Recent Enhancements To Pvm

Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek and Vaidy Sunderam

International Journal of High Performance Computing Applications 1995 9: 108
DOI: 10.1177/109434209500900204

RECENT ENHANCEMENTS TO PVM

Adam Beguelin[1]
Jack Dongarra[2,3]
Al Geist[3]
Robert Manchek[2]
Vaidy Sunderam[4]

[1]SCHOOL OF COMPUTER SCIENCE,
CARNEGIE MELLON UNIVERSITY,
PITTSBURGH, PENNSYLVANIA 15213,
AND
PITTSBURGH SUPERCOMPUTING
CENTER
[2]UNIVERSITY OF TENNESSEE,
KNOXVILLE, TENNESSEE 37996-1301
[3]OAK RIDGE NATIONAL LABORATORY
BOX 2008, BUILDING 6012
OAK RIDGE, TENNESSEE 37831-6367
[4]EMORY UNIVERSITY
DEPARTMENT OF MATHEMATICS AND
COMPUTER SCIENCE
ATLANTA, GEORGIA 30322

Summary

This paper presents new features of PVM, a popular standard for writing parallel programs that 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. PVM's new tracing facility and a graphical console that utilizes this capability are described. Finally, we discuss future extensions to PVM now under investigation.

The International Journal of Supercomputer Applications, Volume 9, No. 2, Summer 1995, pp. 108–127

Introduction

The past several years have witnessed 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) (Geist et al., 1993) 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 lets programmers utilize a network of heterogeneous computers (some of which may be MPPs) as a single multicomputer. The 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 process, called pvmd, that resides on all the computers making up the virtual computer. pvmd is designed so that any user with a valid login can install it on a machine. A user who wishes to run a PVM application executes pvmd on one of the computers which, in turn, starts up pvmd 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 (version 3.3) and present an overview of future directions for PVM.

1 Point-to-Point Communication

In general, PVM programs consist of tasks that com-

municate via messages. A task is a basic unit of computation in PVM, a Unix process for instance. Prior to PVM 3.3 there had 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 using these routines.

The philosophy of PVM has always been to keep the user interface simple and easy to understand in order to improve performance, letting PVM do all the hard work underneath. For example, a user who has sent a message would like the data to arrive instantly at the destination. In reality this can never happen, since there is always some startup latency besides the time it takes to move the data. These overheads cannot be avoided but they can be masked by other work. Some message-passing interfaces, such as MPI (MPI: A message-passing interface standard), go to great lengths to supply many variants of send to allow the user several ways of managing explicitly the masking of the send overheads. This is a good approach if the goal is to provide 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 generally use only 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, in addition, 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 limited solely by the available memory on the machine.

The PVM communication model provides asynchronous blocking send, asynchronous blocking receive, and nonblocking receive functions. A blocking send returns as soon as the send buffer is free for reuse,

"The PVM communication model assumes that any task can send a message to any other PVM task and, in addition, 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."

and an asynchronous send does not depend on the receiver calling a matching receive before the send can return. A nonblocking 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 provide information about received messages.

The PVM model guarantees that message order is preserved between two tasks. For example, if task 1 sends message A to task 2 and then 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, 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 would be initialized by a call to pvm_initsend(). This step also had the effect of clearing any previous send buffer. Second, the message had to be "packed" into this buffer using any number and combination of pvm_pk*() routines. At this step PVM would take care of any data encoding needed for heterogeneity and build the buffer in fragments (required by the network protocols) so that the overhead of fragmenting a large buffer during transmission was avoided. Third, the completed message would be sent to another process by calling the pvm_send() routine.

There were several advantages to this three-step method. It let the user pack a message with several different pieces of data. A message could contain a floating-point array and an integer defining its size. Or a single message might contain an entire "structure," including integer arrays, character strings, and floating-point arrays. Why is this important? Packing a message is faster than transferring the data over a network, although this is beginning to change as networks become faster. By combining several different pieces of information into a single message, the user was able to decrease the number of sends in an algorithm, eliminating the startup latency for all the sends that are saved. Another important advantage was the avoidance of matching "structures" back up on the receiver. Let's illustrate this with a contrived example. Assume we restrict messages to a single data type and specify that the data structure to be sent is a sparse floating-point array without 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 that one task has to receive several structures of this kind from several other tasks. Because messages may come from different sources and because the order in which the floating-point and integer messages arrive at the receiver is arbitrary, several structures could be interleaved in the message queue. The receiver then was responsible for searching the queue and properly reassembling the structures. This search and reconstruct phase was not needed when the various data types are combined into the same message. The philosophy of PVM emphasized simplicity, a feature clearly in evidence here in that it is easy for a nonexpert 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 that the message has to be encoded and fragmented only once. In PVM once the message is packed, it can be sent to several different destinations. There are many parallel scientific applications in which a task must send its data to its "neighbors." In such cases PVM eliminates the overhead of packing for each send separately. A further advantage is that PVM packs only once when a user broadcasts a message.

The separate buffer initialization step also has the advantage that the user can append data to a buffer that has already been 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.

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 three-step send in this case. This is now possible in PVM 3.3 using the new function pvm_psend(), which combines the initialize, pack, and send steps into a single call oriented toward high performance.

The request for a pvm_psend() call and its complement, pvm_precv(), initially came from MPP vendors who were developing optimized PVM versions for their systems. On MPP systems vendors try to supply routines with the smallest possible latency. The overhead of three subroutine calls is high relative to 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. As an example, Table 1 shows that the message-passing performance on the Intel Paragon using pvm_psend/pvm_precv is only 5–8% higher 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 T3D is only 18 microseconds while the bandwidth is over 45 Mbytes/sec.

PVM provides several methods of receiving messages at a task. There is no function-matching requirement in PVM; therefore, 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
- pvm_precv()—combined unpack and blocking receive

PVM 3.3 supplies a timeout version of receive, pvm_trecv. Consider the case in which a message is never going to arrive (due to error or failure). Here 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. pvm_trecv allows the user to specify a timeout period. If the timeout period is set

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

Paragon csend/crecv		
Msg size (bytes)	**Round trip (μsec)**	**bandwidth (MB/s)**
8	376	0.0213
80	315	0.2540
800	314	2.5478
8,000	670	11.9403
80,000	3,978	20.1106
Paragon pvm_psend/pvm_precv		
Msg size (bytes)	**Round trip (μsec)**	**bandwidth (MB/s)**
8	322	0.0248
80	317	0.2524
800	314	2.5478
8,000	676	11.8343
80,000	4,061	19.6996

very large, pvm_trecv acts like pvm_recv. If the timeout period is set to zero, pvm_trecv acts like pvm_nrecv. Thus, pvm_trecv fills the gap between the blocking and nonblocking receive functions.

1.1 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 a MPP and across processors of a shared-memory multiprocessor are automatically optimized using native communication calls and shared memory, respectively. The following discussion is restricted to performance improvements across a network of hosts.

PVM uses UDP and TCP (Comer, 1991) sockets to move data over networks. UDP is a connectionless datagram protocol in which packet delivery is not guaranteed, while TCP requires a connection between processes and implements sophisticated 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 and finally the remote daemon transfers the message to the remote task when requested by a pvm_recv(). Since UDP does not guarantee packet delivery, PVM implements 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. This modification improves 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. Calling pvm_setopt(PvmRoute, PvmRouteDirect) enables PVM to 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 high but all subsequent messages between the same two tasks is 2–3 times faster than the default route method. The primary drawback of this method is that each TCP socket consumes one file descriptor. Thus, there is potential need for $O(n^2)$ 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.

Two encoding options available in PVM 3.3 are intended for boosting communication performance. Since a message may be sent to several destinations, by default PVM encodes messages for heterogeneous delivery during packing. If the message will only be sent to hosts with a compatible data format, the user can tell PVM to skip the encoding step 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 "in-place" 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 and also has the benefit of reducing memory requirements since the send buffer no longer holds a copy of the message.

On the other hand, care must be exercised when using PvmDataInPlace. If the user's data is modified after the pack call but before the send call, 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 in which 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 for sending 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 the direct and default routing, respectively. (The term "hop" is used because the default messages make extra hops through the PVM daemons.) Inplace packing is indicated by inp. Lines marked raw show the case of no data conversion, while xdr indicates conversion of messages into the XDR format before being sent and from the XDR format after being received at the destination. The tests were run on DEC Alpha workstations connected by FDDI. The experi-

ment showed that the avoidance of data copying and conversion along with direct routing enabled PVM to achieve good end-to-end performance for large messages. The peak bandwidth of FDDI is 100Mbit/sec or 12.5 MByte/sec. In the best case, we achieved approximately 8 MByte/sec bandwidth for large messages, which is 64% of the network's peak bandwidth. Note that these times include the time needed to pack the message at the sender and to unpack the message buffer at the receiver. The advantage of inplace packing for large messages is clearly shown. The high cost of heterogeneous data conversion can also be seen from the XDR bandwidth curves.

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

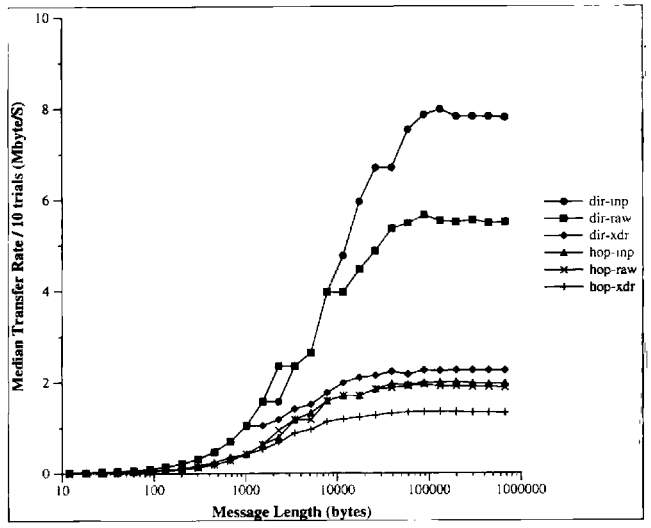

Fig. 1 PVM message bandwidth versus size.

2 Collective Communication

PVM 3 always had a very flexible and powerful model for grouping tasks, but until PVM 3.3 there were only two collective communication routines: either broadcast to a group of tasks or barrier across a group of tasks. PVM 3.3 adds several new collective communication routines, including global sum, global maximum, and scatter/gather. These new routines are described here.

The semantics of the PVM collective communication routines were developed using the MPI draft as a guide, but also adhering to the PVM philosophy to keep the user interface simple and easy to understand. By adding more collective routines, PVM saves users unnecessary effort, and allows 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 maximum. It 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

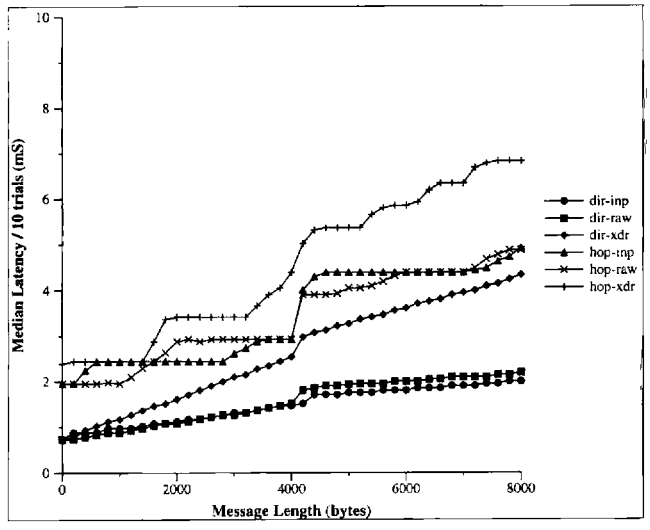

Fig. 2 PVM message latency versus size.

```
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
```

Fig. 3 Fortran reduce example.

- 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 the function is PvmMax, the result contains two numbers—the global maximum of each group member's first number and the global maximum of each member's second number. The Fortran code fragment to do this is shown in Figure 3. If all the group members need to know the result, the root task can broadcast the information.

Optionally, users can define their own function for use by pvm_reduce(). The PVM source distribution includes an example employing a user-defined function. The first argument in pvm_reduce() is a pointer to a function. Users can simply substitute their own function. Unlike the case of MPI, no additional PVM functions are required to define the user function.

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

pvm_gather() gathers data into one task. As in pvm_reduce(), all members of the group must call pvm_gather() with consistent arguments. In particular, a root must be specified. Following the gather, the root task receives the data from all group members, including itself, concatenated into a single vector. The data is concatenated in rank order (defined by the group being used), as in MPI.

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

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

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

pvm_scatter() is the inverse of the gather operation. The root starts out with a large vector containing equal size pieces destined for individual group members. Fol-

lowing the scatter, all group members have their own piece of the vector. 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)
```

Following this call, every task, including the root, has one integer in data. This integer is the same task ID placed into pvm_gather() by the task.

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

3 Extensible Services

The PVM system is used not only by programmers who wish to construct parallel programs, but also by systems builders who are interested in issues related to the design of distributed systems. To aid systems builders, we have discussed some of the functionalities of PVM to show systems programmers how to easily extend the base functionality. Note that the normal PVM programming interface for applications programmers is unaffected by these extensions.

New interfaces in version 3.3 allow PVM tasks to assume functions normally performed by the daemons, such as 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 who want to integrate their additions to PVM and distribute the resulting code. Any PVM task can *register* dynamically with the system, allowing the system to assume the specified function. This registration can occur while the virtual machine and applications are running, which aids in debugging the additions; in addition, all communication is done via normal PVM messages.

In PVM 3.3 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.

3.1 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 to get the maximum efficiency. Using knowledge not available to the pvmds, such as host load averages, an RM can make more informed decisions automatically. For example, when spawning a task, the RM could pick the host 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 (which is where the master pvmd runs) manages any slave pvmds that do not have their own RMs. A task connecting anonymously (not via a pvm_spawn call) 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 an RM assigned to it, service requests from the task to its pvmd are routed to the RM instead. The messages intercepted by the RM and their corresponding libpvm functions are shown in Table 2. Queries also go to the RM, since it knows more about the state of the virtual machine. The query messages are shown in Table 3.

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

Using the two messages SM_EXEC and SM_ADD, the RM can directly command the pvmds to start tasks or to reconfigure the virtual machine. On receiving acknowledgment 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.

Table 2
Messages from the libpvm Functions Intercepted by the Resource Manager

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

Table 3
Query Messages from the libpvm Functions to the Resource Manager

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

3.2 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 dialogue 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 a user to alter this mechanism for adding new hosts to the virtual machine. This might be useful for dealing with systems in which additional security is needed or in which 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 (i.e., upon receipt of a DM_ADD message), the master pvmd sends an SM_STHOST message to the hoster and waits for an SM_STHOSTACK message in reply to complete the operation.

Table 4
Format of Startup Messages

SM_STHOST			
	int nhosts		Number of hosts
	{		
		int tid	Of host
		string options	From hostfile so = field
		string login	In form [username@]hostname.domain
	} [nhosts]	string command	To run on remote host
SM_STHOSTACK			
	{		
		int tid	Of host
		string status	Line of output from slave or error code
	} []		Count is implied

Table 5
Format of the Start Task Message

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 bootstrap protocol between the master pvmd and the slave is designed to minimize what the hoster needs to know about the protocol. The hoster is sent a list of hosts and commands to run. It runs the commands and returns their output to the master pvmd, which does the parsing. The remainder of the startup is always done by the master pvmd. Table 4 shows these message formats.

The 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.

3.3 USE IN CONDOR

The resource manager and hoster interfaces were created in cooperation with members of the Condor project (Litzkow, Livny, and Mutka, 1988) and 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 responsible for all the tasks in the system. 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 the hosts that are 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, a necessary step since they are run under a Condor-owned login.

3.4 TASKER

A *tasker* is a PVM task that starts (*execs*, is the parent of) other tasks. The tasker facility allows a specific PVM task to control the creation and execution of all tasks in the system. This is 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, and is able to 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 of the processes it controls, a situation that is becoming less common with the growing availability of the attachable *ptrace* interface.

Prior to version 3.3, PVM provided a simple debugger interface. If a task is spawned (via the pvm_spawn call) with the flag PvmTaskDebug set, the pvmd now executes 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 must be 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 an SM_STTASK message to the tasker instead of calling execv(). No SM_STTASKACK message is required; as usual closure comes from the task reconnecting to the pvmd. The pvmd does not get SIGCHLD signals when a tasker is in use because it is not the parent process of tasks, so that the tasker must send notification of exited tasks to the pvmd in an SM_TASKX message. Table 5 presents the message format of the start task message.

The tasker interface is the result of collaboration with the Paradyn group (Hollingsworth, Miller, and Cargille, 1994). We hope that others will take advantage of it as well to ensure cleaner integration of their systems.

3.5 IMPLEMENTATION

We will briefly describe implementation details of these

new features. For a more in-depth description see Geist et al. (1994) and Manchek (1994).

We defined a new class of system messages (SM_XXX), to be exchanged among pvmds, resource managers, hosters, and taskers, as well as the 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 so that it could receive messages from arbitrary tasks, not just other PVM daemons and local user tasks. The pvmds do not usually communicate directly with tasks on other hosts. The pvmd has message-reassembly buffers for each foreign pvmd and for each task it manages. Reassembly buffers for foreign tasks would be too complicated. To free 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 at once to the destination pvmd, which reassembles the message. The source address is preserved so as to identify the sender. Libpvm maintains dynamic reassembly buffers, so that messages from pvmd to the 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 the 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 as it is not clear what action should be taken.

3.6 CAVEAT

The features presented in this section are geared to tool developers rather than the casual PVM user. 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 before. We have shown that by making the system dynamically extensible, it can be made more powerful and more general without increasing the amount of code. 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 as other researchers take advantage of them.

4 MPP and Shared Memory Support

PVM Version 3 is designed so that the message-passing calls of a specific system can be compiled into the source. This allows the fast, native message passing typical of 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 MPP subsystem of PVM consists of a daemon that manages the allocation and deallocation of nodes on the multiprocessor. This daemon is implemented using 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.

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 holding a part of the message (Figure 4); PVM's page size can be a multiple of the system page size. Each page has a header containing 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 5). To send a message, the task first packs the message body

into its buffer and 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 to allow the space to be reused. If a task is not able to deliver the header directly because of a full receiving box, 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 in all PVM implementations. If the buffer is full, 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 that multicasting is efficient (tasks do have to decrease the counter afterwards, resulting in some contention). Contention occurs only when two or more processes try 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.

5 New PVM Tracing Features

A new tracing feature in PVM directly supports the tracing of PVM programs. In this section we describe the tracing feature and present the XPVM tool for use in displaying PVM trace information. Tracing is a popular way of debugging parallel programs. There have been many systems that support program tracing for debugging (McDowell and Helmbold, 1989). More recently, systems such as Pablo (Reed et al., 1991), Paragraph (Heath and Etheridge, 1991), PICL (Geist et al., 1990), Bee (Bruegge, 1991), and Xab (Beguelin et al., 1993 and Beguelin, 1993) have been developed to aid the parallel programmer. Pablo, Paragraph, and Bee are intended as general tools for displaying trace information for parallel programs. PICL is a portable com-

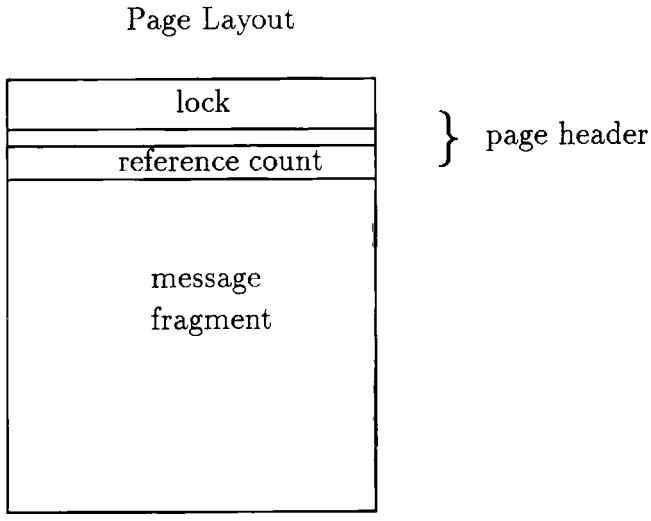

Fig. 4 Structure of a PVM page.

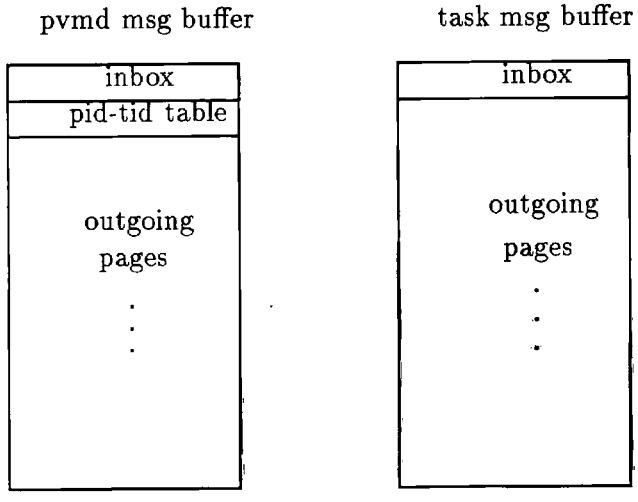

Fig. 5 Structures of shared message buffers.

munication library for multicomputers that 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 previous work done on Xab.

5.1 HOW TO USE TRACING

When tracing is turned on, events are generated for PVM calls that describe calls to the PVM library. For instance, a call to pvm_send() generates 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 the user 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 causes the calc program to be spawned with tracing turned on. Trace events are 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, 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 while the second command turns tracing on for the barrier routine.

Another way of using tracing is to activate it from within a PVM program. When spawning a task, tracing for the spawned tasks can be activated using the PvmTaskTrace flag in the pvm_spawn() call. This flag tells PVM that the tasks created by the spawn call should have tracing turned on. Each call to PVM in the newly spawned task generates trace events.

Each trace event 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 manual pages.

5.2 THE XPVM CONSOLE

Although PVM provides a level of flexibility that allows the programmer to control which events are generated and where event messages will be sent, most programmers do not need this much 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 6 shows an example of the XPVM console in action.

5.3 TRACING OVERHEADS

Since PVM's tracing facilities 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, other external factors such as varying machine load and network traffic can be expected to also affect the computation from run to run. To show how tracing may alter the runtime characteristics of a PVM program, we replicated the bandwidth experiment shown in Figure 1, this time, however, showing the bandwidth versus message size with tracing turned on for *all* PVM calls and without any tracing (see Figure 7). 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 an actual PVM application we would expect tracing to have a smaller effect.

In terms of bandwidth, tracing does not have an extremely detrimental effect, though Figure 8 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

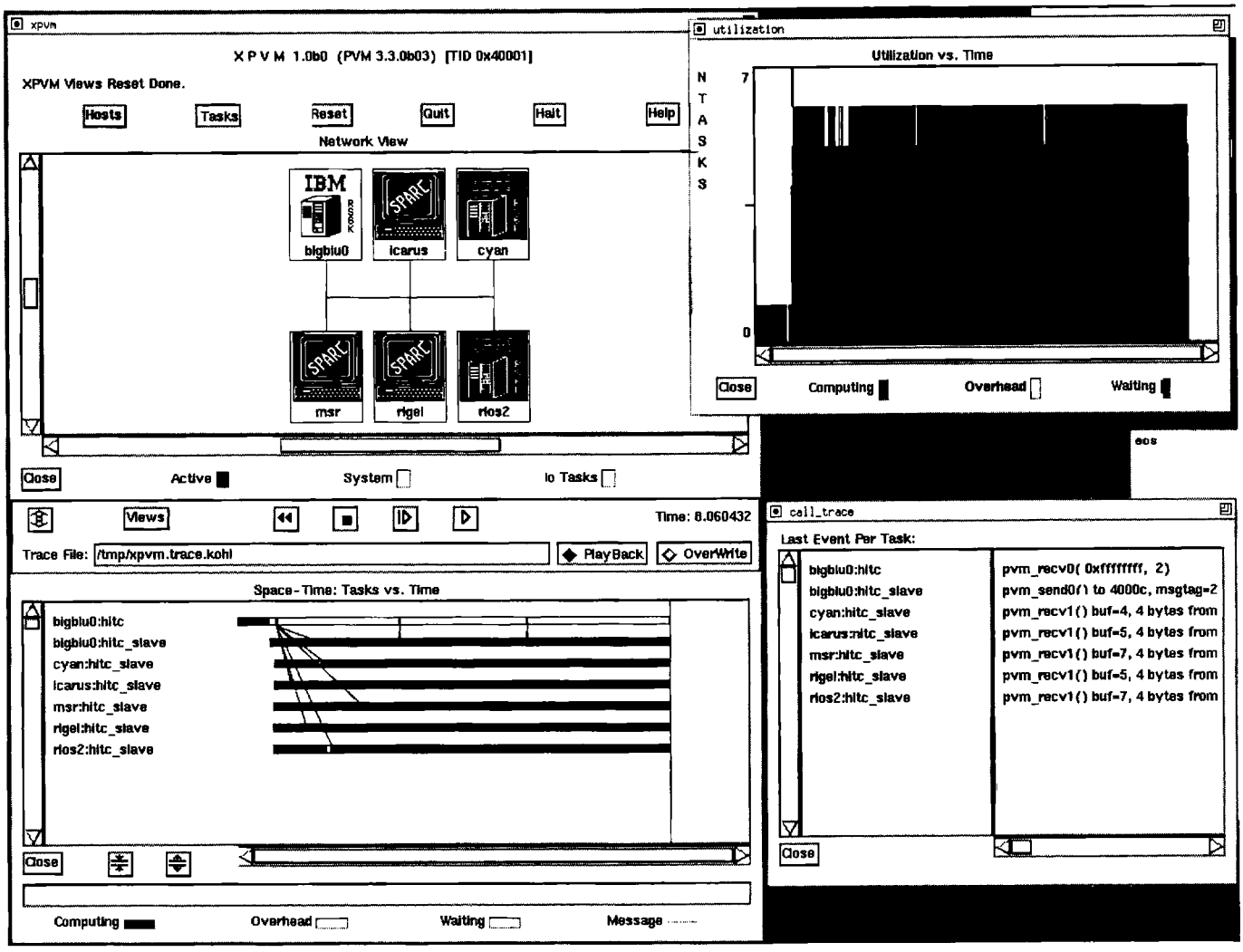

Fig. 6 The XPVM console.

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.

6 Future Research

The PVM project is an ongoing research effort aimed at advancing the state of the art of heterogeneous con-

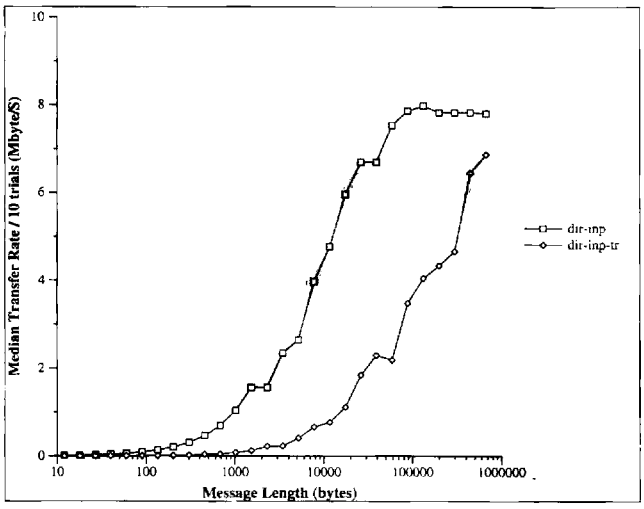

Fig. 7 Effects of tracing on PVM message bandwidth.

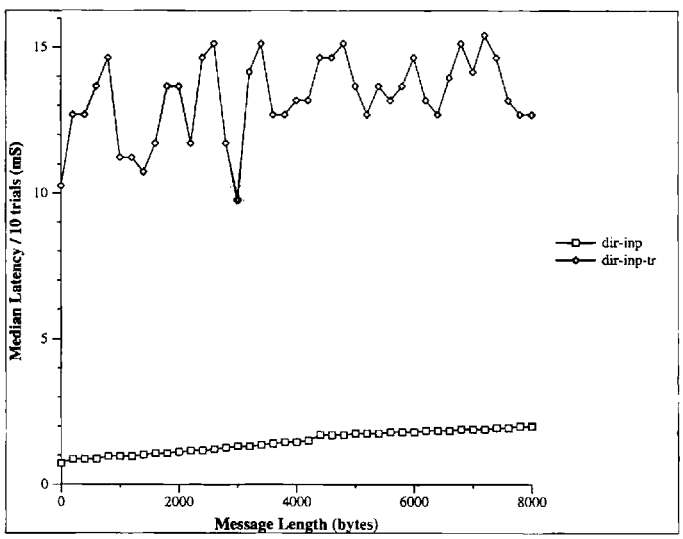

Fig. 8 Effects of tracing on PVM message latency.

current computing through 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 a concise and manageable computing model and programming interface. In this section, we briefly describe two research initiatives that have been recently undertaken as part of the PVM project.

6.1 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 the 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 multidatabases, 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 nonconflicting 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 I/O 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 in that the primitives that are provided 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 toward 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, 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 the sending and receiving of messages, it is unclear what are the correct actions in the case of an abort, restoration of the system to a previously valid state is a complex and possibly intractable procedure. We are exploring several alternatives and will proceed to incorporate these features into the GDC layer as soon as the "correct" semantics have been decided upon.

Our preliminary experiences with the GDC subsystem indicate that enhancing the features of PVM to support generalized distributed computing, with specific focus on commercial, business, and database applications, is a very valuable step 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 here, such as locking, synchronization, and parallel input and output with shadowing.

6.2 THREADS-BASED PARALLEL COMPUTING

Thus far, cluster software systems have used a process-based model of parallelism, as in distributed memory multiprocessors. At the opposite end of the spectrum is loop-level parallelism, a model that is common in vector supercomputers. To enhance functionality as well as performance, we are investigating a threads-based parallelism model within PVM that provides a compromise 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 over-

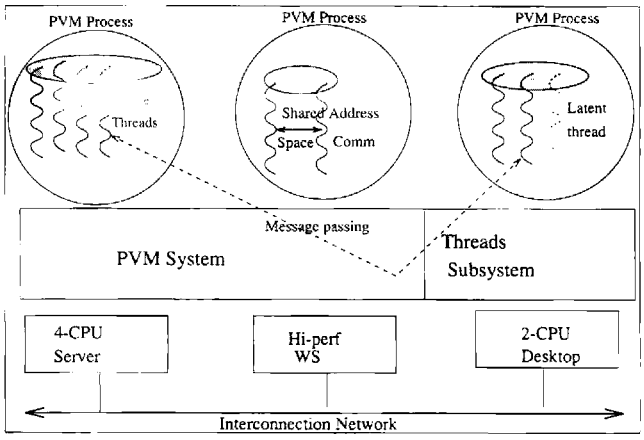

Fig. 9 PVM threads.

heads. Several stand-alone threads packages are available, and operating systems are incorporating native threads into their repertoire. It is anticipated that threads will be a standard feature of most software environments in the near future.

Figure 9 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 runtime 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, allowing for more control in load balancing. Once spawned, threads communicate via explicit message-passing calls. In reality, however, messages are exchanged only when the 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 typical of a process-based model. 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 a 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. With a threads-based model, however, one thread can be productively utilizing the CPU while 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. Sunderam (1994) discusses other aspects of the threads-based implementation of PVM.

7 Summary

PVM is the mainstay of the heterogeneous concurrent computing project which now 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 patch-level 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 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 of having approached 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, has enabled 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.

ACKNOWLEDGMENT

This research was sponsored in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330, issued by DARPA/CMO under contract MDA972-90-C-0035; the Applied Mathematical Sciences program, Office of Basic Energy Sciences, U.S. Department of Energy, under grant no. DE-FG05-91ER25105; the National Science Foundation, under Award Nos. CCR-9118787 and ASC-9214149; and CNRI.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government or of CNRI.

BIOGRAPHIES

Adam Beguelin joined the faculty of Carnegie Mellon University in the

spring of 1992. He holds an appointment with the School of Computer Science and the Pittsburgh Supercomputing Center. He received his Ph.D. in Computer Science from the University of Colorado in 1990. His primary research interests are in the area of computer systems, specifically the design and development of programming tools and environments for high performance parallel and distributed computing. He is currently working on software tools to aid in the programming and performance tuning of parallel and distributed computer systems. The Dome system, for one, provides distributed objects for networks of computers. Dome eases the task of multicomputer programming by supporting automatically distributed objects, dynamic load balancing, and architecture independent checkpoint/restart.

Jack Dongarra holds a joint appointment as Distinguished Professor of Computer Science in the Computer Science Department at the University of Tennessee (UT) and as Distinguished Scientist in the Mathematical Sciences Section at Oak Ridge National Laboratory (ORNL). He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. Other current research also involves the development, testing and documentation of high-quality mathematical software. He was involved in the design and implementation of the software packages EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, the BLACS, MPI, and PVM/HeNCE, and is currently involved in the design of algorithms and techniques for high performance computer architectures.

Al Geist is a computer scientist in the Mathematical Sciences Section of Oak Ridge National Laboratory. His research interests are in the areas of parallel and distributed processing, scientific computing, and high performance numerical software.

Robert Manchek is a Senior Research Associate at the University of Tennessee, Knoxville. His research interests include parallel computing, networking, and operating systems. He received a B.S. in Electrical and Computer Engineering from the University of Colorado, Boulder in 1988 and is currently pursuing a Ph.D. in Computer Science at the University of Tennessee.

Vaidy S. Sunderam received a Ph.D. in Computer Science from the University of Kent, U.K. in 1986, and is currently Associate Professor in the Department of Math and Computer Science at Emory University, Atlanta, USA. His research interests are in parallel and distributed processing, particularly high performance concurrent computing in heterogeneous networked environments. He is the principal architect of the PVM system, in use at several thousand institutions worldwide for heterogeneous concurrent computing, and was awarded the 1990 IBM supercomputing prize for his research contributions in this area. He is also co-principal investigator of the Eclipse research project, a second-generation system for high performance distributed supercomputing, that won the IEEE 1992 Gordon Bell Prize. His other research interests include distributed and parallel I/O and data management, communications protocols, parallel processing tools, and concurrent stochastic simulation. He is the recipient of several research grants, has authored numerous articles on parallel and distributed computing, and is a member of ACM and IEEE.

REFERENCES

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

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

Bruegge, B. 1991. A portable platform for distributed event environments. Proc. ACM/ONR Workshop on Parallel and Distributed Debugging. *ACM SIGPLAN Notices*, 26(12): 184–193.

Comer, D. 1991. *Internetworking with TCP/IP.* Prentice Hall, 2nd edition.

Geist, A., Beguelin, A., Dongarra, J. J., Jiang, W., Manchek, R., and Sunderam, V. S. 1993. PVM 3 User's Guide and Reference Manual. ORNL/TM-12187. Oak Ridge National Laboratory.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. 1994. *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press.

Geist, G. A., Heath, M. T., Peyton, B. W., and Worley, P. H. 1990. A machine-independent communication library. In *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications,* edited by J. Gustafson, Los Altos, CA: Golden Gate Enterprises, pp. 565–568.

Heath, M. and Etheridge, J. 1991. Visualizing the

performance of parallel programs. *IEEE Software* 8(5):29–39.

Hollingsworth, J., Miller, B., and Cargille, J. 1994. Dynamic program instrumentation for scalable performance tools. In *Proc. IEEE Scalable High Performance Computing Conference,* Knoxville, Tenn. pp. 841–891.

Litzkow, M., Livny, M., and Mutka, M. 1988. Condor—A hunter of idle workstations. In *Proc. Eighth Conference on Distributed Computing Systems.* San Jose, California.

Manchek, R. J. 1994. The design and implementation of PVM version 3. Master's thesis. CS-94-232. University of Tennessee, Department of Computer Science, Knoxville.

McDowell, C. E. and Helmbold, D. P. 1989. Debugging concurrent programs. *ACM Computing Surveys* 21(4):593–622.

Message Passing Interface Forum. 1994. MPI: A message-passing interface standard. *Internat. J. Supercomput. Appl.* 8(3/4).

Reed, D. A., Olson, R. D., Aydt, R. A., Madhyastha, T. M., Birkett, T., Jensen, D. W., Nazief, B. A. A., and Totty, B. K. 1991. Scalable performance environments for parallel systems. In *The Sixth Distributed Memory Computing Conference,* edited by Quentin Stout and Michael Wolfe. IEEE Computer Society Press, pp. 562–569.

Sunderam, V. S. 1994. Heterogeneous concurrent computing with exportable services. In *Environments and Tools for Parallel Scientific Computing,* edited by Jack J. Dongarra and Bernard Tourancheau. SIAM Press, pp. 142–151.