The PVM 3.4 Tracing Facility and XPVM 1.1

James Arthur Kohl (kohl@msr.epm.ornl.gov)
G. A. Geist (geist@msr.epm.ornl.gov)
Computer Science & Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6367
Phone: (615) 574-3143
FAX: (615) 574-0680 *

Abstract

One of the more bothersome aspects of developing a parallel program is that of monitoring the behavior of the program for debugging and performance tuning. Often there is no intrinsic support for examining program state and dynamics in a parallel programming system, let alone a useful interface for analyzing or visualizing that information. This paper discusses an enhanced tracing facility and tracing tool for PVM (Parallel Virtual Machine), a message passing library for parallel processing in a heterogeneous environment. PVM supports mixed collections of workstation clusters, shared-memory multiprocessors, and MPPs. The latest release of PVM, Version 3.4, contains a new and improved tracing facility which provides more flexible and efficient access to run-time program information. This new tracing system supports a buffering mechanism to reduce intrusion, and a more flexible trace event definition scheme which is based more directly on SDDF (Self-Defining Data Format, Reed et al). This new scheme expedites the collection and analysis of execution histories, and allows for integration of custom trace events. The tracing instrumentation is built into the PVM library to avoid re-compilation, and supports on-the-fly adjustments to each task's trace event mask, to control the level of tracing detail. Along with this new tracing facility, the graphical console and monitor XPVM has been updated to provide better access to the new tracing functionality. Several new views have been implemented to utilize the additional tracing information now possible, including user-defined events. The system has also been optimized to provide better real-time monitoring capabilities.

*Research supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy.

1 Introduction

A wide variety of scientists and applications programmers are moving into the arena of parallel processing to solve ever larger problems in ever shorter time. Perhaps the greatest challenges to developing good parallel programs are successful debugging and performance tuning. The central problem is one of observability – it is difficult to determine the precise behavior of a set of concurrently executing tasks. Heisenberg uncertainty tells us that we simply cannot examine the behavior of such a system without perturbing it. Therefore, the goal of effective parallel program analysis is to provide information that is as accurate as possible without significantly intruding on the parallel program.

This paper focuses on the tracing facility provided in the latest release of PVM (Parallel Virtual Machine) [3]. PVM is a message passing library which supports portable, interoperable parallel processing in a heterogeneous environment. A single "virtual machine" can consist of clusters of workstations, shared-memory multiprocessors, and Massively Parallel Processors (MPPs). The tracing facility described here supports the retrieval of program state information during run-time, in the form of events which can be viewed either in "real-time" or used in "post-mortem" analyses.

A related tool XPVM [6] is a graphical console and trace monitor that provides a graphical interface for visualizing and processing PVM execution traces. XPVM provides access to the PVM console commands for manipulating the virtual machine and spawning parallel tasks, and also provides several animated views which monitor the status of hosts and the execution of tasks with communication. These views provide information about the interactions among tasks in a parallel program, to assist in debugging and performance tuning.

The most recent release of PVM, version 3.4, contains an enhanced tracing facility which provides better flexibility and efficiency in gathering traces. The new facility supports *buffered* tracing, such that a large number of trace events can be combined into a single message for transferral to a trace replay tool, e.g. XPVM. Buffering trace events reduces the intrusion to the parallel application, thereby improving the accuracy of the resulting traces. A new trace event definition scheme has also been implemented to increase flexibility and expedite the processing of traces. The new scheme provides the capability for user-defined "custom" trace events.

To better support the processing and analysis of traces generated using the new 3.4 facility, a new release of XPVM, version 1.1, is now available. XPVM 1.1 has been updated to include several new view features, and has been significantly optimized to improve its real-time monitoring and trace playback capabilities.

The remainder of this paper discusses the new PVM 3.4 tracing facility in more detail. Section 2 overviews the old PVM 3.3 tracing support for comparison. Sections 3 and 4 describe the design of the new tracing facility and the new features in XPVM 1.1, respectively. Section 5 presents some experimental results comparing the new systems to the original PVM 3.3 and XPVM 1.0 releases. Section 6 proposes areas for future work.

2 Background

The first tracing facility for PVM was implemented in version 3.3. This tracing facility supported built-in tracing instrumentation. By integrating the instrumentation for tracing directly into the PVM library, the user application did not have to be re-compiled to affect changes in the tracing output. The destination for trace event collection and the PVM routines to be traced could all be selected at run-time. The new PVM 3.4 tracing facility maintains this integrated functionality.

Each trace event captured in PVM corresponds to either the entry or exit point of a PVM library routine, and contains information about any calling parameters or returned results. The trace events are transferred to a tracing tool such as XPVM using the regular PVM message channels. XPVM receives and processes these trace event messages to produce a "trace file" which can then be read in to drive graphical or textual views. The trace files for both tracing facilities are written in SDDF (Self-Defining Data Format) [1] as designed for the "Pablo" project [4]. SDDF was chosen over the PICL format [5] for the "ParaGraph" system [2] because PICL was not sufficiently flexible to represent the trace information generated by PVM. SDDF allows arbitrary data structuring and also includes some semantic information.

The original tracing facility provided with PVM 3.3 did not support any buffering mechanism, resulting in high tracing overhead. For example, each message sent between PVM tasks generated four trace event messages to XPVM, two for the send and two for the receive. This deluge of trace messages severely perturbed communication-intensive applications, slowing down their execution and making any accurate performance evaluation impossible.

Another problem with the original 3.3 tracing facility was that the contents of trace events were hard-wired in the PVM library. This made proper unpacking of trace event messages difficult for tools like XPVM. It was necessary to manually code the unpacking of several trace events, opening many opportunities for error. In addition, incorporating changes in trace event contents resulted in significant bookkeeping problems. The resulting lack of stability in trace processing made the prospect of trace buffering dangerous, as incorrect unpacking of any single trace event might cause an entire buffer to be lost.

3 PVM 3.4 Tracing Facility

The goal of the new tracing facility for PVM 3.4 was to fix the shortcomings of the 3.3 tracing facility, and as much as possible plan for the future of PVM tracing needs. The primary features of the new facility are trace buffering, user-defined events, and flexible event contents. All of these are made possible through use of a new trace event structure, as discussed in the following subsections.

3.1 Trace Event Structure

The new trace event structure has been designed with flexibility in mind. The general approach is to explicitly define each trace event record at run time, before its appearance

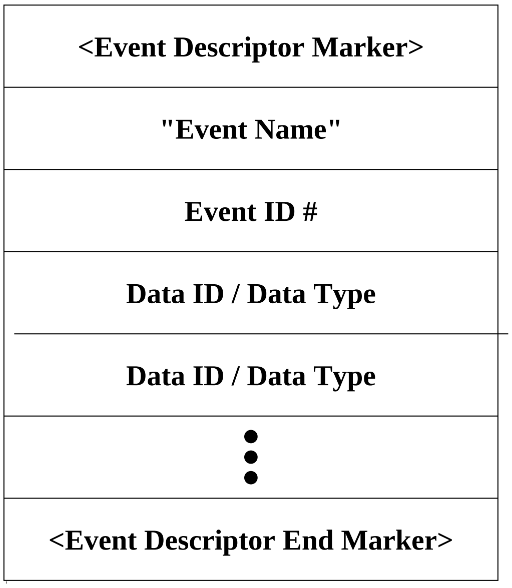

Figure 1: Trace Descriptor Format

in an event trace. This closely models the SDDF trace file format by including "trace descriptors" in the event stream. These descriptors precisely define the contents of each "trace record," and specify the data types and unpacking order of the trace information. In addition, some semantic identification is included to assist tools like XPVM in finding and processing the raw trace data for driving animated views.

Using trace descriptors alleviates the need for packing formatting separators into the trace records. The trace records can therefore contain only the necessary raw data. Trace descriptors are sent once for each task, only as needed, and are subsequently used as templates to unpack each trace record. The format of trace descriptors is shown in Figure 1. As with all the various trace message formats in PVM 3.4, integer markers are packed at the beginning and end of each functional unit to identify the unit and separate it from other units in case of buffering. For the descriptor, in between the "<Event Descriptor Marker>" and "<Event Descriptor End Marker>" markers there is a character string "Name" describing the event, as well as a unique integer "Event ID" number. Note that this event ID is unique only within the scope of any single task, as different tasks might associate different IDs with each event. In terms of XPVM event processing, only the name of the event is used for identification. The remainder of the descriptor unit is a list of "Data ID / Data Type" pairs that describe the individual trace data (see below).

The "Data ID" refers to a semantic notation required for all trace data. The ID is a 3-byte (character) mnemonic name that is associated with a full character string description of the data item. These IDs are well-known and permanent, to allow consistent identification of desired trace data in tools such as XPVM. It is this semantic identifier that allows trace record contents to change and be reordered without affecting XPVM's trace processing for views. Trace event records can be searched for any desired piece of semantic information.

<Event Data ID Marker>
Data ID #
"Data Description"
Data ID #
"Data Description"
⋮
<Event Data ID End Marker>

Figure 2: Semantic Identifier Unit Format

To allow some extensibility for future PVM releases, an additional trace event unit exists to define the above semantic identifiers. This unit can be used to add new semantic identifiers as well as to associate a different character string description with an existing semantic ID. (Note that changing the description text might be applied for clarification of meaning or for special trace file notations, and should not imply an actual semantic change lest backwards compatibility be lost.) The format of the semantic unit is shown in Figure 2. The unit consists simply of a list of mnemonic ID / description pairs.

The data type information included in trace descriptors is encoded in a single byte, so as to fit with a semantic data ID mnemonic into a single word for convenience. The possible data types supported are shown in Figure 3.1. Most of these data types are self explanatory, however a few are special-purpose in nature. The "Null" type indicates the absence of data and corresponds to an empty trace data marker. This may be used for special trace tool directives or other custom needs. The "Structure Start" and "Structure End" types are to be used in future extensions to define structured data, and act as markers around a structure definition consisting of a sub-list of data types. The "Deferred" type is intended for any trace data which may be of a variable type depending on the particular calling parameters of the traced routine. For deferred data an additional data type identifier will be packed with the actual data in the trace record event.

Note that the above data types require only 4 bits to specify, leaving an additional 4 bits for extensions. One of these bits has already been reserved for determining whether the trace data is scalar or an array, as shown in Figure 3.1. Arrays are considered one-dimensional for simplicity, hence the number of dimensions need not be specified.

Once the trace descriptors have been sent the corresponding trace records can also be sent, and then properly unpacked and placed into a trace file. The format of trace record events is

Null	0
Byte	1
Complex	2
Double Complex	3
Double	4
Float	5
Integer	6
Unsigned Integer	7
Long Integer	8
Unsigned Long	9
Short Integer	10
Unsigned Short	11
String	12
Structure Start	13
Structure End	14
Deferred	15

Figure 3: Descriptor Data Types

Data Scalar	0x00
Data Array	0x80

Figure 4: Descriptor Data Type Dimension

<Event Record Marker>
Event ID #
{ Event Data... }
<Event Record End Marker>

Figure 5: Trace Record Format

very simple, consisting only of an event ID number and the trace data itself. This format is shown in Figure 5. The "<Event Data Marker>" and "<Event Data End Marker>" markers are present to verify the boundaries among events in a buffer. It should be noted that the event end markers are employed to simplify the unpacking of a trace event buffer – they are not strictly necessary but are a handy safeguard and do not incur significant overhead.

Another performance issue concerns trace descriptors. Trace descriptors are constructed when the PVM library is compiled, not at run time. The descriptors are automatically generated by parsing the code that actually packs the trace records. In this way, there is no[1] chance for discrepancy between corresponding descriptor / record pairs. A direct result of this is the capability for reliable trace event buffering.

There is, of course, some overhead associated with sending the additional trace descriptor events. But, because trace buffering is possible the descriptors can easily be "piggy-backed" with the first instance of each trace record, thereby adding little message overhead. Also, because the descriptors are already constructed at compile time, a single conditional can be used to trigger the descriptor. On each event invocation (at the entry or exit to a traced routine), a simple flag is checked to determine if the descriptor has already been sent or should be packed into the event buffer.

Several alternate methods exist for communicating the trace descriptors, but they each have significant drawbacks. The descriptors could be constructed on-the-fly instead of at compile time, avoiding the pre-parsing step. But then significant overhead would result from the repeated descriptor checking at the packing of each piece of event data, let alone the complexity and bookkeeping of generating the descriptor and record simultaneously. Likewise, another alternative involves a special "merged" descriptor-record, with both formatting and data in one event. This approach would also incur repeated data packing overhead long after the descriptor information was sent, as the same conditionals would be executed again and again.

There are other benefits to the new tracing facility, in terms of maintenance, interoperability and extensibility. Because each task sends its own descriptors, it is possible for a variety of vendor and public PVM distributions, at different patch levels, to interoperate in

[1]theoretically...

the same trace. Tools such as XPVM need not endure the hassle of extensive bookkeeping to determine the various trace record contents for different versions, but instead can just maintain a list of the current descriptors for any given trace. New trace events can be added and trace record contents rearranged without modifying XPVM's trace file generation code. Also, any additions to view functionality, made possible by new events or information, can be implemented in a generic manner using the semantic identifiers and event names.

3.2 Trace Buffering

With the above new event structure, trace event buffering becomes a straightforward and safe feature to provide. Arbitrarily large collections of trace events can be correctly unpacked using the well-defined trace descriptor format information. Trace buffering in PVM 3.4 consists of packing multiple event units into the same PVM message buffer. Rather than sending many small event messages, one large message can be sent to reduce intrusion on the user application. This improves the accuracy of the tracing and the convenience to the user (in terms of less overhead).

The user interface to trace buffering in PVM 3.4 uses the existing PVM options routines, `pvm_setopt()` and `pvm_getopt()`. A user application can manually set its level of trace buffering using a call like:

`pvm_setopt( PvmTraceBuffer,` *size* `);`

where the *size* argument can be an arbitrary positive integer, or zero to buffer the entire execution's trace events. Note that in cases where the entire execution is to be buffered, the PVM task must exit normally using `pvm_exit()` or else the trace buffer will be lost.

Typically, the user will not need to manually set the level of trace buffering within the application, but can rely on XPVM to handle this automatically. When tasks are spawned from XPVM, they are set to trace certain routines and the trace buffering is set. The settings for the trace mask and the trace buffer size are both set initially on the XPVM Spawn dialog. XPVM version 1.1 also supports "on-the-fly" adjustments to various tracing parameters, including the trace buffer size.

3.3 User-Defined Trace Events

Another useful feature made possible with trace descriptors is the capability for the user to define custom trace events. Custom events make the tracing facility extensible for any special user needs. In addition, they provide a means for the advanced PVM user to determine the behavior of an application outside the scope of PVM system calls. While the remainder of the tracing facility is solely designed for providing information about the invocations of PVM library routines, user-defined trace events allow the user to collect information about the actual computation that is using PVM. With user-defined trace events, a programmer can send back intermediate results or just mark the passing of simple atomic occurrences, such as phases of a computation.

```
pvm_initsend( PvmDataTrace );
pvm_pkstr( "PVM is Great, Entering Phase III" );
pvm_pkint( &my_int_array, 10, 3 );
pvm_pkdouble( &my_double, 1, 1 );
pvm_send( PVM_TRACE_TID, PVM_TRACE_CODE );
```

Figure 6: Custom Trace Event Generation

For simplicity, the interface for custom trace events precisely resembles the regular PVM message interface. A special message encoding "PvmDataTrace" has been added, so that the user can use the existing `pvm_pk*()` routines to pack up an event as easily as any other PVM message. An example custom event is shown in Figure 6. The PvmDataTrace encoding automatically inserts sufficient data typing information in the message so as to allow correct unpacking of the event by a tool such as XPVM. If trace buffering is in effect, the message will be appended directly to any existing trace buffer. After the event is packed, `pvm_send()` is called with special predefined constants for the destination task ID and message code. These constants dereference to the correct internal tracing destination, e.g. XPVM. If the trace buffer is full or buffering is not activated, then the `pvm_send()` call will flush the custom event, along with any other buffered events. Otherwise, the custom event will wait in the buffer for a future flush.

Like other trace events, user-defined events can be controlled by setting the PVM trace mask. A single trace mask entry can be used to activate or deactivate all the user-defined trace events in an application. This control can be exerted at run time without re-compilation, either at spawn time or on-the-fly using XPVM. If tracing is not activated or the trace mask for the user-defined events is not set, then the PVM packing and sending routines will become NOPs, and have no effect.

If XPVM is being used to collect trace events, then the custom events will show up in the XPVM views with the other regular events. A special "tick" mark will appear on the Space-Time View, and a textual representation of the data sent will appear in the Call Trace View.

4 XPVM 1.1

To fully support the new features and information available in the PVM 3.4 tracing facility, a new release of XPVM, version 1.1, has been developed. This new version includes a new Message Queue View and enhancements to several existing views. This new version has also been optimized and re-organized to improve the "friendliness" of its user interface. The resulting interface is shown at the end of the paper in Figure 12.

The new XPVM is fully backward compatible with the previous PVM 3.3 tracing facility event formats, and can simultaneously handle either format. Because, however, PVM 3.3

and 3.4 support sufficiently different message passing protocols, XPVM must be compiled for one or the other PVM version to actually receive particular trace event messages. (Note however that, for example, vendor releases of PVM which are message compatible, but use the old trace format, can still interoperate.)

XPVM 1.1 is also fully compatible with both sets of the TCL and TK tool language systems. XPVM is written using the interpreted TCL and TK script languages for interface layout and graphics, with integrated C routines for internal computation and data structures. In the Spring of 1995, new versions of TCL and TK were released which were not backwards compatible with existing TCL / TK applications. The new XPVM can be compiled to work with either TCL / TK release.

Some general optimizations have been applied to XPVM 1.1 as well. Because the interpreted TCL and TK languages tended to be rather slow in XPVM 1.0, much of the functionality that did not have to deal directly with the graphical interface has been migrated to faster C code. This resulted in an order of magnitude improvement in many code segments.

The following subsections describe the new view features in XPVM 1.1 in more detail.

4.1 Network View Enhancements

The Network View is a depiction of the high-level activity on the hosts in the virtual machine, with colored icon images that are animated according to the status of each host. One of the drawbacks of the original XPVM 1.0 Network View was that it did not indicate any information regarding the load of the network or its performance. In fact, the so-called "Network" View was really more of a "Hosts" View, and showed only information regarding the utilization of hosts in the virtual machine network.

The new Network View contains animated depictions of the instantaneous network bandwidth and volume on individual network links. At present, the primary network topology supported in XPVM is that of a simple bus, as seen with typical Ethernet networks. Independent of the actual underlying network topology, the Network View maps the host interconnections to the bus layout. Each individual link – from each host to the bus, and the actual bus links between hosts connections – is animated. An example Network View is shown in Figure 7.

The width of a particular link corresponds to the bandwidth most recently realized on that link. Because it is not known in advance how long a particular message will take to be transmitted across the network, the bandwidth is calculated upon message receipt. Specifically, the bandwidth is the average communication rate, in bytes per second, from the ratio of total message size to total travel time. This bandwidth information can be used to determine the load and performance of the network connecting hosts in the virtual machine, thereby providing insight into the impact the network is having on the performance of the application.

The message volume is animated by the color of individual network links, and represents a wide spectrum of ranges from bytes to kilobytes to megabytes. The volume is computed

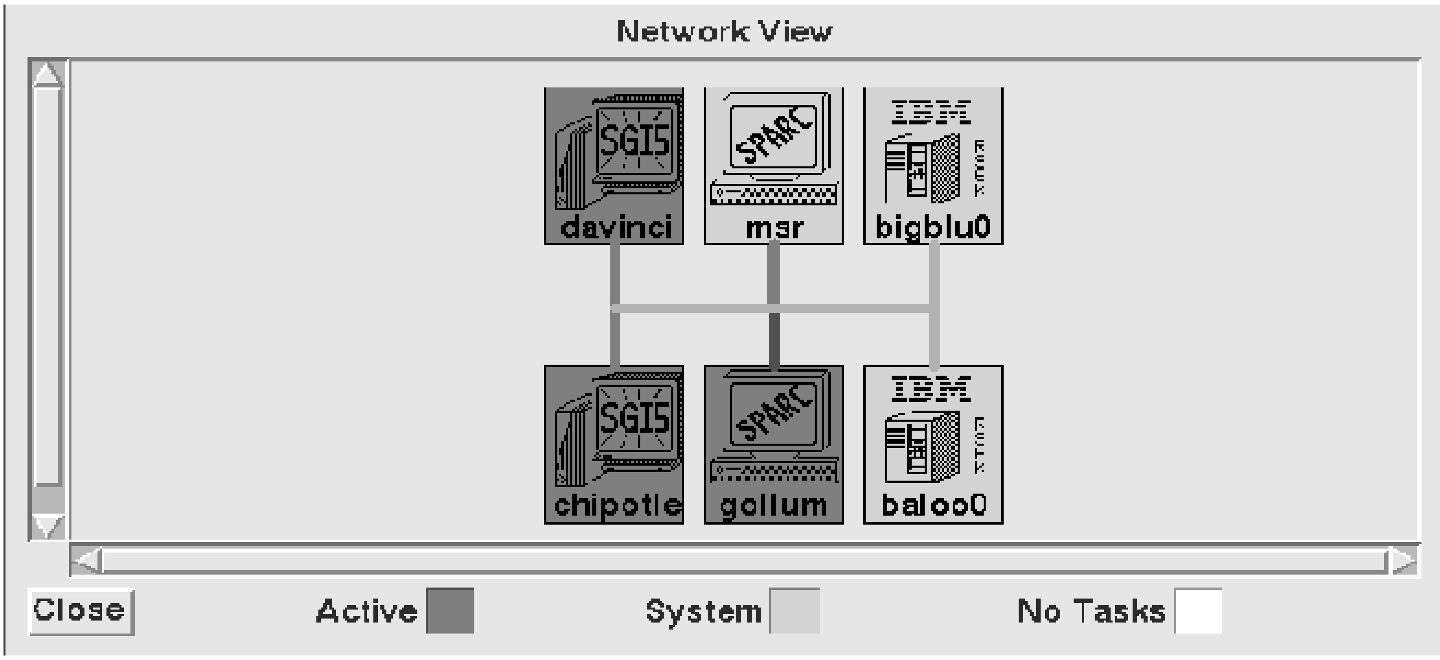

Figure 7: Enhanced Network View

on-the-fly by adding and subtracting the size of messages as they enter and leave in transit over the network. This information provides an indication to the user about the existence and magnitude of any communication bottlenecks in the application.

4.2 Utilization View Optimizations

The Utilization View shows the cumulative status of all tasks executing at each time interval. The view is depicted by a long line of histogram stacks, three rectangles per stack, spread along a horizontal time axis. The three rectangles represent the number of tasks at that time that are busy computing, in system overhead and idle blocked on communication, respectively.

In the previous version of XPVM 1.0, the Utilization View is the most computation-intensive view animation. The problem comes from the fact that events are collected over a network from machines at various physical (and effective) distances. In effect, events from certain hosts will arrive "behind schedule" compared to other hosts, resulting in an unfortunate "re-calculation" of the current state of several views.

Such is the case with the Utilization View, in that events can arrive at times arbitrarily back "in the past," requiring adjustment of the utilization task counts from that time all the way through the current trace time. This repeated recursive traversal of the utilization history becomes an immense performance problem, especially with large, non-trivial traces.

The solution in XPVM 1.1 is a slight modification of the logic of the Utilization View that prevents this costly recursion. The original Utilization View assumes a continuation of

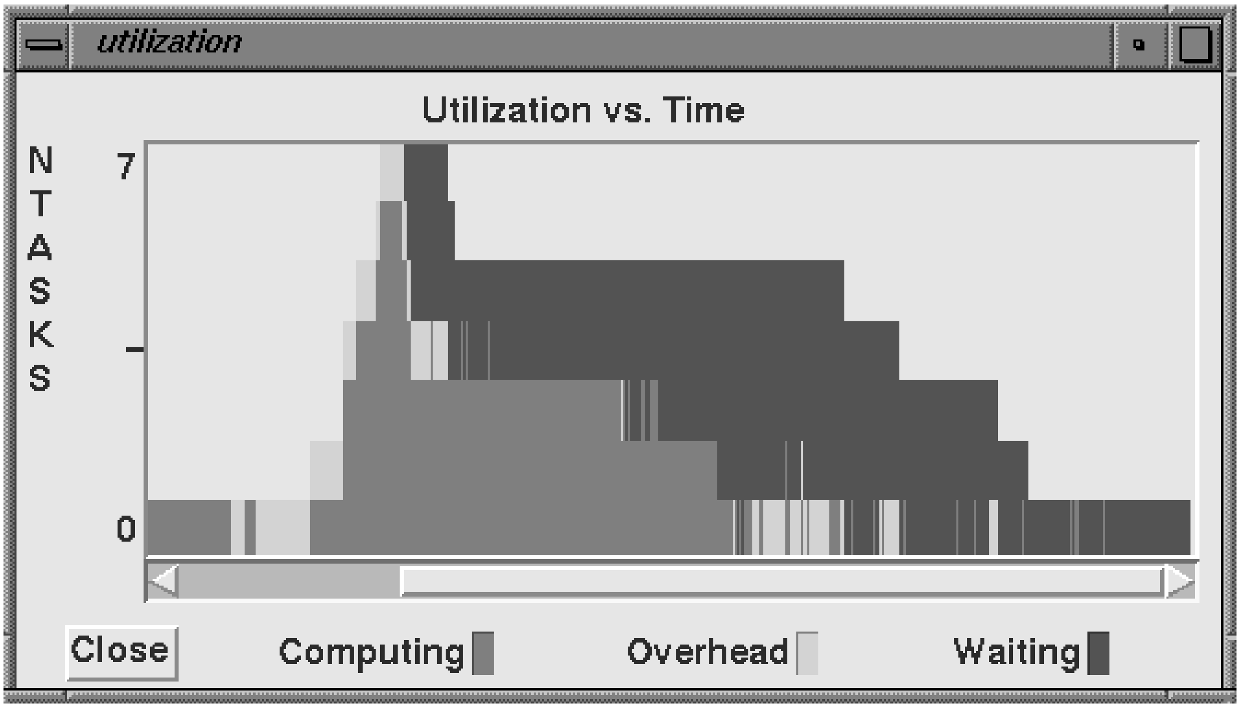

Figure 8: Optimized Utilization View

the last recorded state for a task if no events are received. Thus, the right edge of the view always maintains its full height, as if the state of all tasks is known. In the optimized view, a task's status is not added to the utilization task counts until that task's state is known for a particular time range. This results in a "trailing edge" as tasks gradually fill in their events from left to right, as shown in Figure 8. The overall result is a more correct depiction of the true task utilization (without any recursive "re-calculations") that is significantly more efficient, by several orders of magnitude.

4.3 Message Queue View

The Message Queue View is a new view introduced in XPVM 1.1. This view shows the instantaneous number and size of any messages that are buffered and waiting to be received for each task. The view consists of a collection of "message stacks," one for each task, that represent the pending messages. Each stack is made up of rectangles that represent the individual messages that have been sent, but not yet received. This information is especially useful in identifying any communication "hot spots" at particular tasks – much like the message volume animation in the enhanced Network View, but at the higher task resolution rather than the host level. A sample of this view is shown in Figure 9.

As with many of the views provided in XPVM, the Message Queue View is "clickable" and the user can obtain detailed textual information by clicking on individual message rectangles. This information completely describes the pending message, including the source task of the

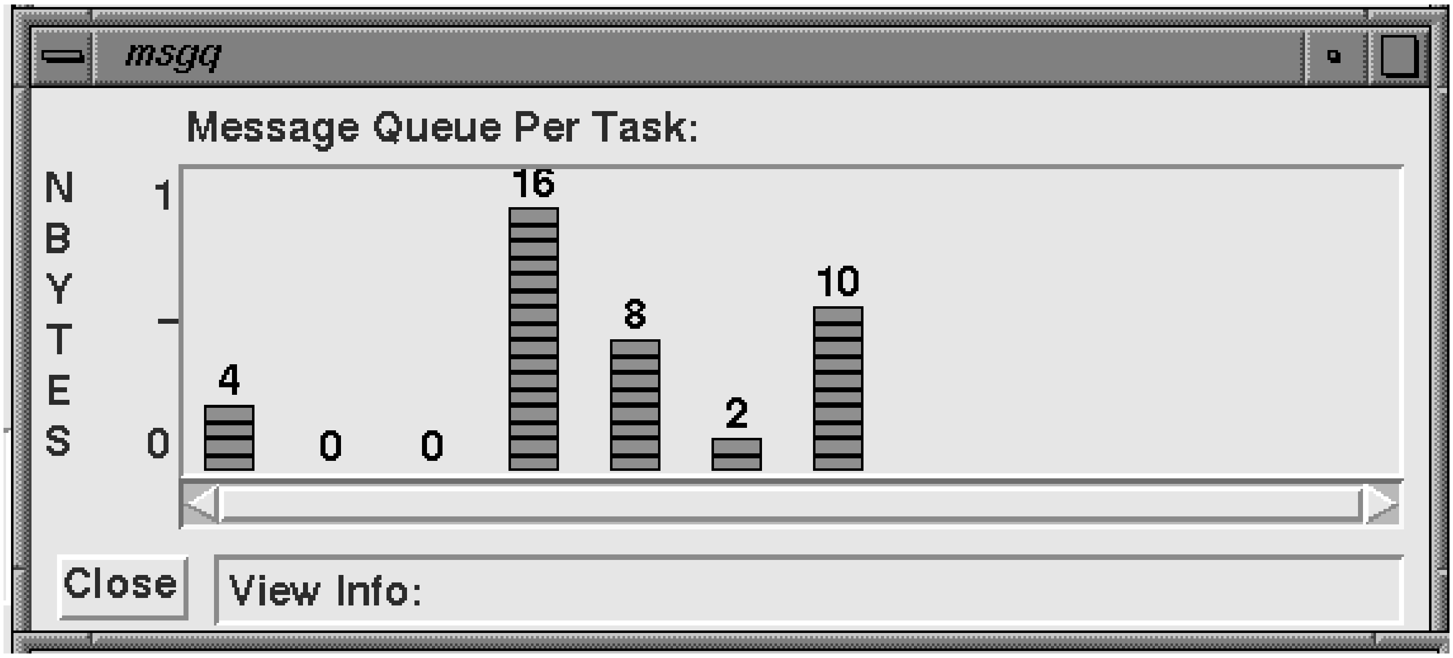

Figure 9: New Message Queue View

message, the designated message code, and the size of the message in bytes (if known, see below).

The full implementation of the Message Queue View required the flexibility of the new tracing facility in PVM 3.4. To correctly depict the size of pending messages their size in bytes must be known at send time. In the original PVM 3.3 tracing facility, the need for this information had not been anticipated, and so none of the PVM message sending routines had been properly instrumented. It was only using the PVM 3.4 tracing facility that the size of the outgoing message could be added to the send events.

For compatibility with PVM 3.3 tracing, when the message size is not known a default size is used in its place. This default size is settable using the "-M *size*" command line option to XPVM. For clarity when viewing the Message Queue View, any messages for which the correct size is not known are animated in an orange color, and rectangles with a known size are shown in purple.

5 Results

To illustrate the improvements in performance gained by the modifications to the tracing facility in PVM, as well as the optimizations in XPVM, a representative set of example timings was collected. A parallel Cholesky matrix factorization algorithm was applied for this example, using a 100x100 matrix. The results of these timings are presented in the following subsections.

5.1 Enhanced PVM Performance

The performance degradation of tracing the PVM Cholesky application was measured for both the old PVM 3.3 and new PVM 3.4 tracing facilities, with all possible trace events activated. For the PVM 3.4 timing runs, three different trace buffer sizes were used, for 1, 8 and 1000 buffered trace events. The results are shown in Figure 10. For the original PVM 3.3 tracing facility, the Cholesky example took over 33 seconds with all events traced, versus only 8 seconds without tracing. This constitutes a 400original tracing facility. For the new PVM 3.4 tracing facility with no trace buffering (buffer size set to 1), a comparable result was experienced, with a time of about 34 seconds for all events traced. The minor difference between the new and old tracing facilities at this level is due to the additional trace descriptor information – clearly not a significant impact on performance.

When trace buffering is applied in the PVM 3.4 tracing, a significant improvement is seen. For a trace buffer size of 8, the execution time drops down to about 23 seconds, for an overhead of less than 300no buffering). For a large trace buffer size of 1000 events, the tracing overhead becomes almost negligible. The execution time there is only a bit over 8 seconds, for a total overhead of only 2-3

5.2 Optimized XPVM Performance

To examine the improvement in efficiency of the new optimized XPVM implementation, a sample trace file from the above PVM timing comparisons was run through both XPVM 1.0 and 1.1. The timing differences were staggering, as shown in Figure 11. For a large trace file (approximately 0.5 Megabytes) representing a 34 second execution, the XPVM 1.1 version was able to animate the entire trace in exactly 2 minutes with the Network and Space-Time Views activated. Playing the same trace through XPVM 1.0 was infeasible, taking a full 30 minutes for the first 6 seconds of trace time. Projecting from this first 30 minutes, the total time to animate the trace in XPVM 1.0 is well over 2.5 hours. This is a difference of 75 times, or almost 2 orders of magnitude.

Similarly, for the same trace file above run through with *all* views activated (including the extra Message Queue View not available in XPVM 1.0), the XPVM 1.1 animation took only 191 seconds compared to an unbelievably slow XPVM 1.0 animation which only reached 4 seconds of trace time after 30 minutes. This also amounts to close to two orders of magnitude improvement.

6 Future Work

A new tracing facility for PVM 3.4 has been implemented which supports more flexible and efficient access to run-time parallel program information. The new tracing system supports buffered tracing to reduce the perturbation of user applications that inhibits debugging. This facility also improves the accuracy of trace information to better assist in performance tuning. The new system supports custom user-defined events as well. In conjunction with the new

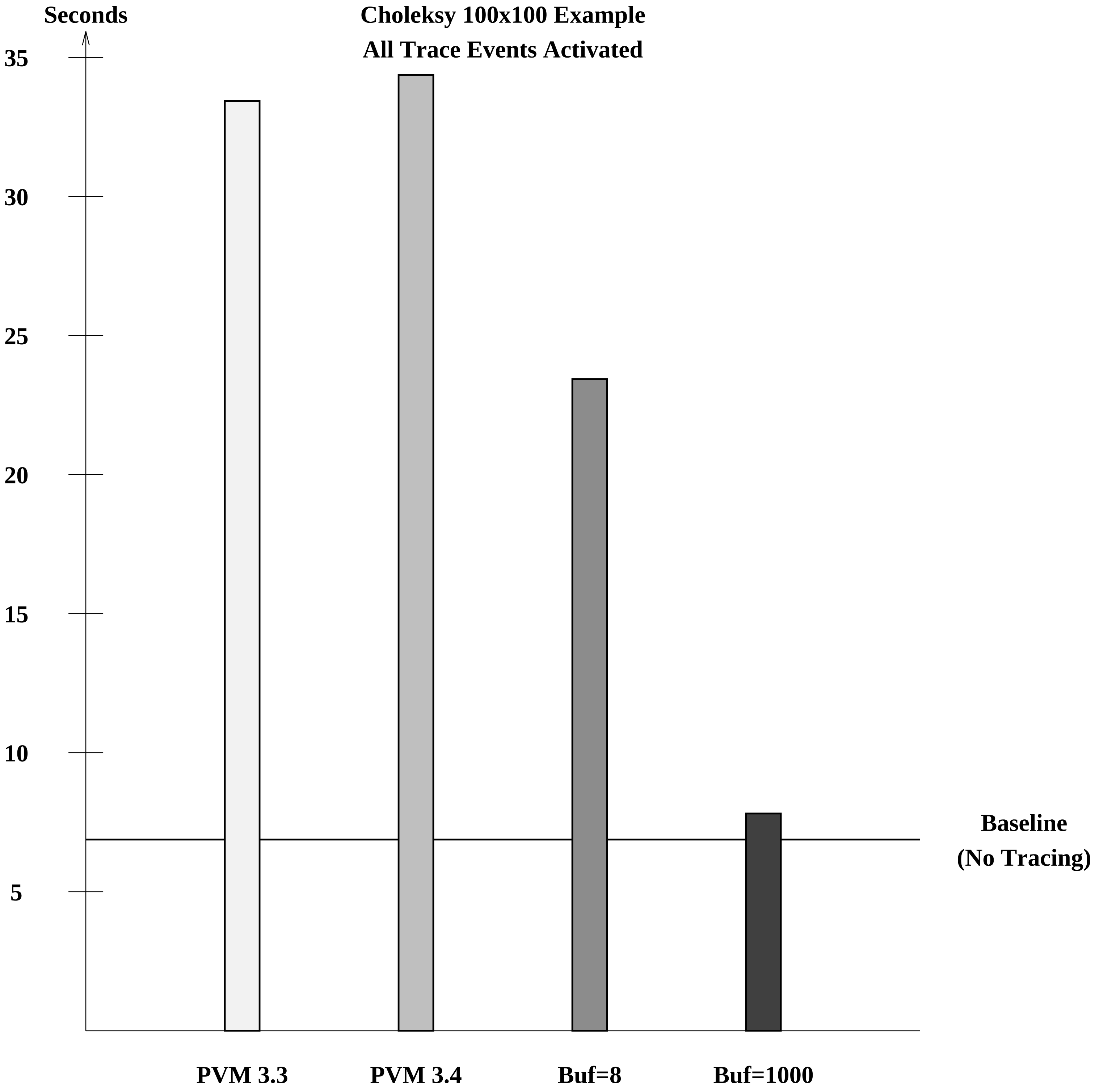

Figure 10: PVM Tracing Facility Comparison

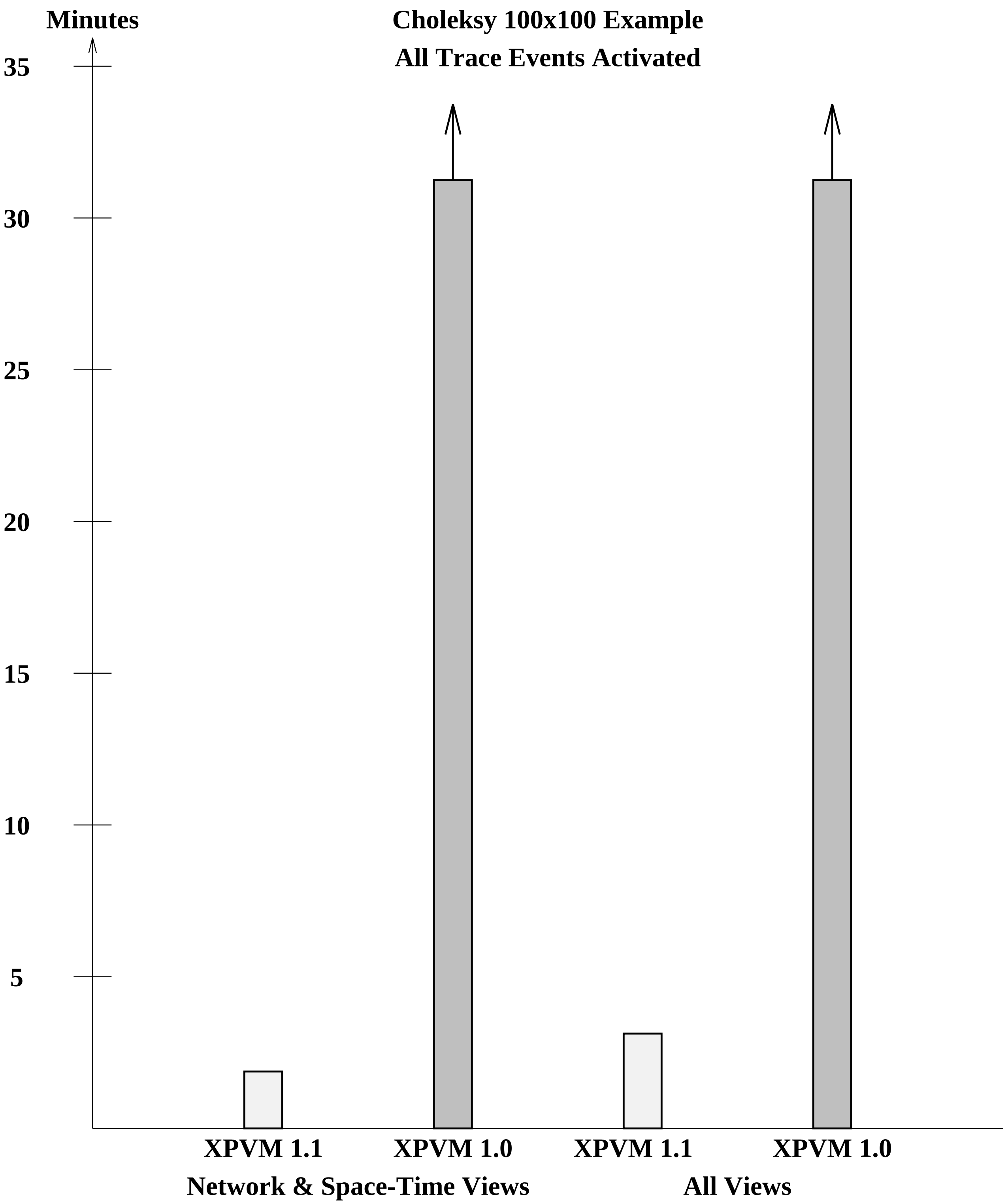

Figure 11: XPVM Efficiency Comparison

PVM 3.4 tracing, an enhanced version of XPVM 1.1 has been developed which utilizes and controls the new tracing information to provide new and extended animated views. Both the new tracing facility and XPVM 1.1 have been shown to perform substantially more efficiently in collecting and animating parallel program traces than the previous systems.

There are several directions for future work related to these systems. In terms of the PVM tracing facility, there is a need for *less* detailed tracing information, with even less impact and perturbation of user applications, that can be used for more effective performance tuning and other general analyses. Due to the immense amount of information that can be obtained regarding a parallel program's execution, it is critical to find ways to reduce the size of execution traces.

Likewise, for tools such as XPVM it is essential to continue searching for better ways to visualize the mountains of trace data collected, so that users can more readily understand the behavior of interest. Furthermore, tools need to assist users by pointing out anomalous behavior which may not be obvious or expected by the user. It seems logical that the wealth of existing statistical analysis techniques should be applied to this end.

References

[1] R. A. Aydt, "The Pablo Self-Defining Data Format," University of Illinois at Urbana-Champaign, Department of Computer Science, February 1993.

[2] M. T. Heath, J. A. Etheridge, "Visualizing the Performance of Parallel Programs," IEEE Software, Volume 8, Number 5, September 1991, pp. 29-40.

[3] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, "PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing," The MIT Press, 1994.

[4] D. Reed, R. Olson, R. Aydt, T. Madhyastha, T. Birkett, D. Jensen, B. Nazief, B. Totty, "Scalable Performance Environments for Parallel Systems," Proceedings of the Sixth Distributed Memory Computing Conference, IEEE Computer Society Press, April 1991.

[5] P. H. Worley, "A New PICL Trace File Format," Technical Report ORNL/TM-12125, Oak Ridge National Laboratory, Oak Ridge, TN, October 1992.

[6] J. A. Kohl, G. A. Geist, "XPVM 1.0 User's Guide," Technical Report ORNL/TM-12981, Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, April 1995.

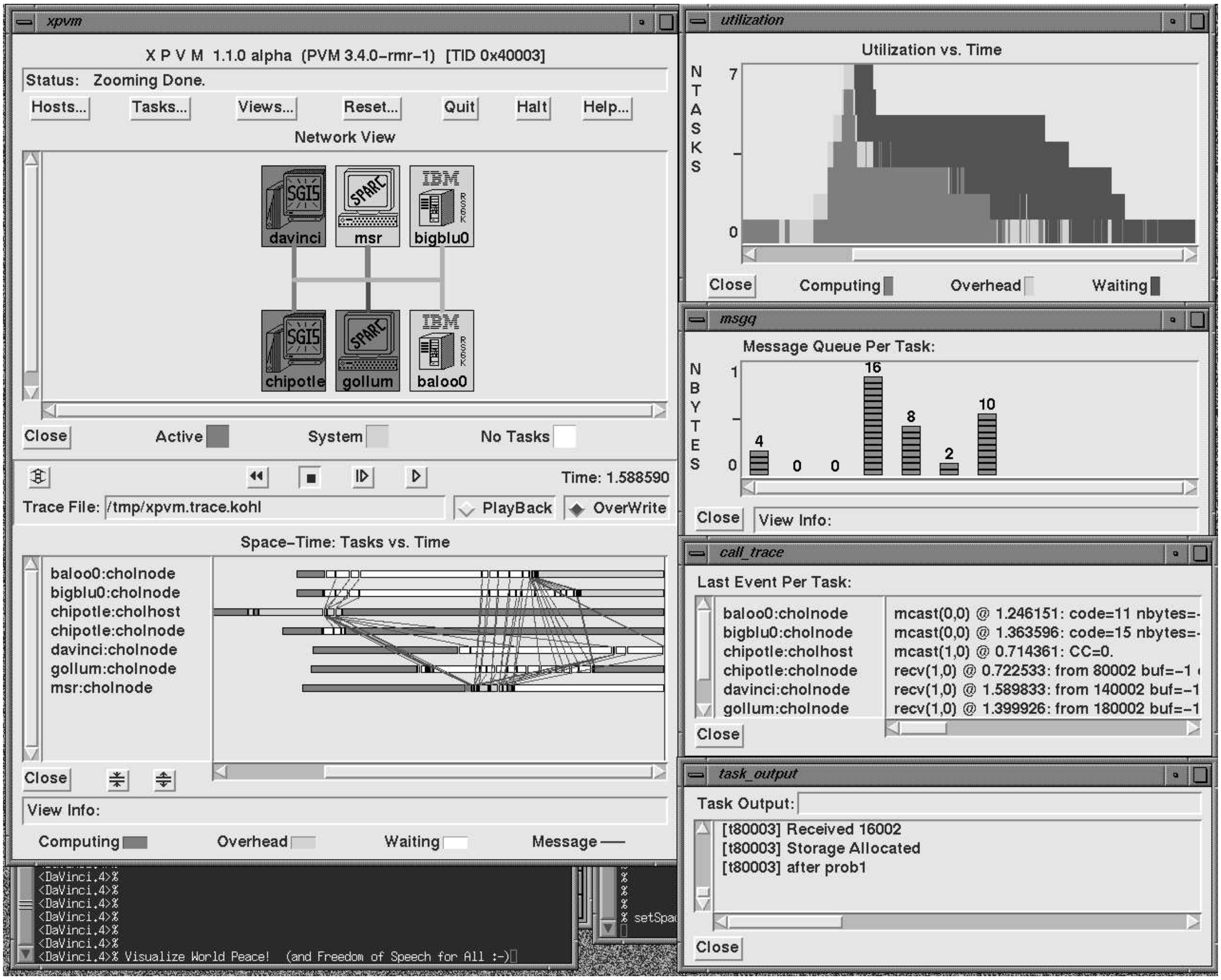

Figure 12: XPVM 1.1 Interface