarchy to reflect the name change. In the case that the resource has been deallocated, the Paradyn daemon sends an update report to the front end indicating that the resource is retired. This causes the resource to be "grayed out" in the display of the resource hierarchy and removes it from being a candidate bottleneck in the Performance Consultant's search.

5 Results

To test our enhanced version of Paradyn, we conducted a variety of tests for MPI-1 and MPI-2 features. We compared Paradyn's measurements against programs with known behavior, the findings of other performance tools, and benchmark programs.

We show diagrams that are a condensed form of the PC's findings. In them, only hypotheses that tested true for at least one of the MPI implementations are shown. In some cases, a mapping is made between what is shown in the diagram and what was displayed in the PC window. We indicate this with a †. For instance, instead of printing a particular machine's hostname, we would show 'node 1 †.'

Some results in this section report values of metric-focus pairs as measured by Paradyn. Performance measurements are collected by Paradyn in the following way. Paradyn keeps track of performance data in array of predefined size. Each element of the array is called a *bin* and contains performance data collected over an interval of time. In order to accommodate long running programs, Paradyn dynamically 'folds' the bins whenever the array becomes full. The values from two neighboring bins are combined, and the new bin represents twice the time period. In this way, half of the array storage is freed to collect more performance data. Over time, the granularity of the measurements decreases. The granularity of the bins starts out at 0.2 seconds. For our experiments, the granularity of the bins ranged from 0.2 to 0.8 seconds. Because of the combination of the bins over time , some amount of error is introduced into the performance data. To reduce error, we eliminated the first and last bins from the calculations. This is because we cannot know exactly when in the time interval represented by the end-point bins that the data collection actually began or ended.

5.1 MPI-1 Features

We tested our measurement of MPI-1 applications by comparing our results to those obtained with MPICH's MPE profiling libraries [34] and the gprof profiler [11]. For the comparison tests, we developed a test suite based on the Grindstone PVM test suite [16], called PPerfMark. For the tests we used LAM/MPI 7.0 with the sysv RPI and MPICH 1.2.5 with the ch_p4mpd device.

Program	Characteristics	Result	Details
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.	Pass	Paradyn showed the clients spending too much time in `MPI_Send`.
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.	Pass	Paradyn showed that the program spent most of its time sending and receiving messages. It also counted the number of bytes transferred.
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.	Pass	Paradyn identified that the program was spending too much time in send and receive operations.
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. The server wastes time before replying, simulating a busy server.	Pass	Paradyn found that the program was spending much time in `MPI_Barrier` because processes were late getting to the barrier. It found the program had a computational bottleneck.
Random-barrier	This program is like the intensive-server program except that no one 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`.	Pass	Paradyn correctly showed that the program was spending too much time in `MPI_Barrier`. Paradyn did not find a computational bottleneck unless the CPU threshold constant was lowered.
Diffuse-procedure	This program demonstrates a bottleneck that is distributed over the processes in the MPI application. The`bottleneckProcedure` consumes the majority of the time for the application. Each of the processes in the application take turns "being the bottleneck" while the others wait in `MPI_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.
System-time	This program spends most of its time executing in system calls.	Fail	Paradyn showed all hypotheses as false. Paradyn does not have default metrics specifically for system time.
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 `irrelevantProcedures` that use hardly any of the program's time.	Pass	Paradyn correctly found that the each process was CPUBound in the function `bottleneckProcedure`.

Table 2: PPerfMark MPI-1 Program Characteristics

5.1.1 PPerfMark

PPerfMark is based on the Grindstone Test Suite for Parallel Performance Tools [16]. We converted the tests from PVM to MPI. We used the PPerfMark programs that measure communication bottlenecks and computational bottlenecks. The test details are shown in Table 2.

We show a summary of the PPerfMark results in Tables 2 and 3. 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. The table clearly shows that Paradyn is able to find the synchronization and computation bottlenecks in LAM programs. The exception is the program system-time. The default Paradyn metrics do not include system time, thus Paradyn did not find bottlenecks in the program.

The following sections show our results for the MPI-1 PPerfMark programs.

5.1.2 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 3 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.

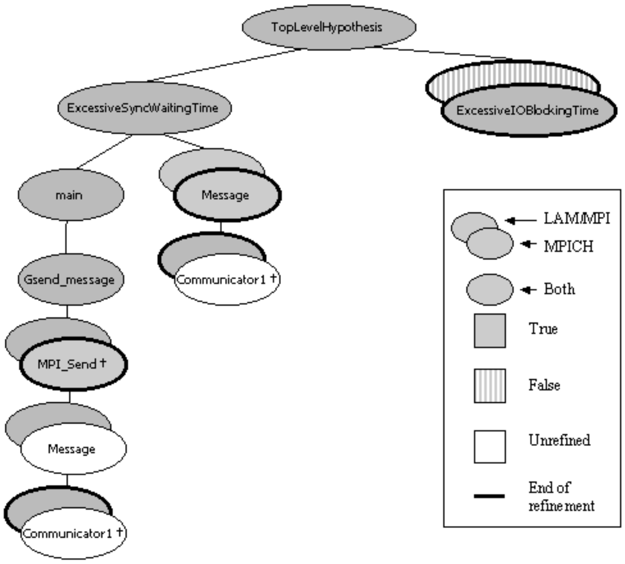
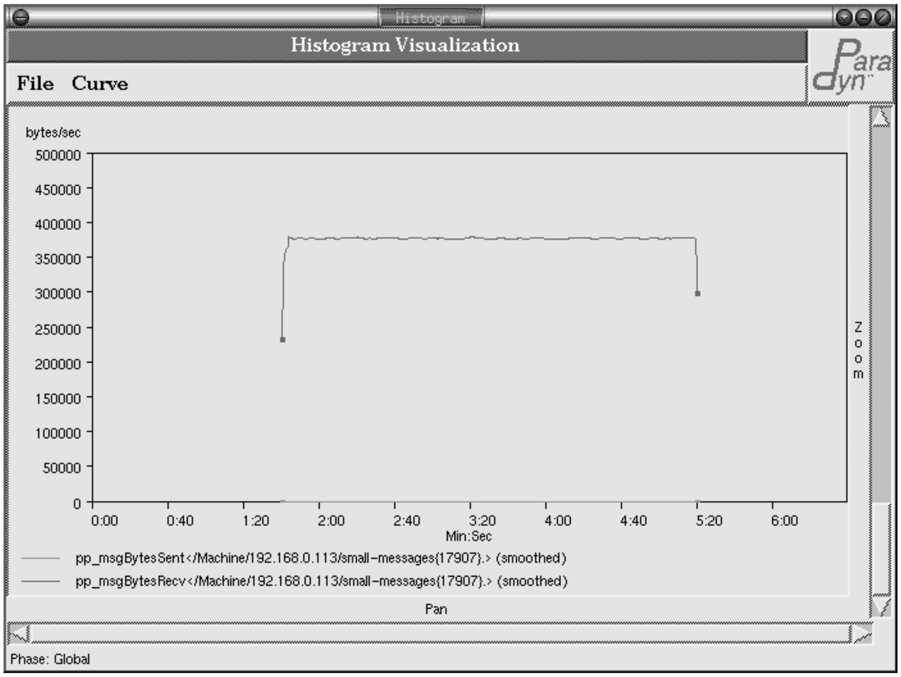


Figure 3: Paradyn PC Output for Small-Messages

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.



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

Phase: Global

Figure 4: 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.

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.

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 4 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 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. Some amount of error is expected in these values as the performance data bin granularity for these experiments was 0.8 seconds.

5.1.3 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 5, 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.

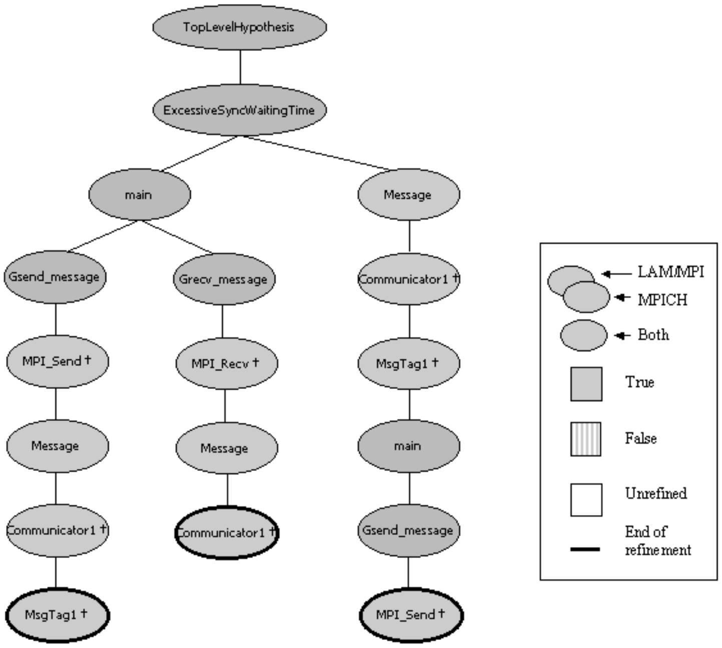


Figure 5: Paradyn PC Output for Big-Messages

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.

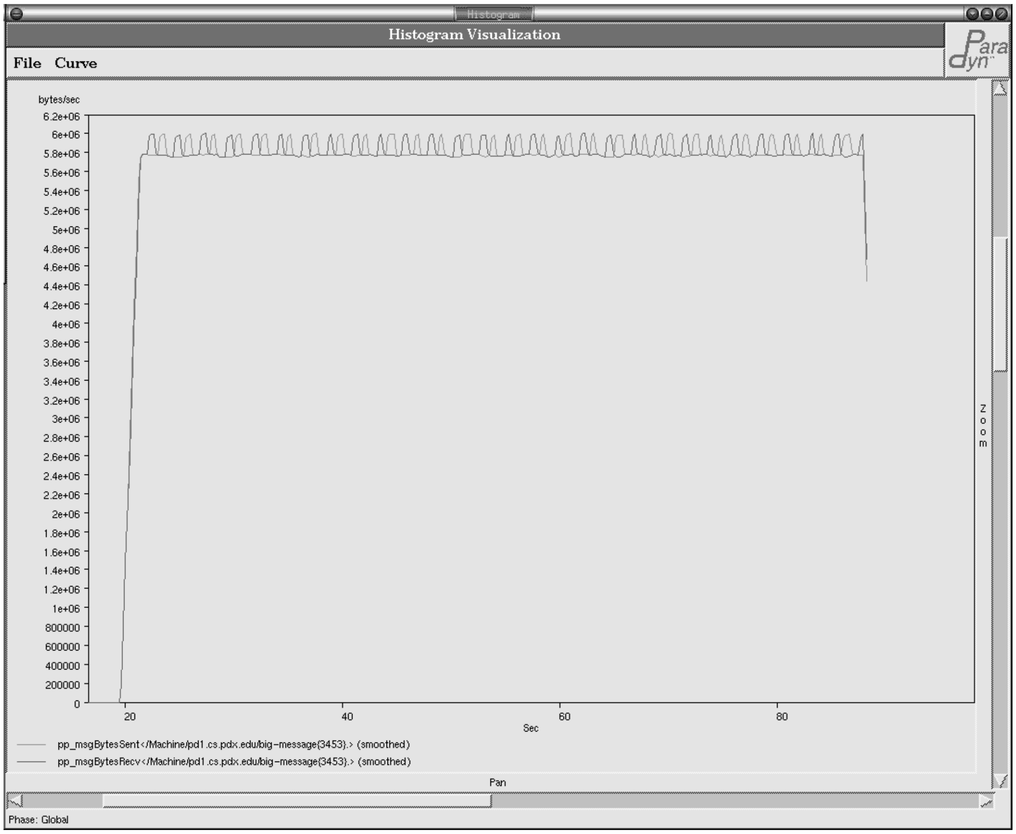


Figure 6: Paradyn Histogram Big-Message with LAM/MPI, Message Bytes Sent and Received
This figure shows the histogram that Paradyn generated for point-to-point message bytes sent and received for one process with LAM/MPI. We calculated the average bytes sent per second to be 5,800,820.4 and the average 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.

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 6, 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. 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. The performance data bin size for this experiment was 0.2 seconds.

5.1.4 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 7, 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.

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

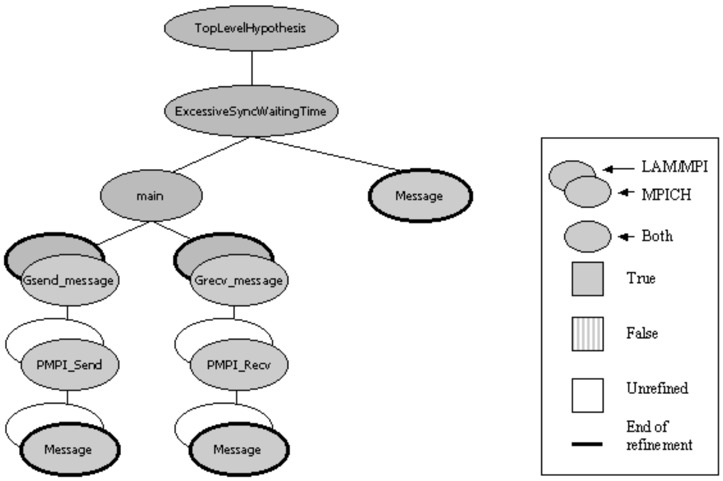


Figure 7: Paradyn PC Output for Wrong-Way

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`.

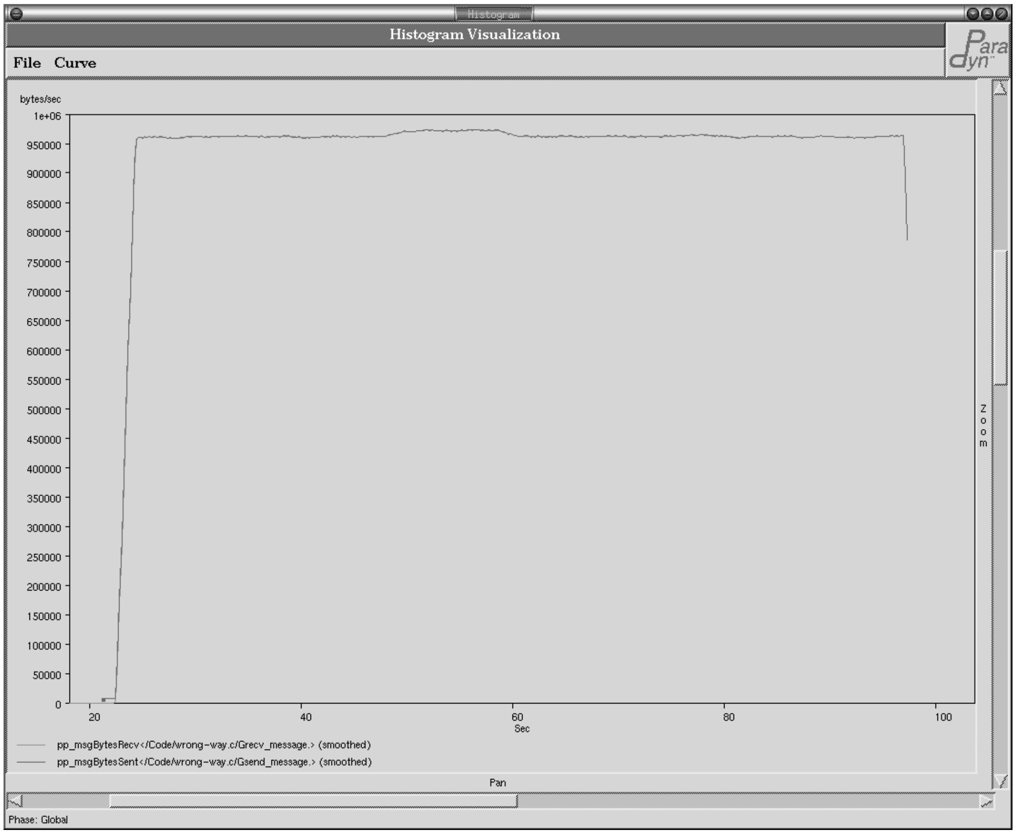


Figure 8: Paradyn Histogram Wrong-Way with LAM/MPI, Message Bytes Sent and Received

This is a histogram showing the bytes sent by process 0 and the bytes received by process 1. We performed calculations on the d ata that Paradyn generated and found that process 0 averaged sending 956,779.2 bytes per second and that process 1 received 944,582.7 byt es 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 b ytes received. Note: The colors in this screenshot were altered for printing purposes.

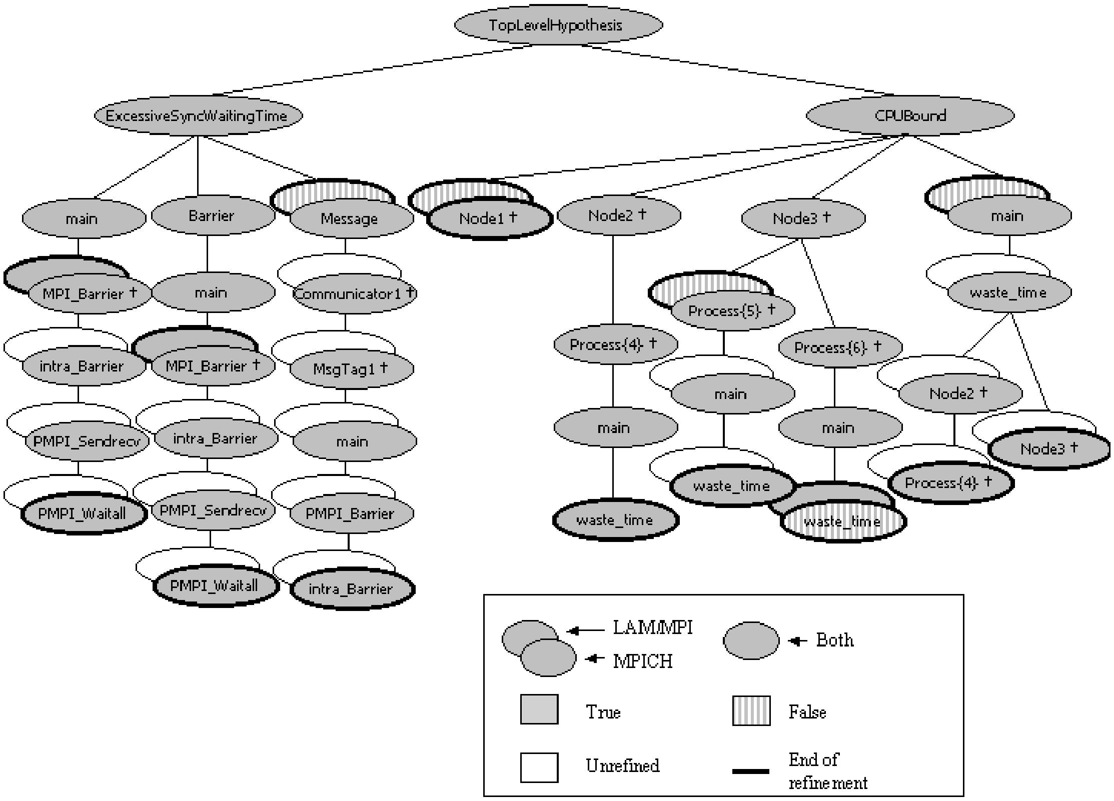


Figure 9: Paradyn PC Output for Random-Barrier

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.

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 8 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 71,375,728 bytes sent and 70,465,869 bytes received. The performance data bin size for this experiment was 0.2 seconds.

5.1.5 Random-barrier

Random-barrier was designed to simulate a program that has a load imbalance that moves to different proceses during execution. The parameters we used for the program runs were: 800 iterations and six processes, two each on three nodes. We set the compile-time constant TIMETOWASTE equal to 5, which specifies a relative amount of time that the bottleneck process will waste. Paradyn was able to correctly identify the bottlenecks for this program. Figure 9 shows a condensed form of the 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 `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.

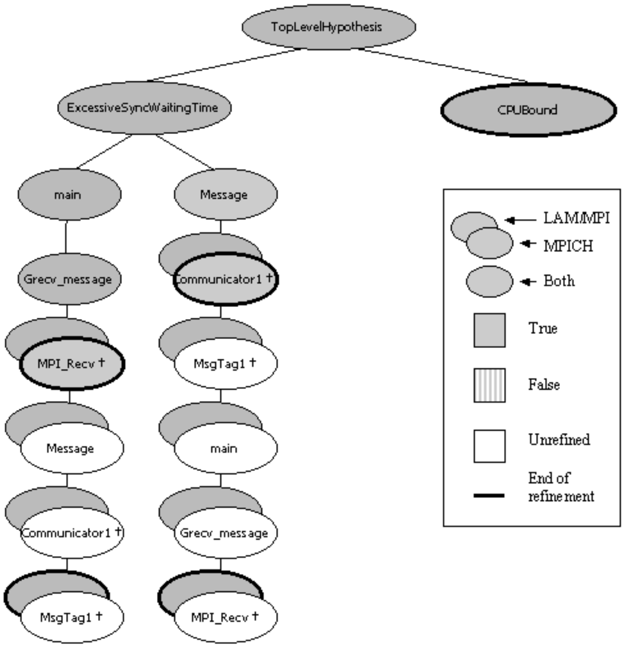


Figure 10: Paradyn PC Output for Intensive-Server

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.

We also used Paradyn to generate histograms of the synchronization time spent in these programs. Figure 18 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%.

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 17 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.

5.1.6 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 10 shows the condensed form of the Performance Consultant's findings for LAM/MPI and MPICH. We see that ExcessiveSyncWaitingTime is true and that the Performance 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 11 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

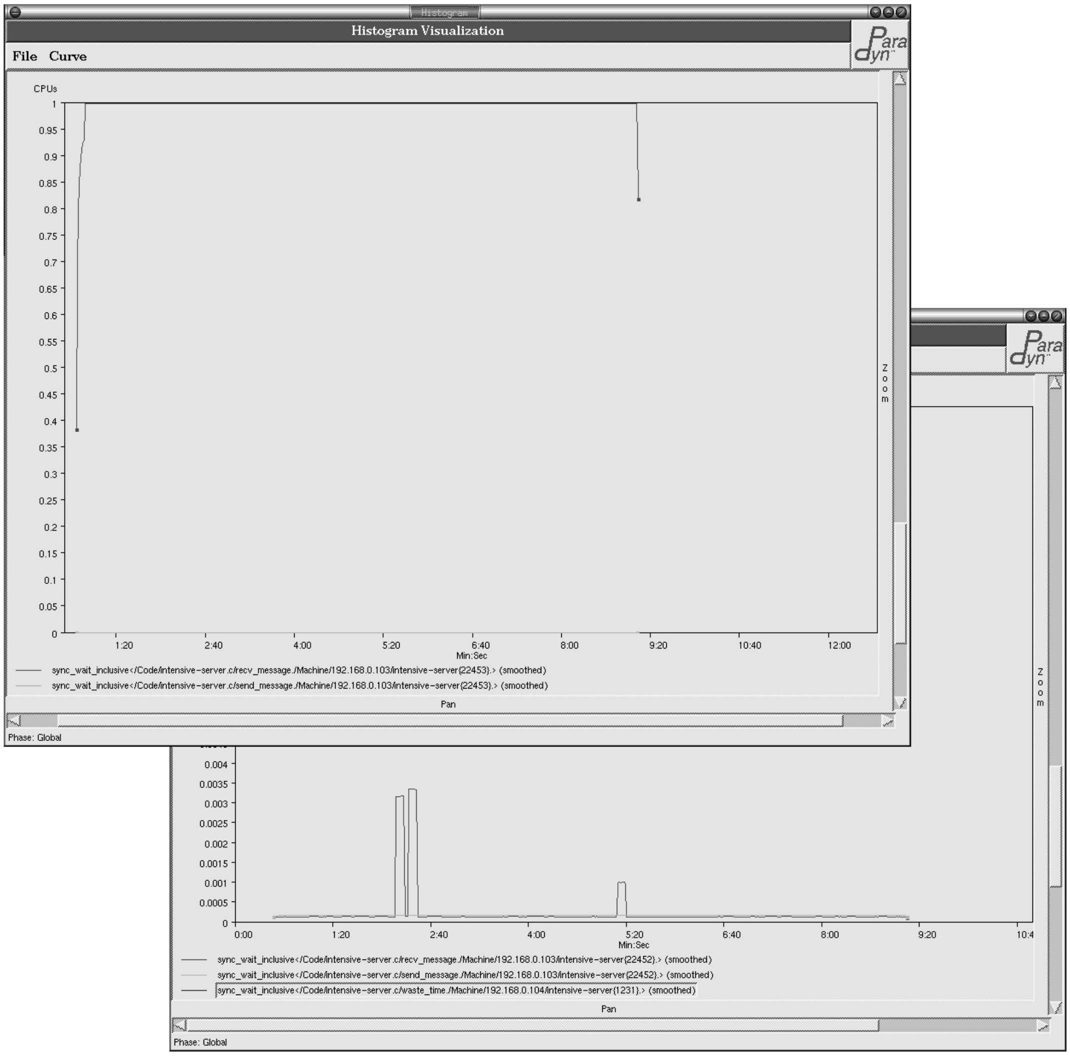
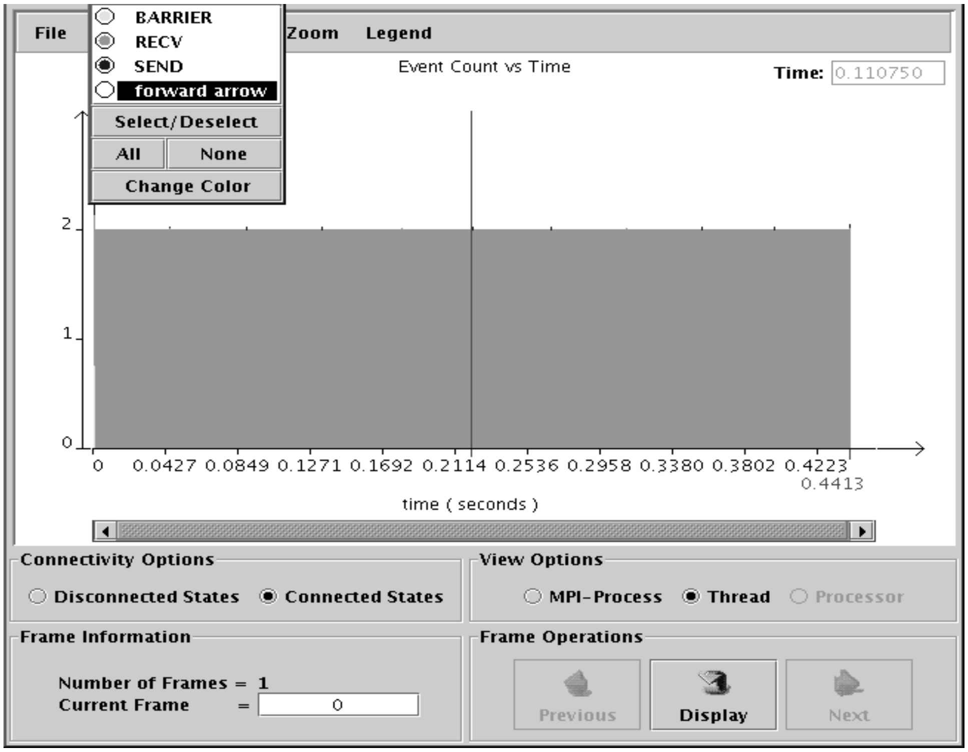

Histogram
Histogram Visualization
File
Curve
Paradyn
CPUs
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
1:20
2:40
4:00
5:20
6:40
8:00
9:20
10:40
12:00
Min:Sec
sync_wait_inclusive</Code/intensive-server.c/recv_message./Machine/192.168.0.103/intensive-server{22453}.> (smoothed)
sync_wait_inclusive</Code/intensive-server.c/send_message./Machine/192.168.0.103/intensive-server{22453}.> (smoothed)
Zoom
Pan
Phase: Global
Paradyn
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
0:00
1:20
2:40
4:00
5:20
6:40
8:00
9:20
10:4
Min:Sec
sync_wait_inclusive</Code/intensive-server.c/recv_message./Machine/192.168.0.103/intensive-server{22452}.> (smoothed)
sync_wait_inclusive</Code/intensive-server.c/send_message./Machine/192.168.0.103/intensive-server{22452}.> (smoothed)
sync_wait_inclusive</Code/intensive-server.c/waste_time./Machine/192.168.0.104/intensive-server{1231}.> (smoothed)
Zoom
Pan
Phase: Global

Figure 11: 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 inGsend_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.

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.

Figures 12 and 13 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 12 shows



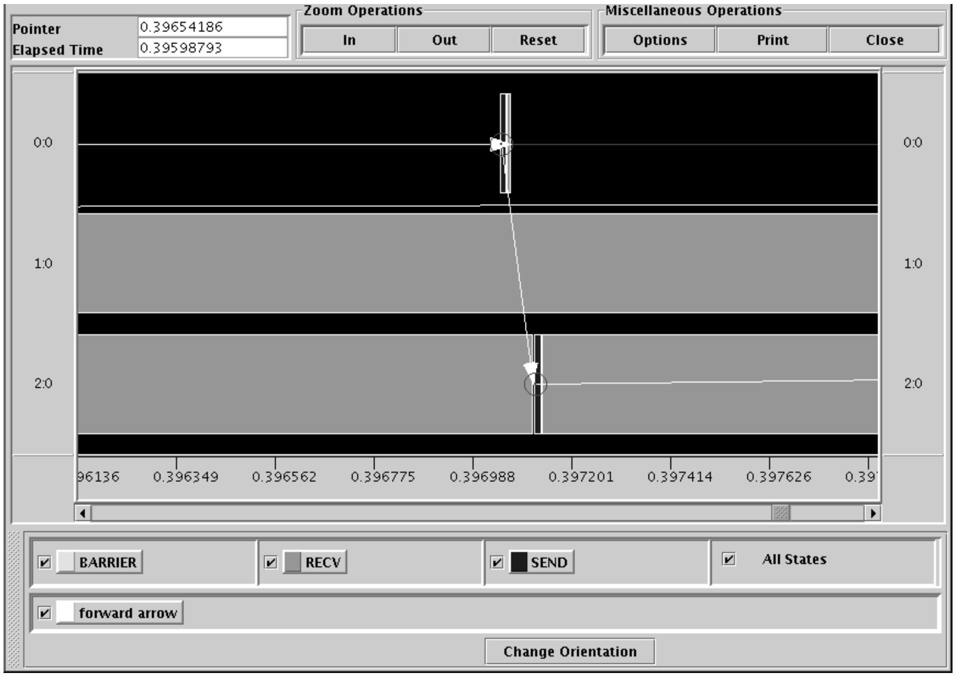
Time: 0.110750

○ Disconnected States ● Connected States

○ MPI-Process ● Thread ○ Processor

Number of Frames = 1
Current Frame = 0

Figure 12: 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`.



Pointer 0.39654186
Elapsed Time 0.39598793

☑ BARRIER ☑ RECV ☑ SEND ☑ All States
☑ forward arrow

Figure 13: 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`.

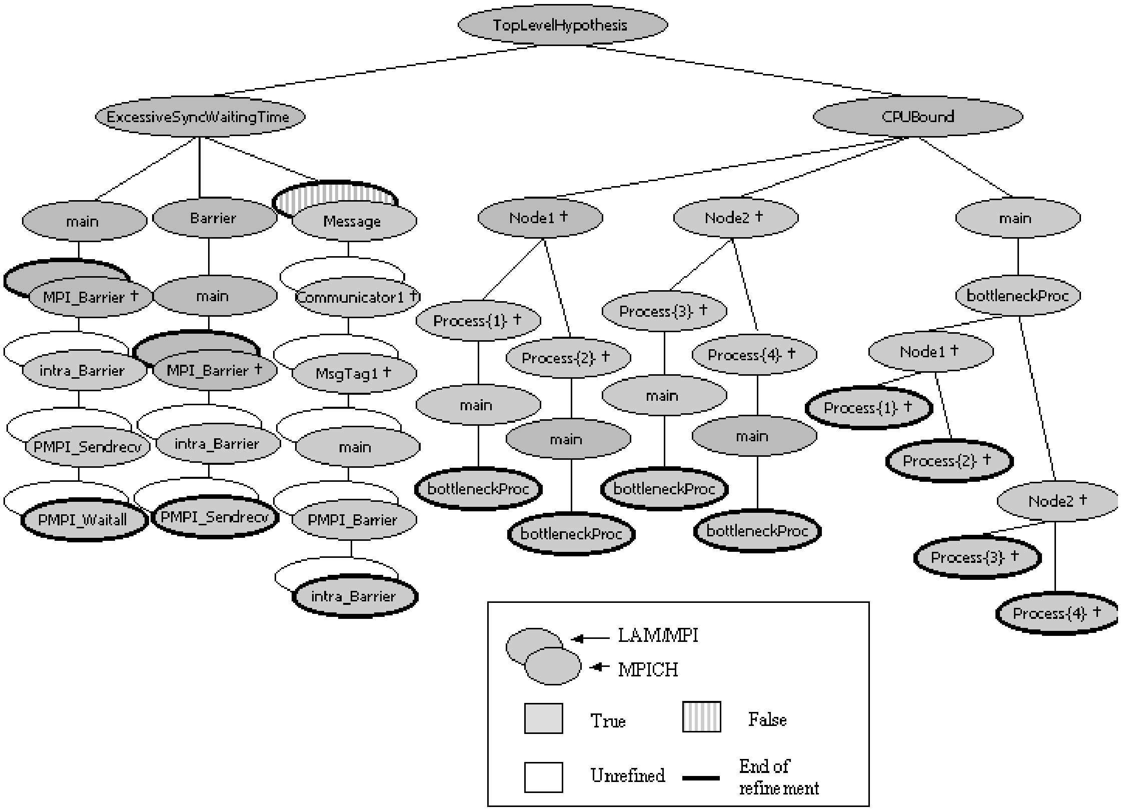


Figure 14: Paradyn PC Output for Diffuse-Procedure

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`.

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 13 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`.

5.1.7 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 14 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`.

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 15 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 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

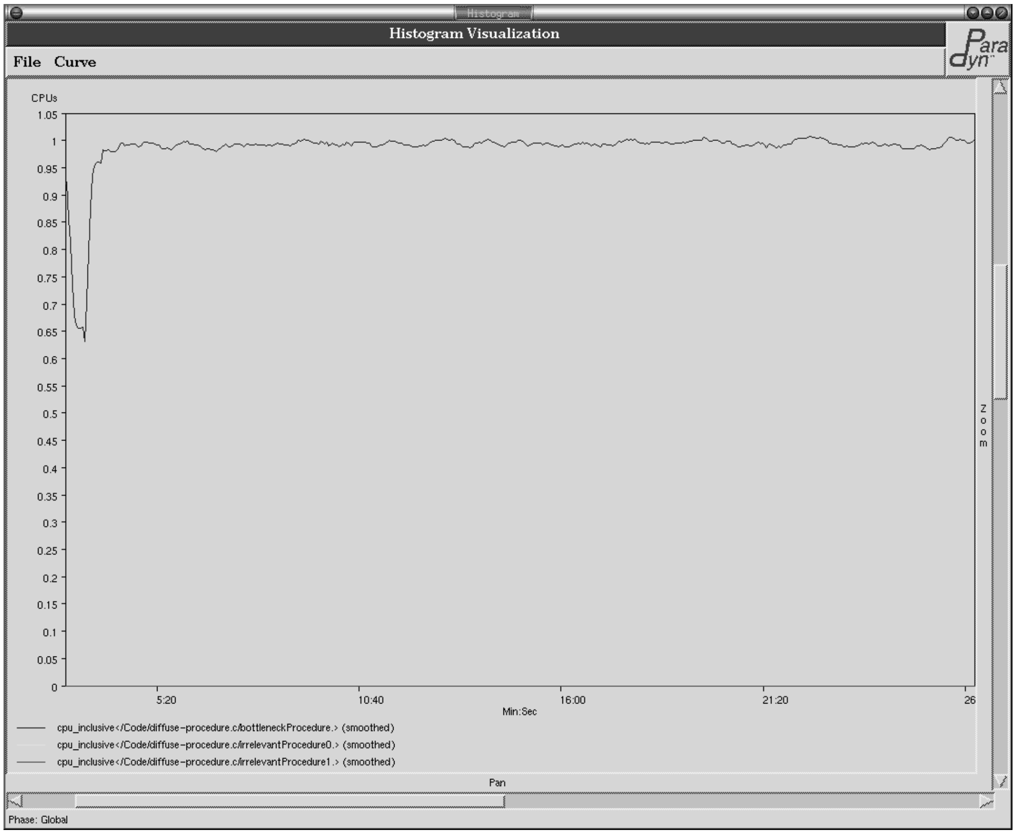

Histogram
Histogram Visualization
File
Curve
Paradyn
CPUs
1.05
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
5:20
10:40
16:00
21:20
26
Min:Sec
Zoom
cpu_inclusive</Code/diffuse-procedure.c/bottleneckProcedure.> (smoothed)
cpu_inclusive</Code/diffuse-procedure.c/irrelevantProcedure0.> (smoothed)
cpu_inclusive</Code/diffuse-procedure.c/irrelevantProcedure1.> (smoothed)
Pan
Phase: Global

Figure 15: 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 `irrelevant-Procedures`.

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.

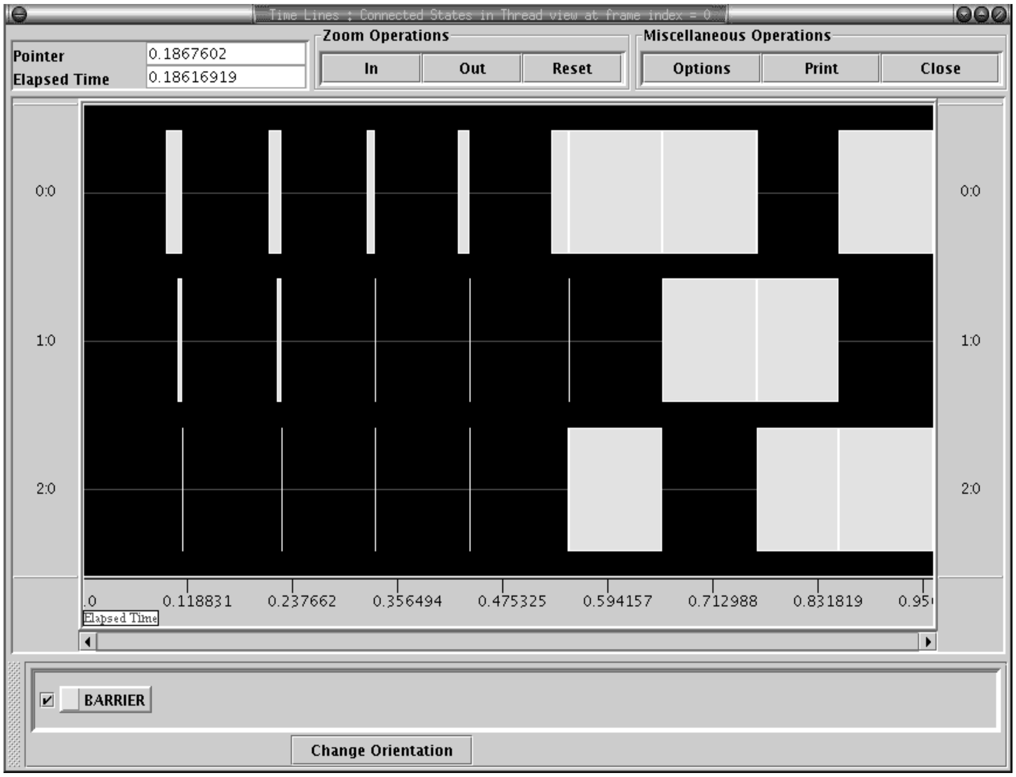
The last test for this program, in Figure 16, shows the Time Line window from Jumpshot-3 for a 10 iteration 3 process run of diffuse-procedure. We had to change the parameters for this run because the trace files got too large. Here Paradyn's synchronization findings are confirmed. The program is indeed spending much time in `MPI_Barrier`.

5.1.8 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.

5.1.9 Hot-Procedure

Hot-procedure is designed to simulate a program with a computational bottleneck in one procedure. The parameters we used for this program were: 1,000,000 iterations and four processes, two each on two nodes. Figure 20 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`.

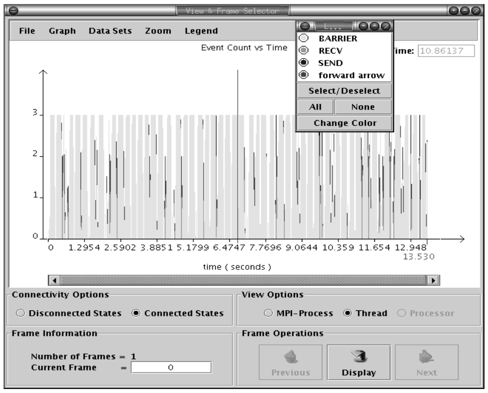


Pointer 0.1867602
Elapsed Time 0.18616919

☑ BARRIER

Figure 16: 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 thsame amount of time inMPI_Barrier, even though at a specific point in the program the distribution might not be balanced.



Time: 10.86137

○ BARRIER
● RECV
● SEND
● forward arrow

○ Disconnected States ● Connected States

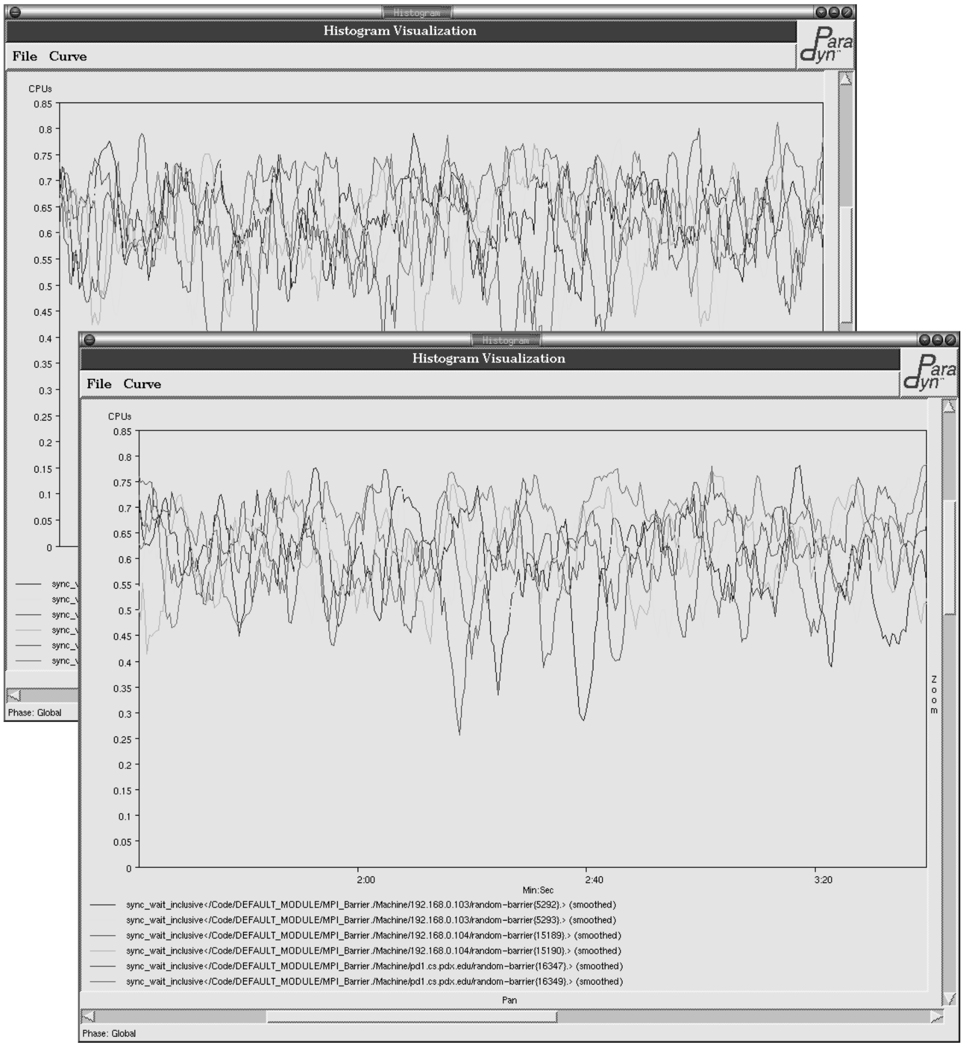
○ MPI-Process ● Thread ○ Processor

Number of Frames = 1
Current Frame = 0

Figure 17: 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.

To verify that Paradyn is correctly measuring the CPU time for the functions in this program, Figure 19 shows a portion of the output from the gprof profiler 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.



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)

Figure 18: 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 th e 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.

```
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  irrelevantProcedure
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 19: Gprof Analysis of Hot-Procedure

This is gprof output for the hot-procedure program. It shows that the`bottleneckProcedure` is indeed a computational bottleneck, consuming 100% of the program's total running time. The second and third columns of data confirms this, saying that all 46.19 seconds of running time were spent in the `bottleneckProcedure`. The fourth column shows us that each of the functions was called an equal number of times. However, the`irrelevantProcedures` took 0 microseconds for each call, while the `bottleneckProcedure` took 46190 µs.

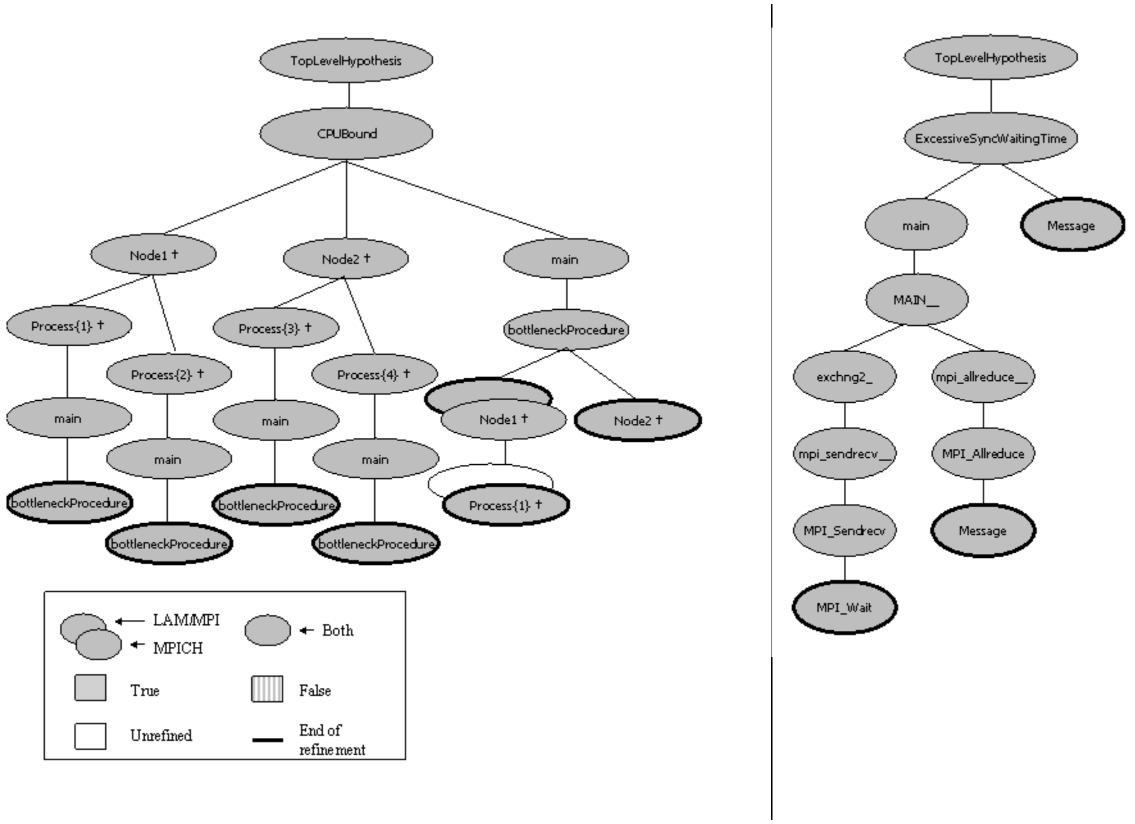


Figure 20: Paradyn PC Output for Hot-procedure and ssTwod

On the left, we show 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. The figure on the right 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

5.1.10 Sstwod

For our final MPI-1 test of Paradyn we use a toy program developed in Using MPI: Portable Parallel Programming with the Message-Passing Interface[13]. 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 20, we show the condensed 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 functionexchng2 to find MPI_Sendrecv to be a bottleneck. It also found a synchronization bottleneck inMPI_Allreduce.

5.1.11 MPI-1 Summary

Our testing shows that Paradyn correctly instruments and measures the performance of the MPI-1 features of LAM and MPICH for the majority of programs. We verified Paradyn's results 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 programs. Last, we used other performance tools and compared their results with those that Paradyn gave.

5.2 MPI-2

We tested our implementation of support for MPI-2 in Paradyn by measuring the performance of programs with known behavior and by comparing our results against benchmark programs. We designed MPI-2 programs for PPerfMark to test our implementation. Descriptions of these programs and test result details are shown in Table 3. In the following sections we give detailed test results for selected programs. For the MPI-2 tests we used LAM/MPI 7.0.4 with the sysv RPI and MPICH2 0.96p2 beta with the sock channel and mpd process manager.

Program	Description	Result	Details
allCount	This program uses a known number of Puts, Gets, and Accumulates to transfer a known amount of data to and from an RMA window.	Pass	Paradyn was able to count the number of RMA operations and the bytes that were transferred by them.
winCreate-Blast	This program creates and deallocates a large number of RMA windows very quickly.	Pass	Paradyn detected and incorporated all windows into the Resource Hierarchy.
winFence-Sync	This program uses`MPI_Win_fence` for synchronization. An artificial bottleneck is introduced in rank 0, which makes it late to the fence operation.	Pass	Paradyn showed that ranks except rank 0 were spending too much time in`MPI_Win_fence`. Rank 0 was found to be CPU bound in waste_time.
winScp-wSync	This is similar to winFenceSync, except that Start/Complete, Post/Wait synchronization is used.	Pass	Paradyn found that all ranks but rank 0 were spending too much time in synchronization. Rank 0 was found to be CPU bound.
spawnCount	This program spawns a known number of child processes. The child processes simply exit.	Pass	Paradyn was able to detect and incorporate all new processes into the Resource Hierarchy.
spawnSync	This program spawns children and then sends a known number of messages on an intracommunicator between the parent and child processes. An artificial bottleneck is introduced in the parent process.	Pass	Paradyn showed that the child processes were spending too much time in MPI_Recv and that the parent process was CPU bound. It was also counted the number of messages and the bytes transferred.
spawnWin-Sync	This program spawns child processes and then sets up an RMA window over an intracommunicator between the parent and child processes. There is an artificial bottleneck in the parent process.	Pass	Paradyn showed that the child processes were waiting in `MPI_Win_fence` and that the parent process was CPU bound. Paradyn counted the number of RMA operations and bytes transferred.

Table 3: PPerfMark MPI-2 Program Characteristics

5.2.1 RMA

We tested our implementation of support for RMA in Paradyn with several programs. First, we used programs that we developed for the PPerfMark for RMA. Second, we used the Performance Consultant to find the bottlenecks in a toy program Oned from Using MPI-2: Advanced Features of the Message-Passing Interface [14]. This is a similar program to the ssTwod program we used for MPI-1 testing, except that it uses RMA for communication. Last, we compared Paradyn's measurements of the RMA metrics against those gathered by the ASCI Purple Presta Stress Test Benchmark [1].

5.2.1.1 PPerfMark

The programs we developed for the PPerfMark for RMA are simple programs that allow us to easily know the behavior of the programs. In this way, we are able to say whether or not Paradyn was able to correctly measure the RMA performance metrics of the programs. The results of our tests are shown in Table 3. We implemented programs to test active target synchronization. We have not yet implemented the passive target test programs because neither LAM nor MPICH2 support passive target synchronization as of this writing.

Here we give details of the test results for the program winScpwSync, which uses the active target synchronization routines: `MPI_Win_start`, `MPI_Win_complete`, `MPI_Win_Post`, and `MPI_Win_wait`. An artificial bottleneck is inserted into the the process with rank 0 by having it call the function`waste_time` between its successive calls to `MPI_Win_wait` and `MPI_Win_post`. This causes the other ranks to wait in the synchronization routines `MPI_Win_start` or `MPI_Win_complete`. Paradyn was able to identify the bottlenecks. For both MPICH2 and LAM, it found that rank 0 was CPU bound in the function `waste_time`, which is what we expect since that is the artificial bottleneck. The other ranks were found to have excessive synchronization waiting time in active target synchronization routines: `MPI_Win_start` or `MPI_Win_complete`. The differences in the findings are due to differences in the MPI implementations. The MPI-2 Standard does not specify which of these routines will be blocking; it is up to the MPI implementor to decide what is most efficient. For both implementations, Paradyn discovered the RMA window upon which the excessive synchronization took place. Paradyn's findings are what we expect given the program's behavioral description.

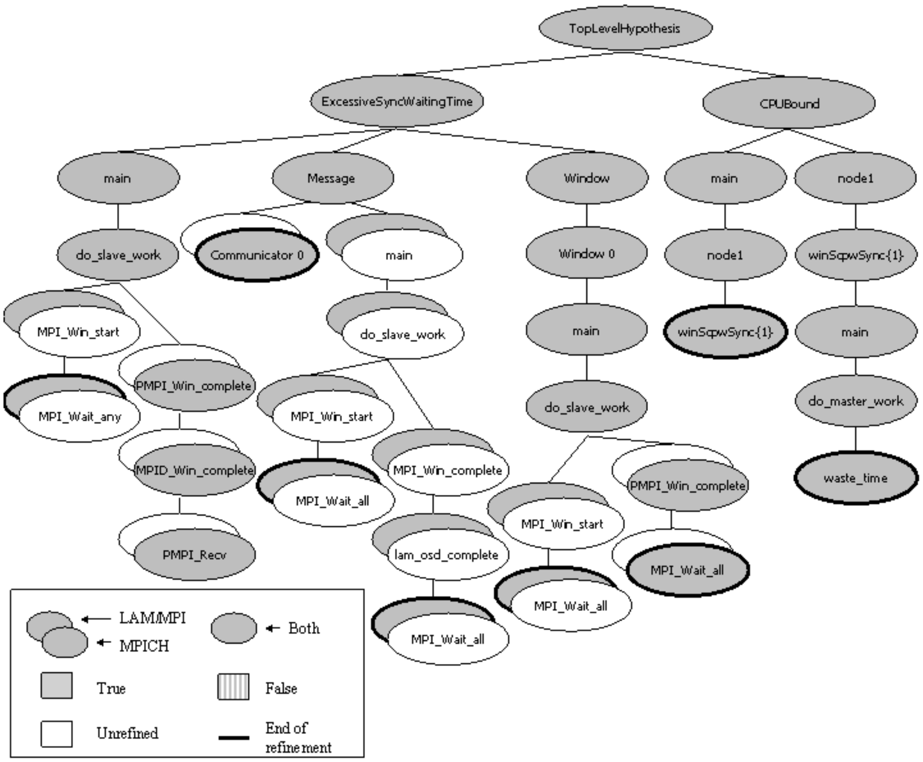


Figure 21: Performance Consultant Findings for winScpwSync for LAM and MPICH2

This figure shows the Performance Consultant's findings for the PPerfMark program winScpwSync. We see that the PC determined that ExcessiveSyncWaitingTime was true, found it to be due to active target synchronization on an RMA window, and determined the responsible RMA window. It also found that the program was CPU bound for the process with rank 0. These findings correspond to the program's behavioral description.

5.2.1.2 Oned

Here we show the results for running the Oned program with Paradyn's Performance Consultant. For both MPICH and LAM, the Performance Consultant discovered the bottleneck to be synchronization waiting time in `MPI_Win_fence` in the function `exchng1`. This is the known communication bottleneck of the program. An interesting difference between the results for the implementations are that LAM had a bottleneck in the synchronization object Barrier, because it implements `MPI_Win_fence` with a call to `MPI_Barrier`.

5.2.1.3 Presta

A test of our implementation of RMA support in Paradyn was to compare Paradyn's measurements of RMA metrics against those obtained by the Presta Benchmark program, rma [1]. Presta's rma measures the throughput of the `MPI_Put` and `MPI_Get` operations and the time per RMA operation for unidirectional `MPI_Put`, unidirectional `MPI_Get`, bidirectional `MPI_Put`, and bidirectional `MPI_Get`. For our testing, we used command line arguments to rma that specified two MPI processes would do remote memory operations of 1024 bytes with 3000 operations per epoch and 200 epochs. In Paradyn, we selected each process as a focus for the metrics: rma_put_ops, rma_get_ops, rma_put_bytes, and rma_get_bytes and generated Paradyn histograms for each process.

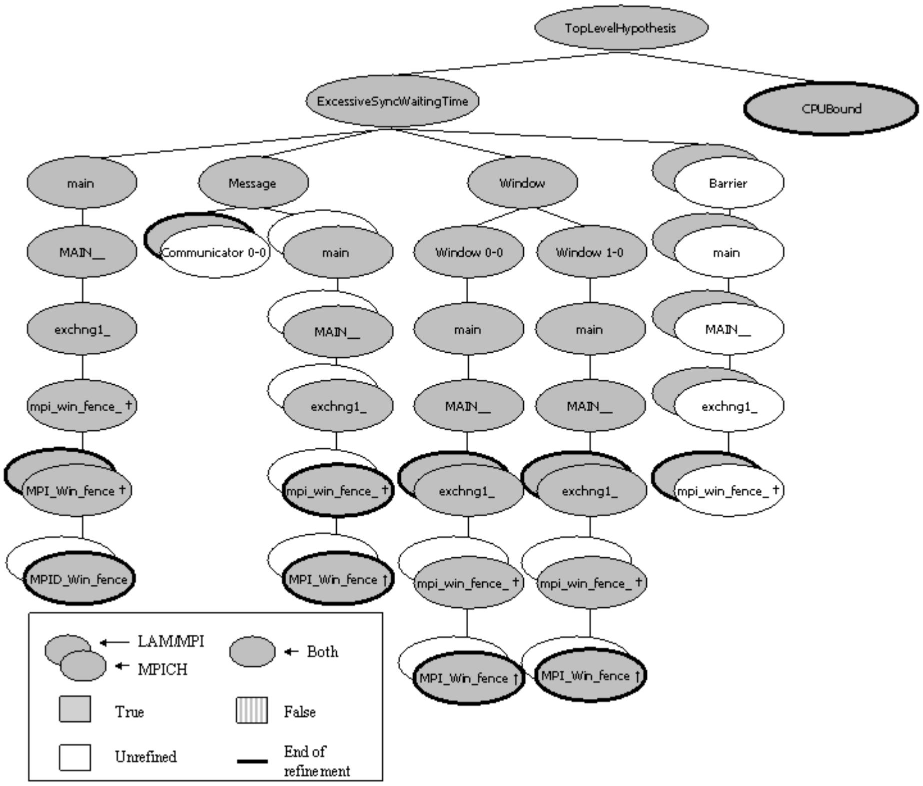


Figure 22: Paradyn Performance Consultant Findings for Oned

This figure shows the Performance Consultant's findings for the Oned program for LAM and MPICH. It discovered the bottleneck to be `MPI_Win_fence` in the function `exchng1`, which is the known communication bottleneck of the program.

For these experiments, the granularity of the performance data bins was 0.4 seconds. To obtain comparable data from Paradyn's output, we multiplied the values in each bin by the amount of time each bin represented and summed these values to get the total number of operations or bytes transferred. The throughput was calculated by dividing the total bytes by an estimate of the running time of the operation. The running time for a particular operation was estimated by counting the number of bins with data for that operation and multiplying that by the amount of time each bin represented. To reduce error in our estimate of time, we eliminated the first and last bins from the calculations. This is because we cannot know exactly when in the time interval represented by the end-point bins that the data transfer actually began or ended. Per operation time was calculated by taking the estimated running time and dividing it by the number of operations.

The data collected from the operation count metrics, rma_put_ops and rma_get_ops, were directly comparable with expected results, derived from knowing the input parameters to the rma program and source code inspection. We determined whether differences in the measurements were statistically significant by inspecting the confidence interval of the mean of the differences of the two sets of measurements. We found that the differences between the operation counts for Presta and Paradyn were not statistically significant, except for the bidirectional Get test. We are currently investigating the reason for this difference.

With the data collected from the byte and operation counting metrics we were able to calculate throughput and per operation time to compare with the measurements reported by the rma program. For LAM, the differences between the throughput and per operation time as calculated by Presta and Paradyn were not statistically significant. For MPICH2, the differences for the most part were not statistically significant. The exceptions were per operation time for unidirectional put and throughput for unidirectional get. In both cases, however, the relative difference in measurements was small, approximately 0.6%.

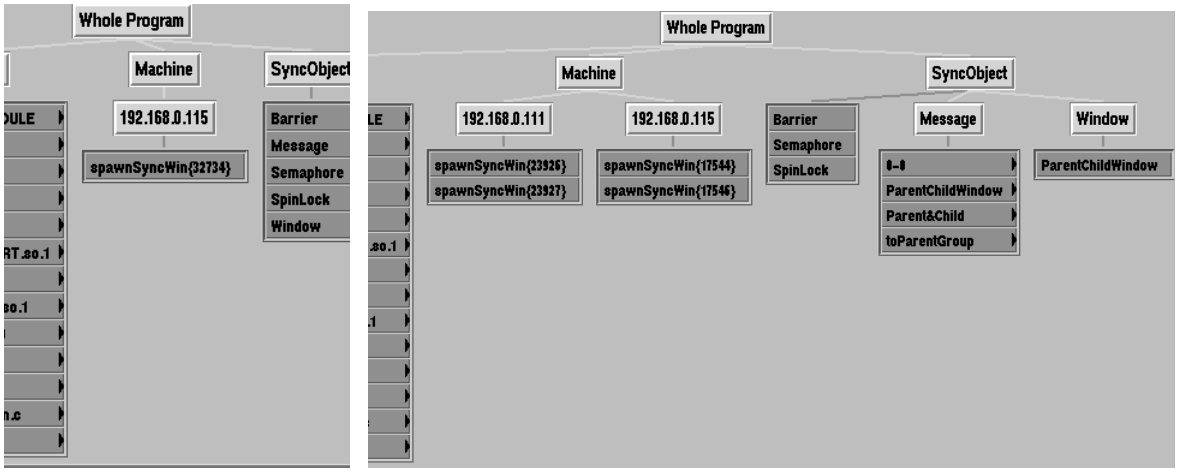


Figure 23: Paradyn's Resource Hierarchy Before and After a Spawn Operation
The figure shows Paradyn's Resource Hierarchy for the spawnSyncWin program. The screenshot on the left shows the Resource Hierarchy before `MPI_Comm_spawn` was executed. The screenshot on the right shows the Resource Hierarchy after the spawn. Note that three new processes have been added and the RMA Window which the child and parent groups of processes created has been detected. We also see that the friendly names that were given to communicators and RMA windows appear here. The reason that ParentChildWindow appears under Message with the other communicators is that LAM's MPI_Win structure contains a communicator that is created when the window is created. LAM stores RMA window names in the communicator structure. The Parent&Child is an intracommunicator created between the parent and child processes. The toParentGroup communicator is the intercommunicator given to the child processes by `MPI_Comm_get_parent`.

5.2.2 Dynamic Process Creation

We tested our implementation of support for dynamic process creation with the programs we developed for the PPerfMark. Our current method of support using the PMPI interface is successful in detecting spawn operations and incorporating the new processes into Paradyn's Resource Hierarchy. Paradyn is able to measure the performance of the new processes. We give evidence of these statements here. For these tests, we use only the LAM implementation of MPI. This is because MPICH2 0.96p2 beta does not yet fully support dynamic process creation.

Figure 23 shows screenshots of Paradyn's display of the Resource Hierarchy before and after `MPI_Comm_spawn` executes in spawnSyncWin. The left figure is the Resource Hierarchy before the spawn operation. The figure on the right is the Resource Hierarchy after the spawn. We see that Paradyn successfully detected and incorported the newly spawned processes. It also detected the RMA window upon which the parent and child groups of processes transfer information.

We used the programs spawnSync and spawnSyncWin to test Paradyn's performance measurement of the child processes. In Figure 24, we show the Performance Consultant's findings for these programs. The PC was able to determine the correctly program's bottlenecks in accordance with the programs' behavioral descriptions. The diagram on the left shows spawnSync, which is a program that passes messages between the parent and children, with an artificial computational bottleneck in the parent. We see that there was excessive synchronization waiting time for the child processes, due to message passing in the function `childFunction`. We also see that the parent was found to be CPU bound in the function `parentFunction`. The diagram on the right shows the PC's analysis of spawnSyncWin. The child processes were found to have excessive synchronization waiting time due to one-sided communication on the window named ParentChildWin in `MPI_Win_fence`. There was also excessive synchronization time due to message passing, which is because LAM uses `MPI_Isend/MPI_Waitall` in its implementation of `MPI_Win_fence`. The program was found to be CPU bound in the parent process in `parentFunction`.

5.2.3 Naming of MPI Objects

We tested our implementation of support for the naming of MPI communicators and RMA windows while we tested other functionality. We found it helpful to set names for these objects so that we could easily choose the appropriate synchronization object in the resource hierarchy for focus selection. It also facilitated our understanding of the

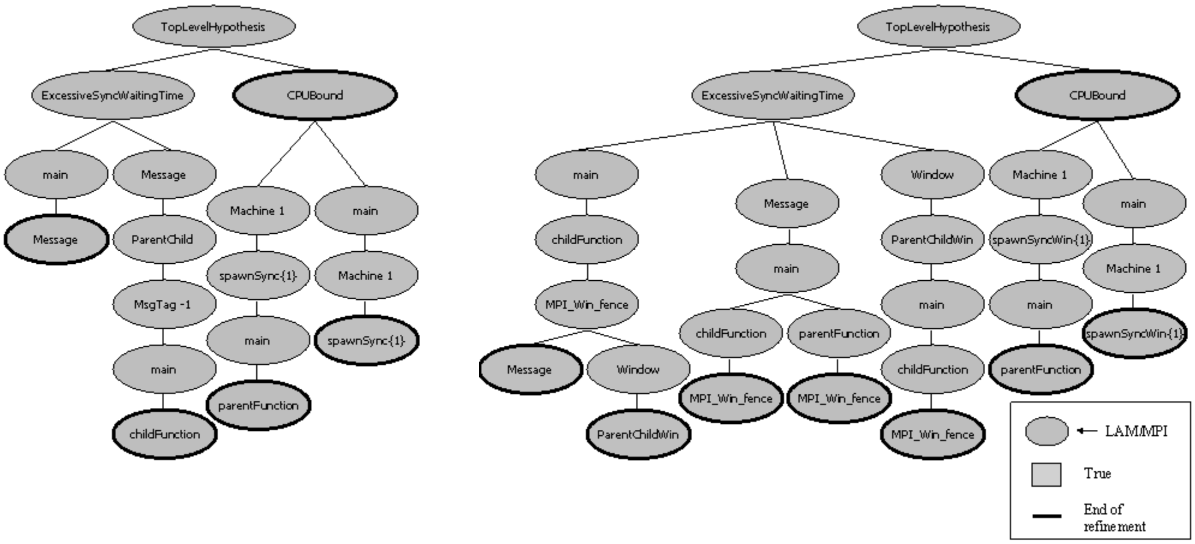


Figure 24: Performance Consultant Findings for spawnSync and spawnSyncWin and LAM
The diagram on the left shows the Performance Consultant's findings for the spawnSync program. The PC was able to determine that the hypothesis ExcessiveSyncWaitingTime was true, and drilled down to find the synchronization bottleneck was due to message passing in the `childFunction`. The PC also found that CPUBound was true and that it occurred in the parent process in the function parentFunction. The diagram on the right shows the PC's findings for spawnSyncWin. We see that the ExcessiveSyncWaitingTime is true and is due to message passing and one-sided communication. The PC determined the RMA window upon which the excessive synchronization took place, parentChildWin. Note that the friendly name of the RMA window was detected by Paradyn and displayed in the PC window. It also found that CPUBound was true in the parent process in the function `parentFunction`.

Performance Consultant's findings. The right screenshot of Figure 23 shows the resource hierarchy displaying some names that we gave to MPI communicators and windows.

6 Conclusions

We have described our work enhancing the Paradyn performance tool to provide support for MPICH and LAM/MPI on Linux clusters. The additional functionality we have developed includes support for non-shared file systems, the LAM implementation, RMA, dynamic process creation, and MPI object naming. We have presented results that show the effectiveness of this new performance tool, including small applications, a benchmark program, plus a performance test suite we have developed, called PPerfMark.

We are continuing to implement support for the remaining MPI-2 features and to develop additional programs for the PPerfMark test suite. We are also performing a case study using our enhanced Paradyn to characterize performance changes in an atmospheric modeling program when MPI-1 communication is replaced with MPI-2 one-sided data transfer routines. We are working with the Paradyn group to incorporate the changes made in this project into the next release of Paradyn.

7 Acknowledgments

We would especially like to thank Phil Roth for his assistance with Paradyn. We are grateful for the expertise and guidance of David McClure and Bryant York. We also thank Charlie Schluting for keeping Wyeast running during our testing, and the entire Paradyn group for their support.

8 References

[1] ASCI Purple Presta Stress Test Benchmark v 1.2. Available at: http://www.llnl.gov/asci/purple/benchmarks/limited/presta/. Apr. 25, 2002.

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

[3] R. Braby, J. Garlick, and R. 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.

[4] G. Burns, R. Daoud, and J.Vaigl. “LAM: An Open Cluster Environment for MPI.” Proceedings of Supercomputing Symposium. pp. 379-386. 1994.

[5] S. Chakravarthi, K. Kumar, A. Skjellum, B. Seshadri, H. Prahalad “A Model for Performance Analysis of MPI Applications on Terascale Systems.” PVM/MPI 2003: 81-87. Venice, Italy. Sept. 29 - Oct. 2, 2003.

[6] J. Cownie and W. Gropp. “A Standard Interface for Debugger Access to Message Queue Information in MPI.” Proceedings of PVMMPI’99, pp. 51-58. June 25, 1999.

[7] L. DeRose, K. Ekanadham, J. Hollingsworth, and S. Sbaraglia. “SIGMA: A Simulator Infrastructure to Guide Memory Analysis.” Supercomputing 2002. Baltimore, MD. Nov. 16-22, 2002.

[8] L. DeRose. “The Dynamic Probe Class Library: An Infrastructure for Developing Instrumentation for Performance Tools.” International Parallel and Distributed Processing Symposium. Apr. 2001.

[9] J. Garlick and C. 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.

[10] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. MPI- The Complete Reference: Volume 2, The MPI Extensions. MIT Press. 1998.

[11] S. Graham, P. Kessler, and M. McKusick. “gprof: a Call Graph Execution Profiler.” Proceedings of the 1982 SIGPLAN symposium on Compiler construction. Boston, Massachusetts. pp. 120-126. 1982.

[12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. “A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard.” Parallel Computing. North-Holland. vol. 22. pp. 789-828. 1996.

[13] W. Gropp, E. Lusk, and A.Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. Second Edition. MIT Press. 1999.

[14] W. Gropp, E. Lusk, and R. Thakur. Using MPI-2: Advanced Features of the Message-Passing Interface. MIT Press. 1999.

[15] J. Hollingsworth, B. Miller, and J. Cargille. “Dynamic Program Instrumentation for Scalable Performance Tools.” Proceedings of SHPCC’94. Knoxville, TN. pp. 841-850. May 1994.

[16] J. Hollingsworth and M. Steel. “Grindstone: A Test Suite for Parallel Performance Tools.” University of Maryland Computer Science Technical Report. CS-TR-3703. Oct., 1996.

[17] B. Mohr, D. Brown, and A. Malony. “TAU: A portable parallel program analysis environment for pC++.” In Proceedings of CONPAR 94 - VAPPVI. University of Linz, Austria. Sept. 1994.

[18] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, B. Irvin, K. Karavanic, K. Kunchithapadam and T. Newhall. “The Paradyn Performance Tools.” IEEE Computer 28(11):37-46. Nov.1995.

[19] K. Mohror. “Infrastructure For Performance Tuning MPI Applications.” Computer Science Master’s Thesis. Portland State University. Portland, OR. Available at: http://www.cs.pdx.edu/~karavan/pperfdb.html. Jan. 2004.

[20] P. Mucci. “Dynaprof 0.8 User’s Guide.” Available at: http://www.cs.utk.edu/~mucci/dynaprof/dynaprof.html. Nov. 2002.

[21] W. Nagel, A. Arnold, M.Weber, H. Hoppe, and K. Solchenbach. “VAMPIR: Visualization and analysis of MPI resources.” Supercomputer, 12(1):69-80. Jan. 1996.

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

[23] Parallel Environment for AIX 5L V4.1: Operation and Use, Vol 2. IBM Document SA22-7949-00. Available at: http://publib.boulder.ibm.com/clresctr/docs/pe/pe_41/ 200310/am103000/am10300002.html. Apr. 2004.

[24] W. Putman, J. Chern, S. Lin, W. Sawyer, and B. Shen. “Modeling the Earth’s Atmosphere.” NASA Goddard Space Flight Center. Presented at SC2002. Information available at: http://www.nas.nasa.gov/About/Media/SC02 /Goddard/goddard.html. Nov. 16-22, 2002.

[25] V. Pillet, J. Labarta, T. Cortes, and S. Girona. “Paraver: A Tool to Visualize and Analyze Parallel Code.” Technical report UPC-CEPBA 95-3. Available at: http://www.ac.upc.es/recerca/reports. 1995.

[26] D. Reed, R. Aydt, R. Noe, P. Roth, K. Shields, B. Schwartz, and L.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.

[27] R. Ribler, J. Vetter, H. Simitci, and D. Reed. "Autopilot: Adaptive Control of Distributed Applications." Proceedings of th e 7th IEEE Symposium on High-Performance Distributed Computing. Chicago, IL. July 1998.

[28] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. MPI - The Complete Reference. Volume 1, The MPI Core . 2nd Ed. MIT Press. 1999.

[29] TeraGrid Project. “Distributed Terascale Facility to Commence with $53 Million NSF Award.” TeraGrid News Release. Available at: http://www.teragrid.org/news/news01/080901_nsf.html. Aug. 9, 2001.

[30] Top 500 Supercomputer Sites. Available at: http://www.top500.org. June 2003.

[31] V. Taylor, X. Wu, and R. Stevens. “Prophesy: An Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications.” ACM SIGMETRICS Performance Evaluation Review 30(4):13-18. Mar. 2003.

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

[33] F. Wolf and B. Mohr. "Automatic Performance Analysis of MPI Applications Based on Event Traces." Proceedings of European Conference on Parallel Computing (Euro-Par). Munich, Germany. Aug. 2000.

[34] O. Zaki, E. Lusk, W. Gropp, and D. Swider. "Toward scalable performance visualization with Jumpshot." High Performance Computing Applications. 13(2):277-288. Fall 1999.