Recent Enhancements to PVM

Adam Beguelin
Carnegie Mellon University, and
Pittsburgh Supercomputing Center

Jack Dongarra
University of Tennessee, and
Oak Ridge National Laboratory

Al Geist
Oak Ridge National Laboratory

Robert Manchek
University of Tennessee

Vaidy Sunderam
Emory University

December 13, 1994

Abstract

This paper presents new features of PVM, a popular standard for writing parallel programs which execute over networks of heterogeneous machines. Although PVM has become an important infrastructure for parallel programmers, we continue to develop the system based both on user feedback and our own research interests. In this paper we present new communications routines and briefly characterize their performance. We describe new extensible services that allow advanced users to customize certain aspects of the default PVM functionality. An overview of shared memory PVM optimizations is presented. We also describe PVM's new tracing facility and a graphical console which utilizes this capability. Finally, we discuss future extensions to PVM which are under investigation.

1 Introduction

The past several years have witnessed an ever-increasing acceptance and adoption of parallel processing, both for high-performance scientific computing as well as for more general-purpose applications. Furthermore, the message-passing model appears to be gaining predominance as the paradigm of choice, in terms of multiprocessor architectures as well as applications, languages, and software systems for message passing support.

PVM (Parallel Virtual Machine) [6] was produced by the Heterogeneous Network Project — a collaborative effort by researchers at Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie Mellon University specifically to facilitate heterogeneous parallel computing. PVM is a software system that allows programmers to utilize a network of heterogeneous computers, some of which may be MPPs, as a single multicomputer. This system has become popular both for developing parallel applications and as an infrastructure for developing more advanced parallel programming tools.

Version 3 of the PVM system is composed of two parts. The first part is a daemon, called *pvmd3* that resides on all the computers making up the virtual computer. (An example of a daemon program is *sendmail*, which handles all the incoming and outgoing electronic mail on a Unix system.) *pvmd3* is designed so that any user with a valid login can install this daemon on a machine. When a user wishes to run a PVM application, he executes *pvmd3* on one of the computers which in turn starts up *pvmd3* on each of the computers making up the user-defined virtual machine. A PVM application can then be started from a Unix prompt on any of these computers.

The second part of the system is a library of PVM interface routines. This library contains user-callable routines for passing messages, spawning tasks, coordinating those tasks, and modifying the virtual machine. Application programs must be linked with this library to use PVM.

In this paper we describe new features of the latest version of PVM (3.3) and present an overview of future directions we are exploring with respect to PVM.

2 Point to Point Communication

In general, PVM programs consist of tasks that communicate via messages. A task is a basic unit of computation in PVM, a Unix process for instance. From the first release of PVM 3.0 until the recent release of PVM 3.3 there has been only a single routine to send a message to another task. In PVM 3.3 there is an additional routine for sending and two new routines for receiving messages. In this section we describe the point to point communication routines in PVM 3.3 and show how to enhance the performance of applications by using these routines.

The philosophy of PVM has always been to keep the user interface simple and easy to understand, letting PVM do all the hard work underneath in order to improve performance. For example, when a user sends a message, he would like the data to arrive instantly at the destination. In reality this can never happen. There is always some startup latency plus the time it takes to actually move the data. These overheads can not be avoided but they can be masked by other work. Some message passing interfaces, such as MPI [5], go to great lengths to supply many variants of 'send' in order to allow the user several ways to explicitly manage the masking of the send overheads. This is good if the goal is to supply the ability to achieve the ultimate peak performance of a large multiprocessor, but it requires an expert in parallel programming to achieve this peak. The vast majority of scientists and engineers who use parallel programming are not experts in it. They use the basic send and receive primitives in their codes.

The PVM communication model assumes that any task can send a message to any other PVM task, and that there is no limit to the size or number of such messages. While all hosts have physical memory limitations that restrict potential buffer space, the communication model does not restrict itself to a particular machine's limitations. It assumes sufficient memory is available. PVM allocates buffer space dynamically so the size or volume of messages that can arrive at a single host at the same time is only limited by the available memory on the machine.

The PVM communication model provides asynchronous blocking send, asynchronous blocking receive, and non-blocking receive functions. In our terminology, a blocking send returns as soon as the send buffer is free for reuse, and an asynchronous send does not depend on the receiver calling a matching receive before the send can return. A non-blocking receive immediately returns with either the data or a flag that the data has not arrived, while a blocking receive returns only when the data is in the receive buffer. Wildcards can be specified in the receive for the source and message type, allowing either or both of these contexts to be ignored. A routine can be called to return information about received messages.

The PVM model guarantees that message order is preserved between two tasks. If task 1 sends message A to task 2, then task 1 sends message B to task 2, message A will arrive at task 2 before message B. Moreover, if both messages arrive before task 2 does a receive, then a wildcard receive will always return message A. The programmer can also specify a specific message type, called a tag. When a tag is specified, PVM will return the first incoming message with the requested tag.

Until PVM 3.3, sending a message with PVM required three function calls. First, a send buffer is initialized by a call to pvm_initsend(). This step also has the effect of clearing the previous send buffer, if any. Second, the message must be "packed" into this buffer using any number and combination of pvm_pk*() routines. At this step PVM takes care of any data encoding needed for heterogeneity and builds the buffer in fragments (required by the network protocols) so the overhead of fragmenting a large buffer during transmission is avoided. Third, the completed message is sent to another process by calling the pvm_send() routine.

There are several advantages of having this three step method. The method allows the user to pack a message with several different pieces of data. For example, a message may contain a floating point array and an integer defining its size. Or a single message may contain an entire 'structure' including integer arrays, character strings, and floating point arrays. Why is this important? Packing a message is fast relative to transferring the data over a network, although this is beginning to change as networks become faster. By being able to combine several different pieces of information into a single message, the user can decrease the number of sends in an algorithm. This eliminates the startup latency for all the sends that are saved. Another important advantage is the avoidance of matching 'structures' back up on the receiver. Let's illustrate with a contrived example. Assume we restrict messages to a single data type and the data structure to be sent is a sparse floating point array without the zeros, an integer specifying the number of floats, and an integer array of indices corresponding to the matrix location of each floating point value. Now assume one task has to receive a few structures of this kind from several other tasks. Since the order that the floating point and integer messages arrive at the receiver is arbitrary, because messages may come from different sources, several structures could be interleaved in the message queue. The receiver then is responsible for searching the queue and properly reassembling the structures. This search and reconstruct phase is not needed when the various data types are combined into the same message. The philosophy of PVM is to be simple to use. It is easy for a non-expert to understand the concept of packing up a structure of data, sending it, and unpacking the message at the receiver.

Another advantage of the three step method is the need to encode and fragment the message only once. In PVM once the message is packed, it can be sent to several different destinations. There are many parallel scientific applications were a task must send its data to its 'neighbors'. In cases like this, PVM eliminates the overhead of packing each send separately. PVM also takes advantage of packing only once when a user broadcasts a message.

The separate buffer initialization step also has the advantage that the user can append data onto a buffer that is already sent. Since PVM doesn't clear the buffer until the next pvm_initsend() call, a task can pack and send a message to one destination then append to that message and send it to another destination and so on. There are certain ring algorithms that benefit from such a capability, but the ability to append to messages is not commonly used.

Although there are several advantages to the three step send, there are many parallel algorithms that just need to send one array of a given data type to one destination. Because this type of message is so common, it would be useful to avoid the the three step send in this case. In PVM 3.3 this is now possible by using the new function pvm_psend(). pvm_psend() combines the initialize, pack, and send steps into a single call with an orientation towards performance.

The request to have pvm_psend() and its compliment pvm_precv() initially came from MPP vendors who were developing optimized PVM versions for their systems. On MPP systems vendors try to supply the smallest possible latency. The overhead of three subroutine calls is large relative to the raw communication times on MPP systems. The addition of pvm_psend/pvm_precv to PVM has significantly boosted the performance of point-to-point PVM communication on MPP machines. For example, Table 1 shows that the message passing passing performance on the Intel Paragon using pvm_psend/pvm_precv is only 5-8% larger than the native calls csend/crecv. This low overhead on the Paragon can be attributed to the close mapping between the functionality of the PVM calls and Intel's native calls. On the Cray T3D, PVM is the native message passing interface. The latency for pvm_psend() on the T3D is only 18 microseconds and the bandwidth is over 45 Mbytes/sec.

PVM contains several methods of receiving messages at a task. There is no function matching requirement in PVM, for example it is not necessary that a pvm_psend be matched with a pvm_precv. Any of the following routines can be called for any incoming message no matter how it was sent (or multicast).

- pvm_recv() – blocking receive
- pvm_trecv() – timeout receive
- pvm_nrecv() – nonblocking receive

Paragon csend/crecv

Msg size (bytes)	Round trip (usec)	bandwidth (MB/s)
8	376	0.0213
80	315	0.2540
800	314	2.5478
8000	670	11.9403
80000	3978	20.1106

Paragon pvm_psend/pvm_precv

Msg size (bytes)	Round trip (usec)	bandwidth (MB/s)
8	322	0.0248
80	317	0.2524
800	314	2.5478
8000	676	11.8343
80000	4061	19.6996

Table 1: Paragon node to node round trip comparison of PVM and native calls.

- pvm_precv() – combined unpack and blocking receive

As a new function in PVM 3.3, PVM supplies a timeout version of receive. Consider the case where a message is never going to arrive (due to error or failure). The routine pvm_recv would block forever. There are times when the user wants to give up after waiting for a fixed amount of time. The routine pvm_trecv allows the user to specify a timeout period. If the timeout period is set very large then pvm_trecv acts like pvm_recv. If the timeout period is set to zero then pvm_trecv acts like pvm_nrecv. Thus, pvm_trecv fills the gap between the blocking and nonblocking receive functions.

2.1 Performance

There are several options at the user's disposal that allow PVM to optimize communication on a given virtual machine. Communication across nodes of an MPP and across processors of a shared memory multiprocessor are automatically optimized by using the native communication calls and shared memory respectively. The following discussion restricts itself to performance improvements across a network of hosts.

PVM uses UDP and TCP [4] sockets to move data over networks. UDP is a connectionless datagram protocol packet delivery is not guaranteed. TCP requires a connection between processes and implements sophistocated retry algorithms to ensure data delivery. In PVM the default, scalable transfer method is for a task to send the message to the local PVM daemon. The local daemon transfers the message to the remote daemon using UDP, finally the remote daemon transfers the message to the remote task when requested by a pvm_recv(). Since UDP does not guarentee packet delivery, we implement a lightweight protocol to assure full message delivery between daemons. PVM 3.3 improves the performance of this route by using Unix domain sockets between tasks and the local PVM daemon. Measurements showed this improved the task to daemon latency and bandwidth by a factor of 1.5 to 2.

A less scalable but faster transfer method is available in PVM. By calling `pvm_setopt(PvmRoute, PvmRouteDirect)`, PVM will set up a direct task to task TCP link between the calling task and any other task it sends to. The initial TCP set up time is large but all subsequent messages between the same two tasks have been measured to be 2-3x faster than the default route. The primary drawback of this method is that each TCP socket consumes one file descriptor (fd). Thus there is the potential to need $O(n^2)$ file descriptors, where n is the number of tasks in the virtual machine. Since direct routing only involves a single call at the top of a PVM program, it is reasonable to try PvmRouteDirect to see if it improves the performance of an application.

There are two encoding options available in PVM 3.3 whose purpose is to boost communication performance. Since a message may be sent to several destinations, by default PVM will encode it for heterogeneous delivery during packing. If the message will only be sent to hosts with compatible data format, then the user can tell PVM to skip the encoding step. This is done by calling `pvm_initsend(PvmDataRaw)`.

The second encoding option is `pvm_initsend(PvmDataInplace)`. When PvmDataInplace is specified the data is never packed into a buffer. Instead it is left "inplace" in user memory until pvm_send() is called and then copied directly from user memory to the network. During the packing steps PVM simply keeps track of where and how much data is specified. This option reduces the pack time dramatically. It also has the benefit of reducing memory requirements since the send buffer no longer holds a copy of the message.

One must exercise care when using PvmDataInPlace. If the user's data is modified after the pack call but before the send call then the modified data will be sent, not the data originally specified in the pack call. This behavior is different from using the other pvm_initsend() modes where the data is copied at pack time.

As mentioned earlier, pvm_psend() was implemented for performance reasons. As such it uses PvmDataInplace. This coupled with only one call overhead makes pvm_psend(), when combined with PvmRouteDirect, the fastest method to send data in PVM 3.3.

Figure 1 plots bandwidth versus message size for various packing and routing options. The lines marked `dir` and `hop` indicate direct and default routing respectively. (Hop because the default messages make extra hops through the PVM daemons.) Inplace packing and unpacking is indicated by `inp`. Lines marked with `raw` show the case where no data conversion was done while `xdr` indicates messages were converted into the XDR format before sending and from the XDR format after recieving them at the destination. The tests were run on Dec Alpha workstations connected by FDDI. This experiment shows that the avoidance of data copying and conversion along with direct routing allows PVM to achieve good end to end performance for large messages. The peak bandwidth of FDDI is 100Mbit/s or 12.5 MByte/s. In the best case we achieve approximately 8 MByte/second bandwidth for large messages, 64% of the network's peak bandwidth. Note that these times include both packing the message at the sender and unpacking the message buffer at the receiver. The advantage of inplace packing for large messages is clearly shown. The high cost of using heterogeneous data conversion can also bee seen from the XDR bandwidth curves.

Figure 2 shows latency measurements for the same experiment. We see that latency is much smaller when using directly connected message routing. Both raw and inplace packing achieve the lowest latency with inplace being slightly better for large messages.

3 Collective Communication

PVM 3 has always had a very flexible and powerful model of groups of tasks but until PVM 3.3 there were only two collective communication routines: broadcast to a group of tasks and barrier across a group of tasks. PVM 3.3 adds several new collective communication routines including global sum, global max, and scatter/gather.

The semantics of the PVM collective communication routines were developed using the MPI draft as a guide but also sticking with the PVM philosophy of keeping the user interface simple and easy to understand. The purpose of adding more collective routines is to avoid users reinventing the wheel, and to allow MPP implementations to exploit any built-in native collective routines

The pvm_reduce() function performs a global arithmetic operation across the group, for example, global sum or global max. The routine is called by all members of the group, and the result of the reduction operation appears on the member specified as root, also called the root task. PVM supplies four predefined reduce functions:

- `PvmMax` – global maximum
- `PvmMin` – global minimum
- `PvmSum` – global sum
- `PvmProduct` – global product

These reduction operations are performed element-wise on the input data. For example, if the data array contains two floating point numbers and function is PvmMax, then the result contains two numbers — the global maximum of each group members first number and the global maximum of each members second number. The Fortran code fragment to do this is:

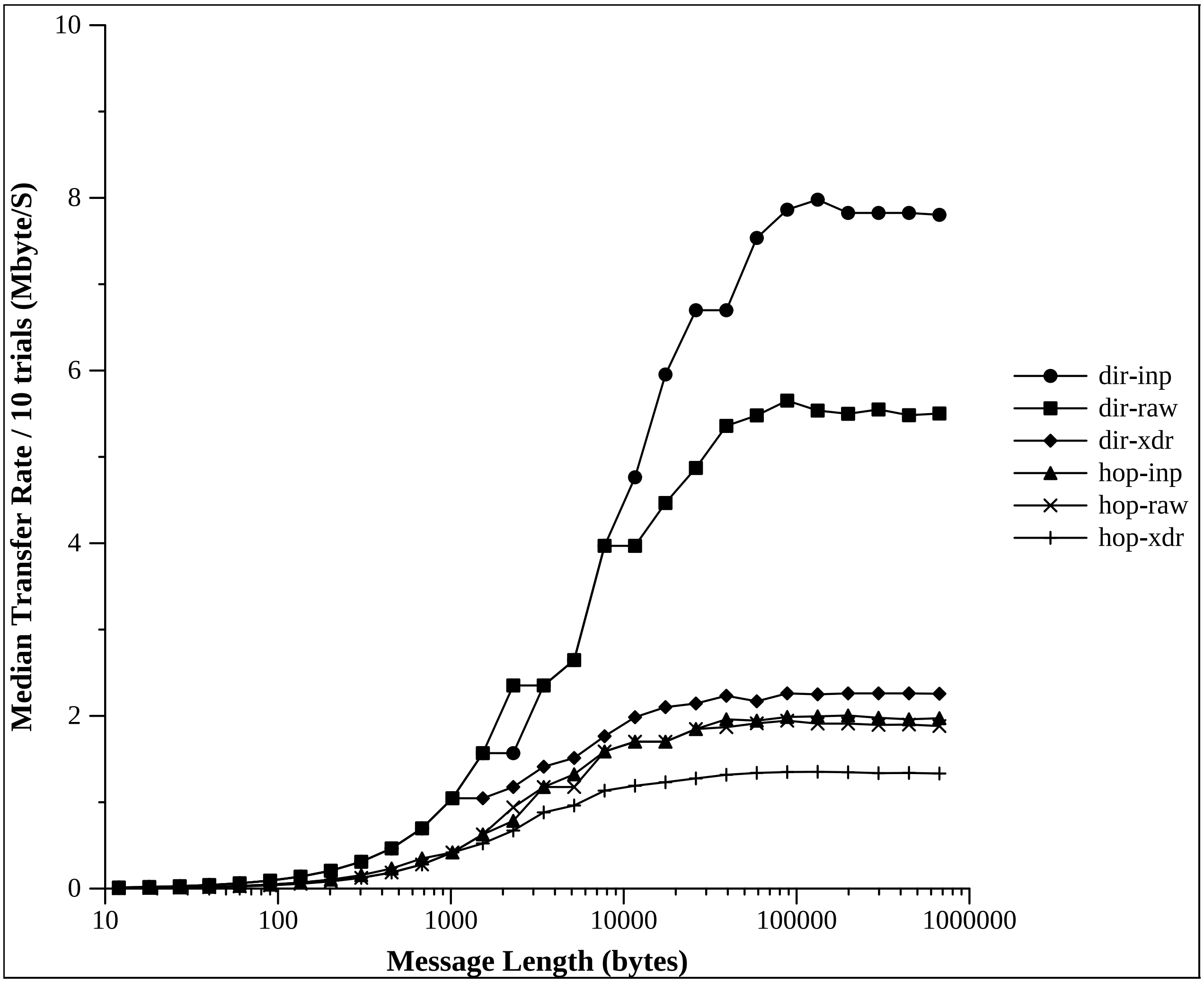

Figure 1: PVM message bandwidth versus size.

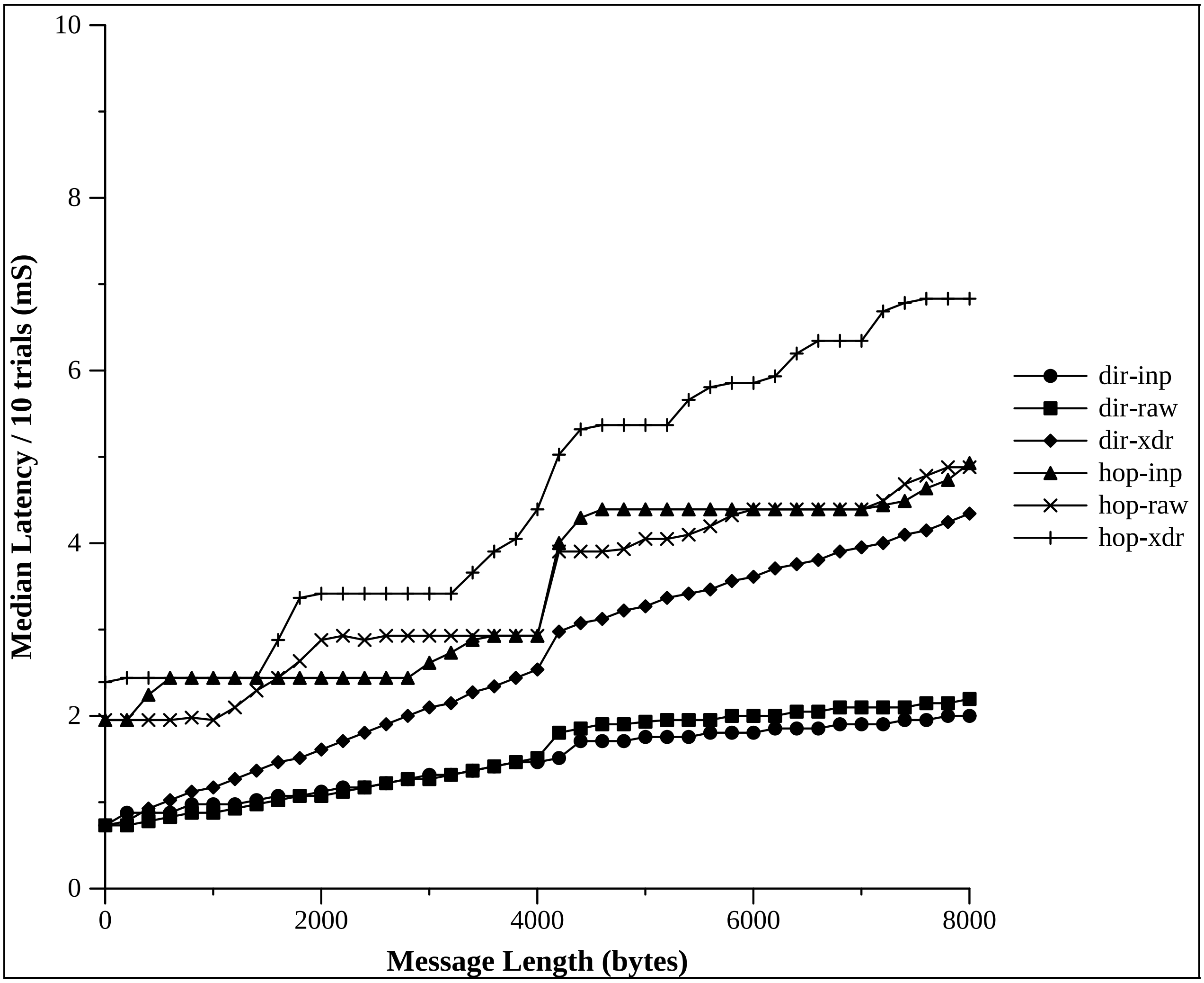

Figure 2: PVM message latency versus size.

```
A(1) = localmax1
A(2) = localmax2
root = 0
call pvmfreduce( PvmMax, A, 2, REAL8, msgtag, mygroup, root, info )
if ( me .eq. root ) then
   globalmax1 = A(1)
   globalmax2 = A(2)
endif
```

If all the group members need to know the result then the root task can broadcast the information to them.

Optionally, users can define their own global operation function to be used inside reduce. The PVM source distribution includes an example using a user defined function. The first argument in pvm_reduce() is a pointer to a function. A user can just substitute his own function. No additional PVM functions are required to define the user function, unlike MPI.

The pvm_reduce() function is built on top of the point-to-point routines and supports all the basic data types supported in point to point PVM messages.

As in pvm_reduce(), all members of the group must call pvm_gather() with consistent arguments. In particular a root must be specified. At the end of the gather the root task will have concatenated the data from all group members including itself into a single vector. The data is concatenated in rank order, as is done in MPI.

The use and syntax of pvm_gather() is illustrated in the following example where the PVM task IDs for the group members are collected in order into a vector.

```
call pvmfmytid(data)
call pvmfgather( result, data, 1, INTEGER4, msgtag, group, root, info)
```

After this call the root task has a result vector containing: task ID for group member 0, task ID for group member 1, etc. As in MPI, the result vector is significant only on the root task, all the other tasks can use a dummy argument for `result`.

The pvm_scatter() operation is the inverse of the gather operation. The root starts out with a large vector containing equal size pieces destined for individual group members. At the end of the scatter all group members have their piece of the vector. So for example, to scatter the previous task ID result back out to the group members (assuming `result` is still a dummy argument for every task except the root:

```
call pvmfscatter( data, result, 1, INTEGER4, msgtag, group, root, info)
```

After this call every task including the root has one integer in data. This integer is the same task ID that the task put into the pvm_gather().

Typically gather and scatter operations are used to gather data from a group of tasks, modify this data using some global information or information that requires all the data, and then scattering the modified data back out to the tasks.

4 Extensible Services

The PVM system is not only used by programmers who wish to construct parallel programs, but systems builders who are interested in the issues related to building distributed systems. To aid systems builders we have exposed certain functionalities of PVM, allowing systems programmers to easily extend these base functionalities. We would like to emphasize that the normal PVM programming interface for applications programmers is unaffected by these extensions.

New interfaces in version 3.3 allow PVM tasks to take over functions normally performed by the pvmds: Starting hosts and tasks, and making scheduling decisions. These interfaces allow the PVM system to be extended without modifications to the source code; this can be an important labor-saving device for researchers wanting to integrate their additions to PVM and distribute the resulting code. The tasks can be *registered* dynamically while the virtual machine and applications are running, which aids in debugging the additions, and all communication is done via normal PVM messages.

The pvmds were modified to allow them to receive messages from arbitrary tasks (tasks of other pvmds). A new entry point in the pvmd, `schentry()`, serves all three new interfaces.

4.1 Resource Manager

A *resource manager* (RM) is responsible for making task and host scheduling (placement) decisions. The simple schedulers embedded in the pvmd handle many common conditions, but require the user to explicitly place program components in order to get the maximum efficiency. Using knowledge not available to the pvmds, such as host load averages, a RM can make more informed decisions automatically. For example, when spawning a task, the RM could pick the host in order to balance the computing load. Or, when reconfiguring the virtual machine, the RM could interact with an external queuing system to allocate a new host.

The number of RMs registered can vary from one for an entire virtual machine to one per pvmd. The RM running on the master host (where the master pvmd runs) manages any slave pvmds that don't have their own RMs. A task connecting anonymously to a virtual machine is assigned the default RM of the pvmd to which it connects. A task spawned from within the system inherits the RM of its parent task.

If a task has a RM assigned to it, service requests from the task to its pvmd are routed to the RM instead. Messages from the following libpvm functions are intercepted:

Libpvm function	Default Message	RM Message
`pvm_addhost()`	TM_ADDHOST	SM_ADDHOST
`pvm_delhost()`	TM_DELHOST	SM_DELHOST
`pvm_spawn()`	TM_SPAWN	SM_SPAWN

Queries also go to the RM, since it knows more about the state of the virtual machine:

Libpvm function	Default Message	RM Message
`pvm_config()`	TM_CONFIG	SM_CONFIG
`pvm_notify()`	TM_NOTIFY	SM_NOTIFY
`pvm_task()`	TM_TASK	SM_TASK

The call to register a task as a RM (`pvm_reg_rm()`) is also redirected if a RM is already running. In this way the existing RM learns of the new one, and can grant or refuse the request to register.

Using messages `SM_EXEC` and `SM_ADD`, the RM can directly command the pvmds to start tasks or reconfigure the virtual machine. On receiving acknowledgement for the commands, it replies to the client task. The RM is free to interpret service request parameters in any way it wishes. For example, the architecture class given to `pvm_spawn()` could be used to distinguish hosts by memory size or CPU speed.

4.2 Hoster

A *hoster* is a task that starts slave pvmd processes on command from the master pvmd. Normally, the master pvmd uses the `rsh` program or the `rexec()` function (depending on whether a password is used) to start the pvmd process on a new slave host. Over the socket created by `rsh`, the master and slave have a short dialog to bootstrap the pvmd-pvmd message drivers, which allows the slave to be brought up the rest of the way via normal PVM messages. The hoster allows one to alter this mechanism for adding new hosts to the virtual machine. This might be useful for dealing with systems where additional security is needed or where the new host, say an MPP, does not support standard rsh/rexec interfaces.

If a hoster task is registered (using `pvm_reg_hoster()`) with the master pvmd when a host-add is requested (a `DM_ADD` message is received), the master pvmd sends a `SM_STHOST` message to the hoster and waits for a `SM_STHOSTACK` message in reply to complete the operation.

The bootstrap protocol between the master pvmd and slave is designed so that the hoster need understand as little as possible about the protocol. The hoster is sent a list of hosts and commands to run. It runs the commands and returns output from them to the master pvmd, which does the parsing. The remainder of the startup is always done by the master pvmd. The message formats are:

```
SM_STHOST
            int nhosts                      Number of hosts
            {
                      int tid               Of host
                      string options        From hostfile so= field
                      string login          In form [username@]hostname.domain
                      string command        To run on remote host
            } [nhosts]

SM_STHOSTACK
            {
                      int tid               Of host
                      string status         Line of output from slave or error code
            } []                            Count is implied
```

Host file startup options `pw` (password) and `ms` (manual startup) were combined into a single option, `so=` (startup option). The new forms are `so=pw` and `so=ms`. The value of `so` can be set to any string, and is null by default. This allows information specific to a custom hoster to be passed from the host file to the hoster without being processed by the pvmd. The default "hoster" (built into the pvmd) understands only *pw* and *ms*. A replacement hoster might accept those, or expect completely different options.

4.3 Use in Condor

The resource manager and hoster interfaces were created in cooperation with members of the Condor project [10]. They are used together by Condor. There was initially a single interface for both functions, but the two were logically separated because either part is generally useful.

In Condor, the scheduler is the boss of everything else. If an application attempts to reconfigure the virtual machine, PVM calls Condor through the resource manager interface to determine whether another host can be allocated, and if so, which one. The names of hosts requested refer to classes of machines, instead of specific hosts.

Pvmds and user processes can run under a borrowed login assigned to Condor, using cycles from idle workstations. If the workstation owner should return, the processes must be stopped immediately and cleared off the host in a timely manner; any temporary files must be moved as well. The hoster interface allows Condor to start slave pvmds, necessary because they are run under a Condor-owned login.

4.4 Tasker

A *tasker* is a PVM task that starts (*execs*, and is the parent of) other tasks. The tasker facility allows a secific PVM task to control the creation and execution of all tasks in the system. This could be useful when newly spawned tasks need to be under the control of some other process for debugging or performance monitoring reasons. In general, a debugger is a process that controls the execution of other processes — can read and write their memories and start and stop instruction counters. On many versions of Unix, a debugger must be the direct parent of any processes it controls. This is becoming less common with growing availability of the attachable *ptrace* interface.

PVM already provided a simple debugger interface: If a task is spawned (via the pvm_spawn call) with flag `PvmTaskDebug` set, the pvmd execs a debugger program instead of the actual task executable. The debugger arguments are the executable file and arguments for the task. The debugger can then start the task to be debugged.

The tasker interface coexists with this simple debugger interface, but is fundamentally different for two reasons: First, the tasker cannot be enabled or disabled by spawn flags, so it is always in control. Second, all tasks running under a pvmd (during the life of the tasker) may be children of a single tasker process. With `PvmTaskDebug`, a new debugger is necessarily started for each task.

If a tasker is registered (using `pvm_reg_tasker()`) with a pvmd when a `DM_EXEC` message is received to start new tasks, the pvmd sends a `SM_STTASK` message to the tasker instead of calling `execv()`. No *SM_STTASKACK* message is required, closure comes from the task reconnecting to the pvmd as usual. The pvmd doesn't get `SIGCHLD` signals when a tasker is in use, because it's not the parent process of tasks, so the tasker must send notification of exited tasks to the pvmd in a `SM_TASKX` message.

The message format is:

`SM_STTASK`

`int tid`	Of task
`int flags`	As passed to `pvm_spawn()`
`string path`	Absolute path of the executable
`int argc`	Number of args for process
`string argv[argc]`	Arg strings
`int nenv`	Number of environment for process
`string env[nenv]`	Environment strings

The tasker interface is the result of collaboration with the Paradyn group [9]. We hope that others will take advantage of it as well, to allow cleaner integration of their systems.

4.5 Implementation

We will briefly describe the implementation details of these new features. For a more indepth description see [?, 11].

A new class of system messages (`SM_XXX`) was defined, to be sent between pvmds, resource managers, hosters and taskers, as well as client tasks of a resource manager.

A new entry point in the pvmd, `schentry()`, serves messages of the SM class for all three new interfaces. The pvmd was modified to allow it to receive messages from arbitrary tasks (tasks of other pvmds). The pvmds don't usually communicate with foreign tasks (those on other hosts). The pvmd has message reassembly buffers for each foreign pvmd and each task it manages. Reassembly buffers for foreign tasks would be too complicated. To free up the reassembly buffer for a foreign task (if the task dies), the pvmd would have to request notification from the task's pvmd, causing extra communication. For the sake of simplicity the pvmd local to the sending task serves as a message repeater. The message is reassembled by the task's local pvmd as if it were the receiver, then forwarded all at once to the destination pvmd, which reassembles the message again. The source address is preserved, so the sender can be identified. Libpvm maintains dynamic reassembly buffers, so messages from pvmd to task do not cause a problem.

The existing fault recovery mechanisms were mostly adequate to serve the new system tasks. For example, if `pvm_addhosts()` is called to add hosts to the virtual machine and the hoster task fails while starting the new pvmds, the master pvmd enters the normal task-exit cleanup routine, which cancels the startup operation and returns error code `PvmDSysErr` for each host in the result vector. Likewise, if the tasker fails, the pvmd can find and terminate the tasks for which it was responsible. The resource manager operations are not currently recovered, because it's not clear what action to take.

4.6 Caveat

The features presented in this section are undoubtedly not for the casual PVM user, but for tool developers. Collaboration with other research groups and the addition of these features has had a positive effect on PVM. The protocols are now conceptually "cleaner" than they were before. We have shown that the system can be made more powerful and general without increasing the amount of code, by making it dynamically extensible. The result has been directly useful to all the projects involved, and we hope it will have even more widespread application as the interfaces become more stable and other researchers take advantage of them.

5 MPP and Shared Memory Support

PVM Version 3 is designed so that message passing calls of a specific system can be compiled into the source. This allows the fast, native, message-passing for a particular system to be realized by the PVM application. Messages between two nodes of a multiprocessor use its native message-passing routines, while messages destined for an external host are routed via the user's PVM daemon on the multiprocessor. The

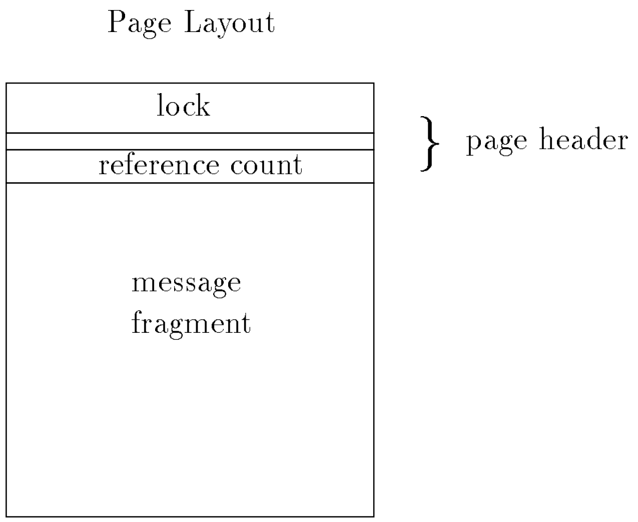

Figure 3: Structure of a PVM page.

MPP subsystem of PVM consists of a daemon that manages the allocation and deallocation of nodes on the multiprocessor. This daemon is implemented in terms of PVM 3 core routines. The second part of the MPP port is a specialized libpvm library for this architecture that contains the fast routing calls between nodes of this host. On shared-memory systems the data movement can be implemented with a shared buffer pool and lock primitives.

5.1 Shared Memory

The shared-memory architecture provides a very efficient medium for processes to exchange data. In our implementation, each task owns a shared buffer created with the shmget() system call. The task ID is used as the "key" to the shared segment. A task communicates with other tasks by mapping their message buffers into its own memory space.

To enroll in PVM, the task first writes its Unix process ID into pvmd's incoming box. It then looks for the assigned task ID in pvmd's pid→tid table. The message buffer is divided into pages, each holds a part of the message or one fragment (Figure 3). PVM's page size can be a multiple of the system page size. Each page has a header, which contains the lock and the reference count. The first few pages are used as the incoming box, while the rest of the pages hold outgoing fragments (Figure 4). To send a message, the task first packs the message body into its buffer, then delivers the message header, which contains the sender's TID and the location of the data, to the incoming box of the intended recipient. When pvm_recv() is called, PVM checks the incoming box, locates and unpacks the messages (if any), and decreases the reference count so the space can be reused. If a task is not able to deliver the header directly because the receiving box is full, it will block until the other task is ready.

Inevitably some overhead will be incurred when a message is packed into and unpacked from the buffer, as is the case with all other PVM implementations. If the buffer is full, then the data must first be copied into a temporary buffer in the process's private space and later transferred to the shared buffer.

Memory contention is usually not a problem. Each process has its own buffer and each page of the buffer has its own lock. Only the page being written to is locked, and no process should be trying to read from this page because the header has not been sent out. Different processes can read from the same page without interfering with each other, so multicasting is efficient (tasks do have to decrease the counter afterwards, resulting in some contention). The only time contention occurs is when two or more processes trying to deliver the message header to the same process at the same time. But since the header is very short (16 bytes), such contention should not cause any significant delay.

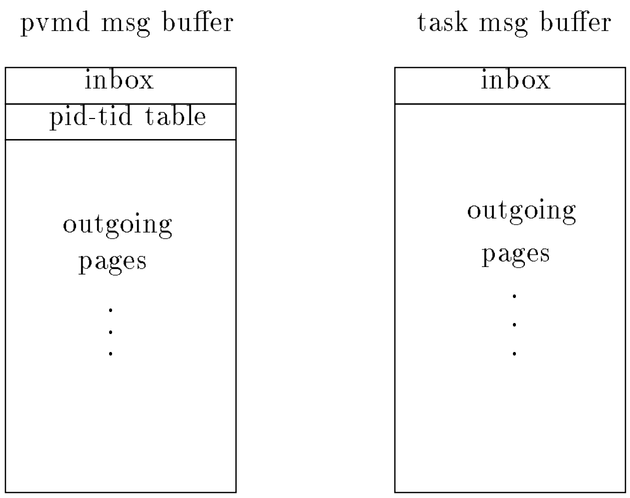

Figure 4: Structures of shared message buffers.

6 New PVM Tracing Features

A new tracing feature in PVM now directly supports the tracing of PVM programs. In this section we describe the tracing feature and present a tool, XPVM, which displays PVM trace information. Tracing is a popular way of debugging parallel programs. There have been many systems which support program tracing for debugging [12]. More recently systems such as Pablo [13], Paragraph [8], PICL [7], Bee [3], and Xab [1, 2] have been developed to aid the parallel programmer. Pablo, Paragraph, and Bee are intended to be general tools for displaying trace information for parallel programs. PICL is a communication library for multicomputers which is portable and generates tracefiles. Xab is a specific tool for tracing PVM programs. Xab will generate and display traces of PVM programs. XPVM and the new tracing feature of PVM are based on the previous work done on Xab.

6.1 How to Use Tracing

When tracing is turned on, events are generated for PVM calls. These events describe a call to the PVM library. For instance, a call to `pvm_send()` will generate two events, pvm_send0 and pvm_send1. The pvm_send0 event is generated upon entry into the `pvm_send()` routine. The pvm_send0 event contains a timestamp, the task, and message type used when the `pvm_send()` routine was called. The pvm_send1 event is generated at the end of the `pvm_send()` call. It contains a timestamp and the return value of the send call. Most events come in pairs. This allows one to determine the amount of time spent within a call.

An easy way to trace a PVM program is to simply start it from the PVM console with the trace option:

```
pvm> spawn -@ calc
```

This will cause the calc program to be spawned with tracing turned on. Trace events will be sent to the PVM console and displayed there. Tracing of particular routines can be turned on or off by the PVM console. For instance, if only calls to `pvm_barrier()` are of interest then the following commands in the console will activate tracing for only the barrier calls:

```
pvm> trace - *
pvm> trace + pvm_barrier
```

The first command turns off tracing for all routines. The second command turns on tracing for the barrier routine.

Another way to use tracing is to activate it from within a PVM program. When spawning a task, tracing for the spawned tasks can be activated by using the PvmTaskTrace flag in the `pvm_spawn()` call. This flag

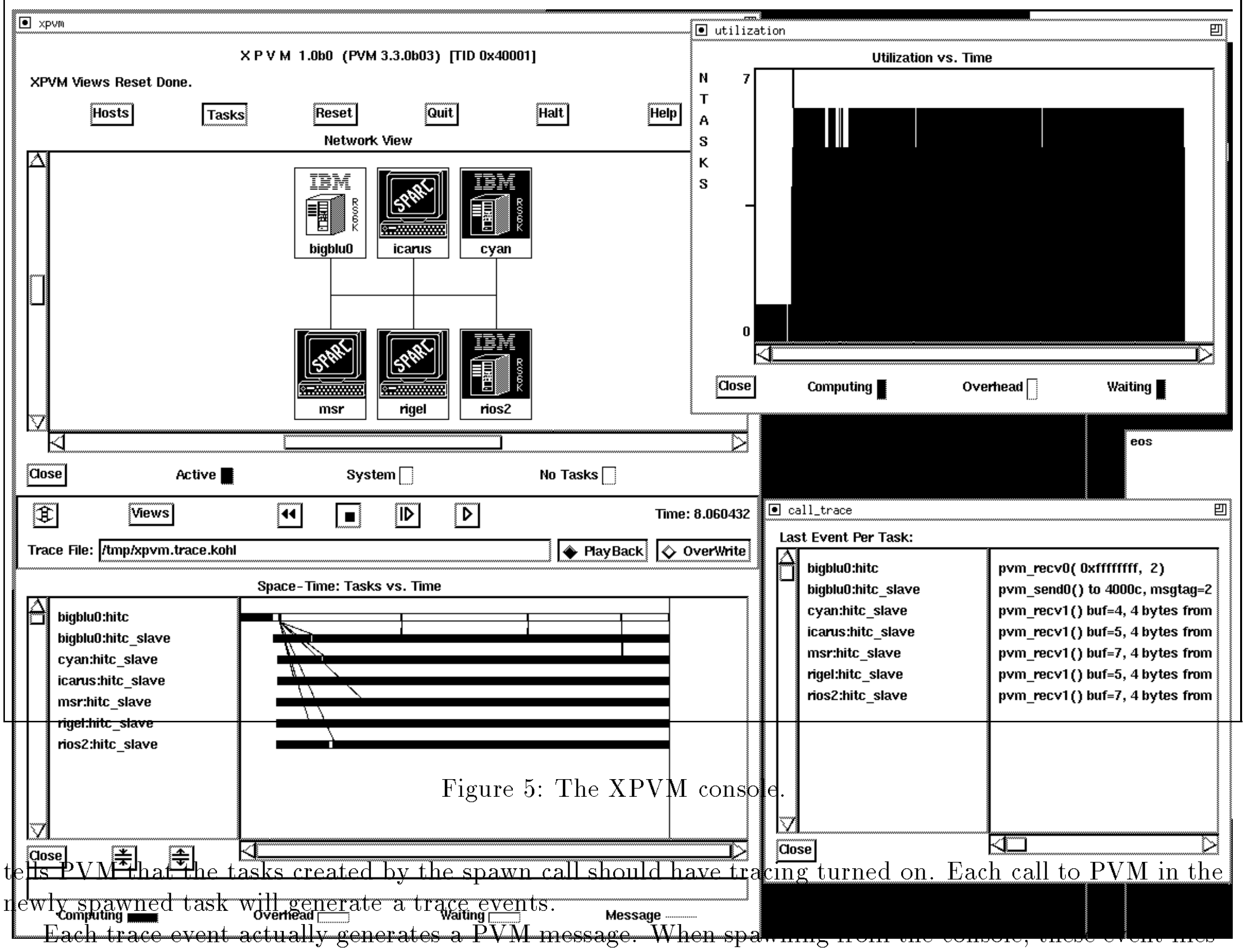

Figure 5: The XPVM console.

tells PVM that the tasks created by the spawn call should have tracing turned on. Each call to PVM in the newly spawned task will generate a trace events.

Each trace event actually generates a PVM message. When spawning from the console, these event messages are sent to the console which displays their contents. Using the `pvm_setopt()` routine, the destination for these trace events can be set to any PVM task. Similarly the `pvm_settmask()` routine can be used to set a mask indicating which PVM routines should generate trace events. More details on how to control tracing can be found in the PVM man pages.

6.2 The XPVM Console

Although PVM provides the flexibility for the programmer to control which events are generated and where event messages will be sent, most programmers do not need this flexibility. A more pleasant way of controlling and displaying events is through the XPVM console. XPVM provides a graphical interface to the functions of the PVM console (i.e. adding hosts, etc) as well as displaying PVM trace events. Figure 5 shows the XPVM in action.

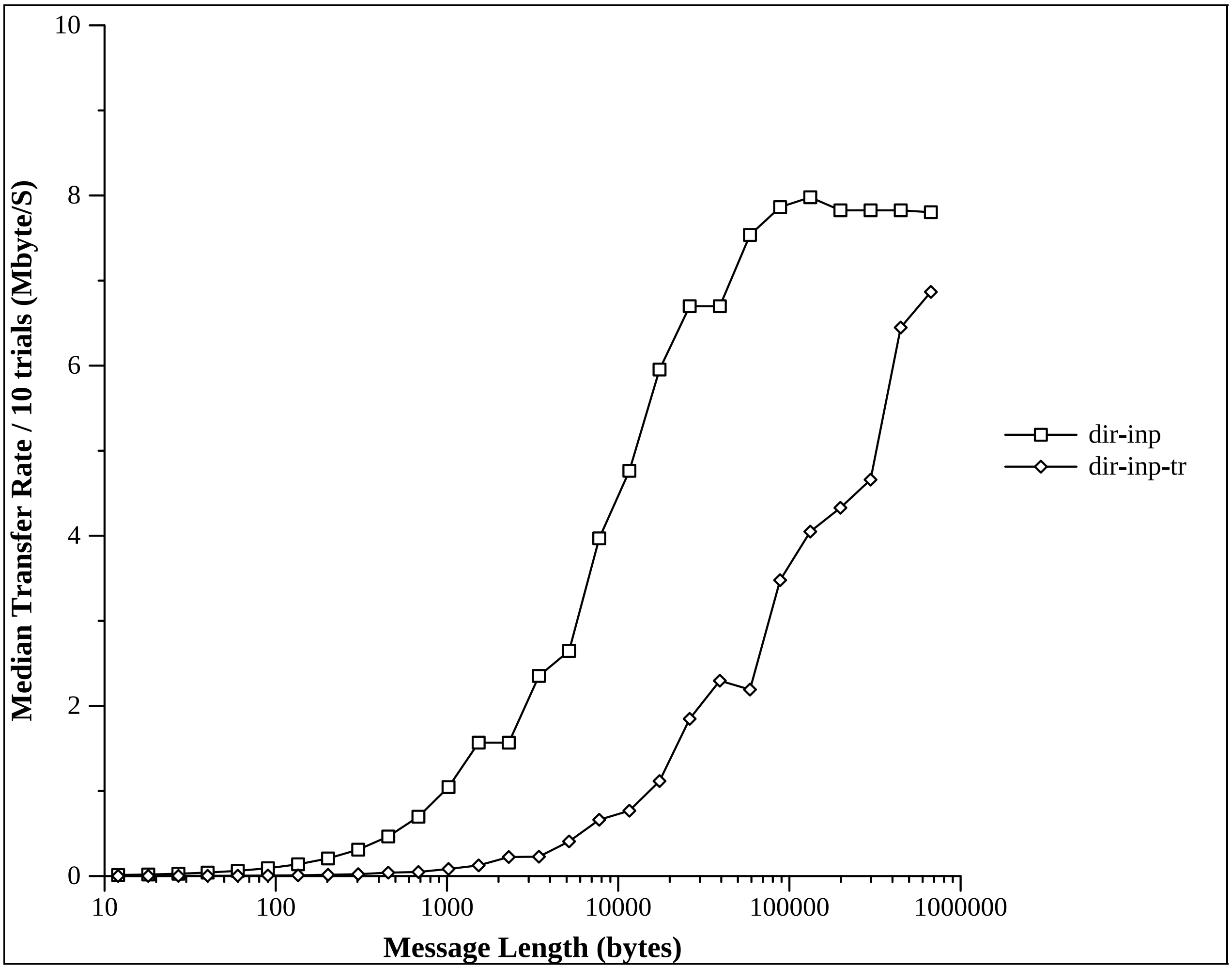

Figure 6: Effects of tracing on PVM message bandwidth.

6.3 Tracing Overheads

Since the tracing facilities of PVM generate extra traffic in the network it is important to realize that this traffic will perturb the runtime characteristics of the program. In a shared environment one must expect that other external factors such as varying machine load and network traffic will also perturb the computation from run to run. To show how tracing may alter the runtime characteristics of a PVM program we present the bandwidth experiment shown in Section 2.1, this time we show the bandwidth versus message size with tracing turned on for *all* PVM calls and with out any tracing. (See Figure 6.) In some sense this experiment shows the worst case tracing behavior since the program is only sending messages and doing no computation. In the case of a real PVM application we would expect the tracing to have somewhat less of an effect.

In terms of bandwidth, tracing does not have an extremely detrimental effect, however, as Figure 7 shows, the effect of tracing on latency is considerable. The main reason for the increased latency is the extra processing (message sending) PVM must do before and after each PVM call. This effect will most likely be reduced when we add buffering of trace events to the PVM trace facility. Currently each trace event generates a separate PVM message. By buffering multiple events and sending them in a single message, the per event overhead should be reduced. We have shown this to be the case in the Xab tracing tool for PVM.

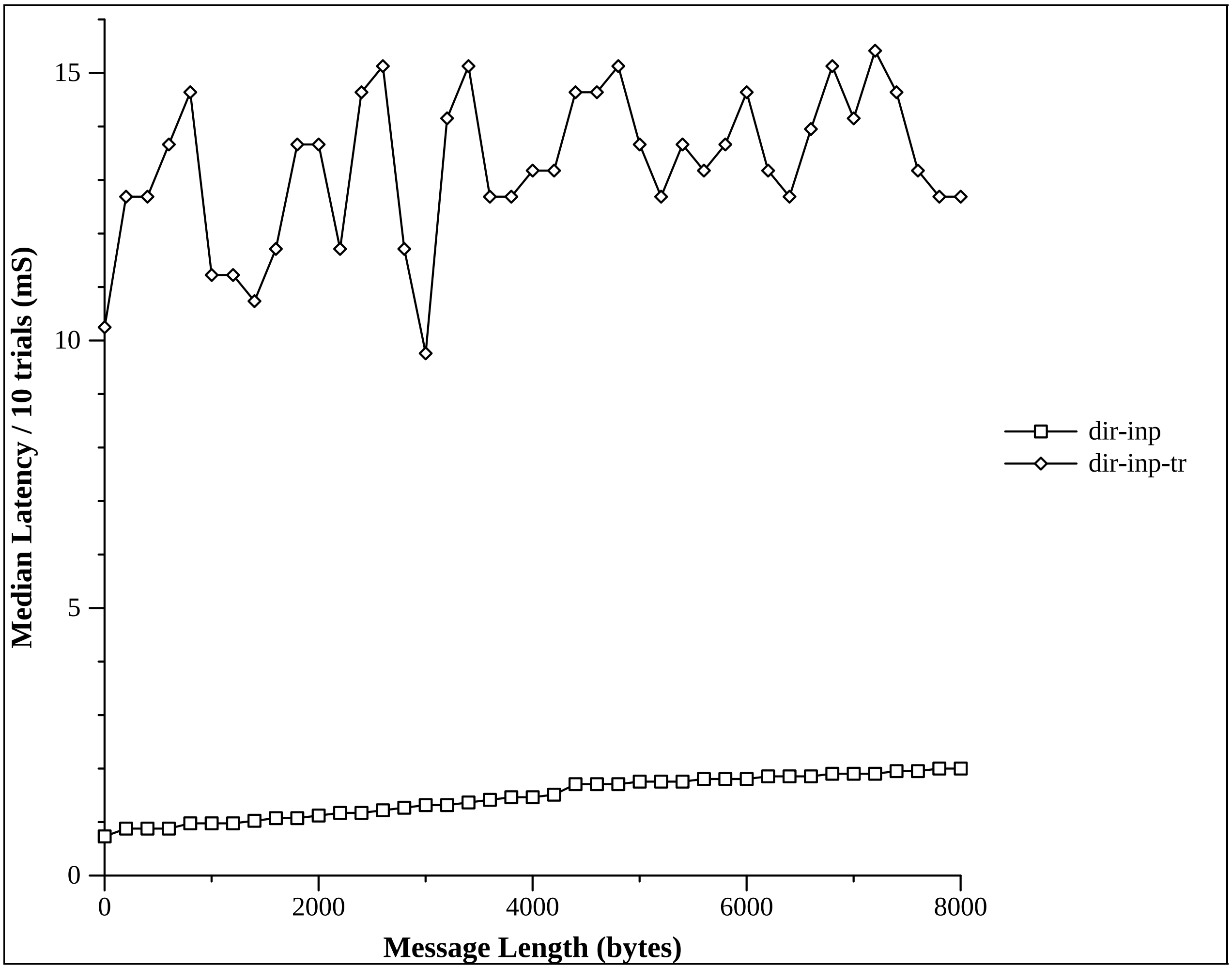

Figure 7: Effects of tracing on PVM message bandwidth.

7 Future Research

The PVM project is an ongoing research effort aimed at advancing the state of the art of heterogeneous concurrent computing via the design and development of experimental software systems. The PVM system is therefore constantly evolving, and improving in terms of both functionality and performance, while retaining the concise and manageable computing model and programming interface. In this section, we briefly describe two research initiatives that have been recently undertaken within the context of the PVM project.

7.1 Generalized Distributed Computing with PVM

In the evolution of the PVM system for cluster and heterogeneous distributed computing, high-performance scientific applications have thus far been the main technical drivers. The computing model, as well as specific software features have been influenced by the requirements of scientific algorithms and their parallel implementations. We believe that by extending this infrastructure along certain important dimensions, the PVM system will be able to cater to a much larger class of application categories. The goal therefore is to enable generalized distributed computing within heterogeneous networked frameworks, i.e. to evolve both a conceptual model and a software infrastructure that integrally support high performance scientific applications as well as other general purpose applications, including, but not limited to, distributed teleconferencing and groupware systems, heterogeneous and multi-databases, high speed on-line transaction processing and geographically distributed information systems. This enhancement to PVM, termed the General Distributed Computing (GDC) layer has recently been undertaken; GDC facilities will consist of infrastructural support for the required operations, and are briefly described below

- Parallel IO facilities: the GDC layer extends PVM functionality by providing support for distributed and parallel input and output to disk files as well as for terminal interaction. The standard Unix file semantics are retained to the extent possible; in addition, facilities for shared but non-conflicting reading and writing, using a variety of different interleaving and consistency semantics are provided. In essence, exclusive, independent, interleaved, and serialized access are supported by the parallel IO subsystem. In addition, support exists for data compression and encryption, as well as for file shadowing — a valuable feature for reliability.

- Synchronization and Locking: the GDC subsystem provides facilities for mutually exclusive access to resources. The model permits these resources to be application dependent — the provided primitives allow for locking an abstract resource identified by a string valued identifier and an integer. Thus applications may establish a convention according to the nature of their requirements and utilize the GDC facilities without any loss of generality or functionality, but with substantial flexibility. For example, to implement record-level file locking, applications may request a lock on the abstraction identified by the filename and record number. In addition to efficient locks, the GDC subsystem also incorporates certain deadlock detection heuristics and, based on option switches, will either attempt recovery or return control to the user after setting locks to a "safe" state.

- Client-Server Support: The native PVM facilities are geared towards asynchronous, communicating processes, and do not provide sufficiently high-level access to applications using the client-server paradigm. The GDC subsystem alleviates this deficiency by permitting server components of applications to **export** services that are identified by symbolic names, and for client components to **invoke** these services in a location-transparent, heterogeneous, and efficient manner. These features comprise a significant extension of the standard remote procedure call model in that (1) PVM and GDC automatically locate remote services; (2) support for load balancing, using multiple servers, is provided; (3) invocation semantics may be either procedure-argument based or message based; and (4) a certain level of failure resilience is built into the system.

- Transaction Processing: Design and initial implementation and testing efforts are in progress for a distributed transaction facility in the GDC layer. This facility provides the normal transaction processing constructs including beginning and ending transactions, aborting transactions, and nested transactions. These features are consistent with the usual atomicity, consistency, isolation, and durability semantics of traditional database systems. However, since the GDC layer facilities may be used in conjunction with standard PVM message passing features, certain enigmatic situations arise. For example, if a transaction's scope includes sending and receiving of messages, the correct actions in the case of an abort are unclear, as restoring the system to a previously valid state is complex and possibly intractable. We are exploring several alternatives and will proceed incorporating these features into the GDC layer as soon as "correct" semantics are decided upon.

Our preliminary experiences with the GDC subsystem indicate that enhancing the PVM features to support generalized distributed computing, with specific focus on commercial, business, and database applications, is very valuable and is being increasingly accepted and adopted. Our performance measurements have also been very encouraging: during testing, overheads of a few to several tens of milliseconds were observed for most of the facilities outlined above, such as locking, synchronization, and parallel input and output with shadowing.

7.2 Threads Based Parallel Computing

Thus far, cluster software systems have used a process-based model of parallelism, as in distributed memory multiprocessors but on the opposite end of the spectrum from loop-level parallelism common in vector supercomputers. To enhance functionality as well as performance, we are investigating a threads-based parallelism model within PVM, that compromises between the large granularity of processes and the fine granularity in loops. Threads, or lightweight processes, are essentially multiple sequences of control within a single process that share portions of a common address space. A subroutine (or collection of subroutines) is associated with each thread, and these are multiplexed on the basis of priorities and status — thus providing an effective means of context switching with minimal overheads. Several stand-alone threads packages are available, and operating systems are incorporating native threads into their repertoire — and it is anticipated that threads will be a standard feature of most software environments in the near future.

Figure 8 depicts the architecture of the PVM threads system under development. From a program development point of view, threads-based cluster computing will differ minimally from the existing process oriented paradigm. In the PVM-threads system, programs export threads, thereby establishing a mapping between a symbolic name and a subroutine address. PVM processes are initiated as in the current scenario, but subsequently spawn multiple threads, each of which, when activated, is assigned a unique thread identifier. The run time system spawns threads based on user-supplied options as well as relative processing speeds of machines in a cluster — the smaller granularity of threads, when coupled with load-based placement, allows for more control in load balancing. Once spawned, threads communicate via explicit message passing calls — in reality however, messages are exchanged only when communicating threads are situated in distinct processes (local communication transparently takes place via shared memory).

From the functional viewpoint, such a threads-based model offers two main advantages. First, data decomposition based on smaller granularity can be implemented without the loss of efficiency that a process-based model would incur. This is especially important in applications such as tree-search algorithms, integer computations, and database query systems, where the amount of computation between communication phases tends to be small. Second, such a paradigm is natural for client-server computing. Services can be exported using the thread registration mechanism, and invoked via functions akin to remote procedure calls. This facility is very useful for non-numeric computing applications, especially those in the database and transaction processing domain.

In terms of performance enhancement, threads provide tremendously increased potential for overlapping computation and communication. Within a processor, the typical communication-computation-communication cycle of parallel processing results in idle periods when a process based model is used. However, with a threads-based model, one thread can be productively utilizing the CPU when another is communication bound or blocked waiting for data to arrive. In preliminary tests with the threads interface to PVM, performance improvements of up to 35% were attained on several standard algorithms, without any other external optimizations. For more information on the threads based implementation of PVM see [14].

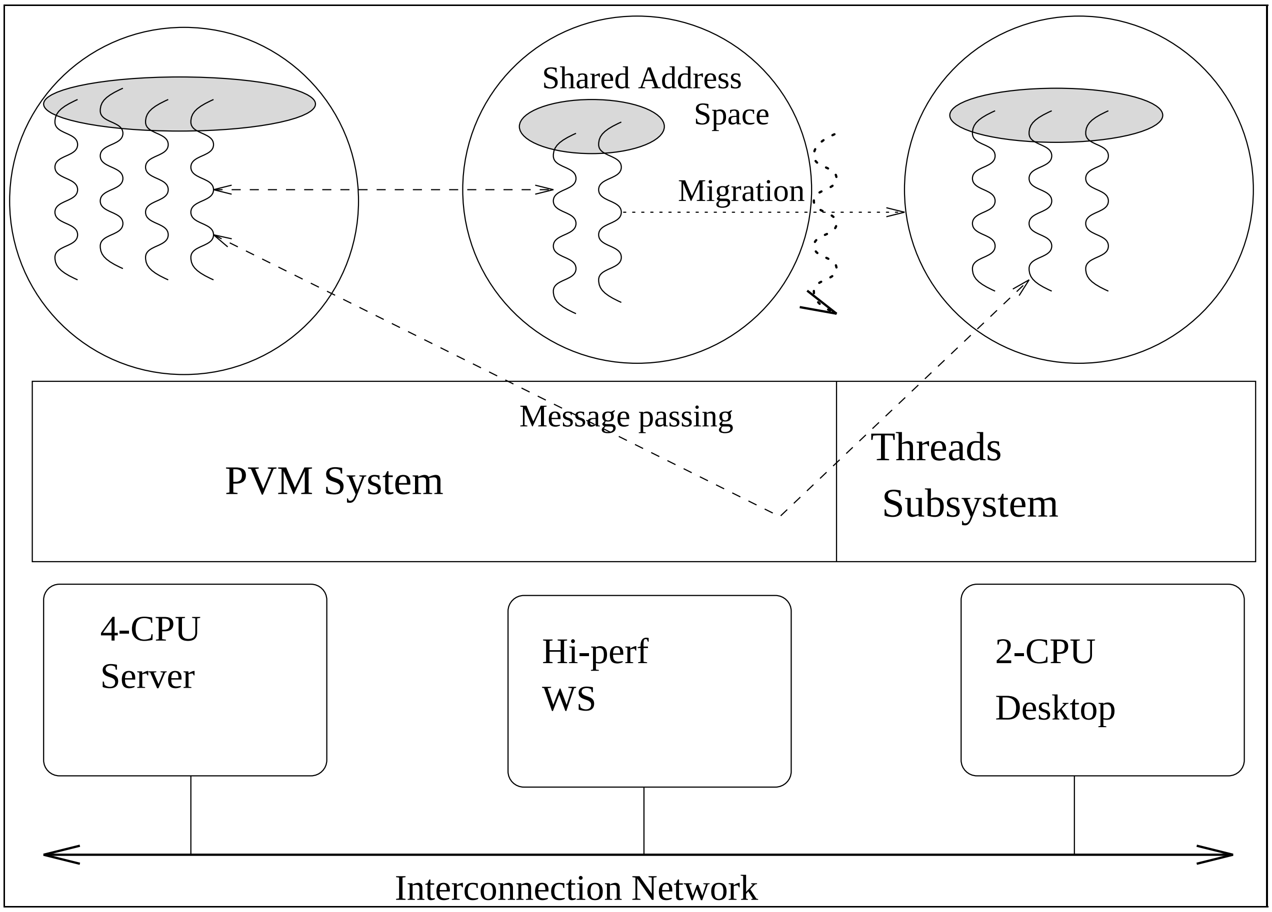

Figure 8: PVM Threads

8 Summary

PVM is the mainstay of the heterogeneous concurrent computing project that, at the time of writing, involves over a dozen researchers and four academic and research institutions. A number of factors, including simplicity of design, the natural and general computing model supported, robustness of implementation, ease of use, high degree of portability, and uncommon levels of support, have contributed to the tremendous popularity of PVM. It is estimated that over 10,000 individuals or installations have retrieved the software, and about 20 to 25% are actively using PVM in their everyday computing, both for experimentation as well as for production quality work. In addition, PVM is increasingly becoming a platform for education and computer science research, as witnessed by the scores of third-party extensions to PVM for load balancing, process migration, profiling, performance optimization, etc.

The PVM system has evolved through three major versions (and numerous patchlevel releases) in the five years it has been in existence, even though the original basic design and computing model has been retained. In this paper, we have described some of the major and significant evolutionary features in PVM, as manifested in release version 3.3 of the system. These enhancements may be categorized as those pertaining to performance, functional enhancements, and auxiliary toolkits. In terms of performance, communication rates — the most critical aspect in network computing — have been significantly improved, to the extent that they are near the achievable maxima for various networks. Further, in response to the increasing prevalence of shared memory multiprocessors, communication optimizations for such machines has resulted in performance levels several times that of previous versions of the system. Other performance improvements are less dramatic though no less important, and represent the results of code analysis and program tuning efforts.

PVM functionality has been greatly improved in version 3.3. Most noteworthy are the new suite of collective communication routines that are required by many application algorithms. The design and implementation of a scheduling *interface*, as opposed to a hardwired scheduling scheme, enables flexible and optimal scheduling, while achieving a clean separation of mechanism and policy. In terms of auxiliary tools, the latest version of PVM has both a significantly enhanced textual console as well as an integrated graphical interface toolkit. The latter, called XPVM, contains an administrative interface for virtual machine and process management, and also provides tracing and profiling facilities appropriate for operation in a general purpose networked environment. Finally, ongoing efforts of a more investigative and exploratory nature seek to complement system capabilities by providing multi-threading support, parallel I/O facilities, and features to support generalized distributed computing with a view to firmly establishing PVM as the de-facto standard for mainstream parallel and distributed computing.

References

[1] A. Beguelin, J. Dongarra, A. Geist, and V. Sunderam. Visualization and debugging in a heterogeneous environment. *IEEE Computer*, 26(6):88–95, June 1993.

[2] Adam Beguelin. Xab: A tool for monitoring pvm programs. In *Workshop on Heterogeneous Processing*, pages 92–97, Los Alamitos, California, April 1993. IEEE Computer Society Press.

[3] Bernd Bruegge. A portable platform for distributed event environments. *ACM SIGPLAN Notices*, 26(12):184–193, December 1991. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging.

[4] Douglas Comer. *Internetworking with TCP/IP*. Prentice Hall, 2nd edition, 1991.

[5] Message Passing Interface Forum. MPI: A message-passing interface standard. Computer Science Dept. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, April 1994. (To appear in the International Journal of Supercomputer Applications, Volume 8, Number 3/4, 1994).

[6] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[7] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A machine-independent communication library. In J. Gustafson, editor, *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pages 565–568, P.O. Box 428, Los Altos, CA, 1990. Golden Gate Enterprises.

[8] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[9] J. Hollingsworth, B. Miller, and J. Cargille. Dynamic program instrumentation for scalable performace tools. In *Proceedings of 1994 Scalable High Performance Computing Conference*, Knoxville, TN, May 1994. to appear.

[10] M. Litzkow, M. Livny, and M. Mutka. Condor — A hunder of idle workstations. In *Proceedings of the Eighth Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[11] R. J. Manchek. The design and implementation of pvm version 3. Master's thesis, University of Tennessee, Department of Computer Science, Knoxville, TN 37909, May 1994. Also available as Computer Science Department Technical Report CS-94-232.

[12] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[13] Daniel A. Reed, Robert D. Olson, Ruth A. Aydt, Tara M. Madhyastha, Thomas Birkett, David W. Jensen, Bobby A. A. Nazief, and Brian K. Totty. Scalable performance environments for parallel systems. In Quentin Stout and Michael Wolfe, editors, *The Sixth Distributed Memory Computing Conference*, pages 562–569. IEEE, IEEE Computer Society Press, April 1991.

[14] V. S. Sunderam. Heterogeneous concurrent computing with exportable services. In Jack J. Dongarra and Bernard Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 142–151. SIAM Press, May 1994.