New Features of PVM 3.4 and Beyond

G.A. Geist J.A. Kohl R.J. Manchek
P.M. Papadopoulos *

August 25, 1995

Abstract

This paper reviews the progress and design ideas for the new features for PVM 3.4 and describes results of our work on the longer range features that may one day be a part of the PVM system.

1 Introduction

PVM (Parallel Virtual Machine) [1] is a byproduct of an ongoing Heterogeneous Distributed Computing research project at Oak Ridge National Laboratory and the University of Tennessee. About a dozen different distributed computing research ideas are being investigated at this time in our project. For PVM 3.4 the new research ideas include context, heterogeneous data encodings, new trace formats, static groups, MPI interfaces, and thread-safety. The PVM team has also done research in low level ATM network interfaces, network multicast, parallel I/O, task migration, and lightweight threads.

This paper reviews the progress and design ideas for the new features for PVM 3.4 and describes results of our work on the longer range features that may one day be a part of the PVM system.

2 Context

As the use of parallel programming grew and applications began calling third party parallel libraries, it became apparent that message tags and sender-id were necessary but not sufficient to guarantee that the correct message was received. For example, consider the case were task A in a parallel application calls a math library routine and this routine contains message-passing. If the math routine has been written by a third party, then the application developer does not know which message tags (or wildcards) are used inside it. There

*This work was supported in part by the U.S. Department of Energy, under Contract DE-AC05-84OR21400, and in part by the the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615.

is no way he can guarantee that a message sent to task A by task B in the application will not be inadvertently intercepted by the math routine. Thus, causing both the math library and the application to fail.

Introducing a communication context is a solution to the above problem. The parallel-library writer wants to guarantee that no message-passing activities within the larger application can interfere with the library routines. He wants all the library communication, both sends and receives, to be completely isolated from all other communication within the larger application. This can be done by proper design of a library using the context concept, which leads to correct partitioning of the message space between the parent application and the called library routine

The PVM team has chosen a different context design than the MPI forum. Both designs give the additional safety required by parallel libraries, but they differ in how hard they are to use. There are tradeoffs between the PVM design and the MPI design. In the following discussion we will present the PVM design and contrast it with the MPI design. Overall, the PVM philosophy has always been to keep things as simple as possible for the user; the MPI philosophy is performance at all costs.

PVM 3.4 will have communication context, but its use will not be required. All existing PVM applications will continue to work unchanged, and all applications that want to use context can incorporate it with a minimal number of calls and source changes. MPI requires the user to always use context in every MPI call.

In our design, context is merely an additional system tag that goes along with the users' ordered pair of (message tag, destination). In MPI, the context is folded, along with group information, into a *communicator* to which the user has a handle.

The PVM interface to context provides a mechanism for a task to allocate a system-wide unique identifier for use as a context tag. In MPI, context tags are not unique. Two MPI groups may (and often) use the same context. This requires MPI to invent a second form of context called an inter-communicator in order to send messages between two groups. Having an inter-communicator allows the context value to be completely hidden from the user which makes the user interface safer at the cost of increased complexity. With PVM's system-wide unique context multiple groups can send to each other as long as they know each others' context. Having a system-wide unique context makes using contexts more intuitive and simplifies the user interface.

The PVM design allows the context interface to be encapsulated into four routines:

```
newcntx = pvm_newcontext();        /* get a new context to use */
oldcntx = pvm_setcontext(newcntx); /* set my context and return present value */
info    = pvm_freecontext(contx);  /* free context */
contx   = pvm_getcontext();        /* get present context */
```

A spawned task inherits the current operational context of its parent. If the task has no parent (i.e., the task was started from the command line), then it operates in the base context, which is the same as no context. Inheritance provides an easy way for a master to encapsulate the messages of several worker groups.

3 Receiver Makes Right

PVM 3.4 is scheduled to use a *Receiver makes Right* (RMR) data encoding rather than the XDR and RAW encodings now available. Previously, PVM used XDR encoding to send messages in heterogeneous environments. XDR has the advantage of being very portable because the vendors supply the routines as part of the RPC library. XDR has the disadvantage of being rather slow both in its implementation and its execution. First, the sending task must convert the entire message into XDR format. After it is transmitted, the receiver must decode the message from XDR into this host's local format. Experiments have been done using a much faster encoding scheme called *Receiver makes Right*. In this scheme the sender always sends the message out as fast as it can, doing no encoding. The receiver has the responsibility of converting the message into the receiving host's format. The advantages are: The routines can be streamlined and thus much faster than XDR; the scheme requires the fewest possible conversions in a heterogeneous environment – at most one vs. always two for XDR; the RMR scheme fits nicely into other PVM extensions such as InPlace data packing; and the RMR scheme avoids the problem XDR has with 64 bit architectures, i.e. XDR is only defined for 32 bit words. The major disadvantage of RMR is that each receiver must be able to decode every other format, thus many decoding routines could be required. Given the number of machines that PVM supports this number could be as high as 580 routines, but by carefully reusing routines such as byte_swap, the actual number of routines that must be written and maintained is approximately 30. Another disadvantage is that RMR is not as portable as XDR because the vendors do not supply the decoders for new architectures they may build. One observation is that almost all new architectures use IEEE data format so this problem may be quite rare.

To increase the portability of the RMR scheme, the data format integer used in PVM 3.3 message headers is replaced by a 32 bit encoding string in PVM 3.4. Each bit represents some feature of the sender's data format, such as, byte order, size of int, etc. The receiver compares the message's encoding string to its own string. The different bits identify which decoders must be applied to this message. One advantage of this design is its efficiency and speed. Another advantage of this design is that (most likely) no new decoders are required if a new machine is ported to PVM. Because the decoders are not dependent on a particular format but rather a property of a particular format, once the new format encoding string is determined the port should work.

With the RMR scheme the issue of switching between RAW and DEFAULT modes goes away. In PVM 3.4 the user automatically has the best of both. Between homogeneous hosts the speed of RMR should effectively be the speed of RAW. And if the hosts are heterogeneous the speed should be faster than the present XDR DEFAULT.

4 Buffered Tracing

One of the unsolved challenges to developing good parallel programs is successful debugging and performance tuning. Two of the main difficulties in this area are developing tools that are intuitive and easy to use and to have these tools determine the precise behavior of a set of concurrently executing tasks without significantly perturbing them.

To address these points, PVM 3.4 will contain a new and improved tracing facility which provides more flexible and efficient access to run-time program information. This new tracing system will support a buffering mechanism to reduce intrusion, and a more flexible trace event definition scheme based on SDDF (Self-Defining Data Format) [3]. This new scheme expedites the collection and analysis of execution histories, and allows for integration of custom trace events. The tracing instrumentation is built into the PVM library to avoid re-compilation, and supports on-the-fly adjustments to each task's trace event mask, to control the level of tracing detail.

Along with this new tracing facility, the graphical console and monitor XPVM has been updated to provide better access to the new tracing functionality. Several new views have been implemented in XPVM 1.1 to utilize the additional tracing information, including user-defined events. One new view is a dynamic view of the message queue size for each PVM task. This view allows the user to see if messages are piling up (and consuming buffer space) on any of the tasks. The number and size of each message is displayed, and submenus allow the user to determine additional information about a particular message. The host view has been modified so that it now displays the volume and number of messages flowwing over the network between hosts. This view allows the user to more easily detect network "hot spots" and configure or code around them. The XPVM has also been optimized to provide better real-time monitoring capabilities.

5 Name Service

PVM is completely dynamic. Hosts may be added to and deleted from the virtual machine. Processes my start asynchronously, join PVM and communicate with other PVM tasks. It is often desirable for two programs to start independently and discover information about each other. One mechanism would be for each of the programs to know about a "well-known name" to use as a key in an information database. A program could look up the name and have essential information such as a task identifier (tid) and context to use when communicating with the named task. The "well-known name" is user-defined and therefore invites name clashes. This is a necessary evil if tasks are to start independently and find out essential information. For example, Unix mailers use a well-known socket number (25 for most systems) on which to provide their service. On the server side, an error will occur if a task tries to bind a name to socket that is already in use. The name server provided in PVM 3.4 will associate a user-defined character string (key) with a message. Only one PVM task may insert a specific string in the database. When any task performs a lookup on the name, the message will be sent to the calling task. This sort of name service is a very general mechanism. For example, to define a group, the message could contain a group id, a list of task id's and an associated context.

There are three basic calls for the name server The call pvm_putinfo() inserts a (name, message) pair into the name server. The message is explicitly packed into a message buffer before pvm_putinfo() is called. The owner of the key is the tid specified in an argument to pvm_putinfo(). Two tasks may not insert the same name since only the first insertion will succeed. This has the advantage that servers may start up and insure that they are unique across the entire virtual machine.

The routine pvm_getinfo() looks up a key, and copies the registered message into a receive buffer labeled by bufid. It is up to the receiver to unpack and interpret this message.

The routine pvm_delinfo() deletes the key and message from the name server database. When the owning tid leaves the virtual machine, the name and associated data is automatically removed from the database by a pvmd.

6 Static Groups

In PVM there is the concept of a user defined "group" of tasks that the user may get to work together on one part of an application while another group works on another part. Groups also need to be defined before collective communication can be performed. Collective communication is message-passing that involves more than a single sender and receiver. An example of a collective operation is broadcasting data to a group of tasks. Presently, all groups are dynamic in PVM, that is, group members can join and leave a group asynchronously. Any task can also be a member of multiple groups. Dynamic groups allow great flexibility in developing applications that add and subtract tasks as load varies, and are fault tolerant to host or task failure. This flexibility comes at the cost of performance. For example, to avoid race conditions PVM implements dynamic groups using a central group server. On massively parallel systems (MPP) the server can become a serial bottleneck that degrades performance.

In static groups the members of the group can never change once formed. We plan to add routines to PVM 3.4 that allow for statically grouping tasks. The current dynamic group server will still exist in PVM 3.4, and dynamic groups will still be available to the programs that need their benefits.

Static groups are created synchronously and take a list of tids as an argument. A context is automatically assigned to the static group at formation time, and encoded into the group ID so the user never needs to see it. Groups may overlap. Static group formation and destruction are both synchronizing routines.

Static group communication routines will be introduced that automatically give applications the safety of context and the performance benefits of static groups. We think the combination of the safety of context and the performance of static groups will make these new group operations very popular.

7 MPI/PVM and PVM/MPI

Over the past three years the Message Passing Interface (MPI) Forum [2] has defined a standard specification for message-passing. Over that same time PVM was becoming the de facto standard for distributed computing worldwide. The packages are designed for different environments, but each is flexible enough to overlap into the other's space. MPI is designed to run best in a multiprocessor environment where the vendor supplies an optimized implementation. PVM is designed to run across a network of heterogeneous hosts.

Questions have arisen from users about if these standards will merge. One observation is that each interface is picking up features from the other. For example, PVM is picking up context, MPI is picking up PACK and also SPAWN. So features are merging but the API's will likely remain distinct. Why? One reason is the PVM development team feels an obligation to the tens of thousands of PVM users to not break their codes by changing the PVM API. A second resaon is the PVM API has evolved over several years based on the needs and requests of distributed computing users, it is well designed for computing on a network of heterogeneous hosts. A third reason is the PVM philosophy is to keep the interface simple and easy to learn and use.

In comparison, the MPI interface is designed so parallel programming experts can squeeze every ounce of performance from a vendor's MPP. The MPI-1 API is a good (although overly large) design for MPPs, but lacks features such as process control and interoperability that are important for writing portable distributed programs.

A second observation is that each package can and has been built on top of the other. In PVM 3.3.8 there is an MPI port to the IBM SP-2 that works like the NX port to the Intel Paragon. A pvm_psend on the SP-2 is translated automatically by PVM into an optimized MPI call supplied by IBM. This way PVM can exploit the speed advantage of MPI on multiprocessors while still retaining the ability to send messages between multiprocessors of different vendors.

One of the omissions in the MPI-1 standard was any form of process control. This was not critical on multiprocessors which had their own, but makes MPI-1 unusable on a network of workstations where some standard form of process control is required. There are a couple of MPI implementations that have or are thinking about using PVM for process control on a network. The goal of this effort would be the ability to start a PVM console, create a virtual machine, and then run MPI applications from the PVM console. Underneath, the PVM daemon structure would perform process control functions while all the task communication would go through the MPI structure.

8 And Beyond

Standard PVM works with ATM networks by using TCP/IP. Experiments have shown that communication between two SPARC2 computers is up to five times faster on ATM than on ethernet. Other experiments have shown that an additional 10% bandwidth improvement in possible by using an API below TCP. (FORE Systems supplies such an interface with their ATM cards.) On the minus side the latency is actually higher using the lower level because this level only has unreliable delivery and the user must write his own handshake protocol. Other groups have confirmed our results that going to FORE's low level ATM interface makes no measurable improvement in their applications.

Network multicast is being incorporated into most new operating systems. Multicast sends one message onto a network that is received by a predetermined set of hosts. This is potentially much faster than the present method in PVM, which sends separately to each host. An experimental version of PVM which uses reliable multicast has been tested. Results are encouraging but not dramatic. The problems of defining what host groups should listen and which should not has pushed this research idea out into the future.

We have created an experimental, lightweight process (thread) based PVM version designed to take advantage of the features of Symmetric Multiprocessors (SMPs) as long as threads and global shared memory (GSM) are supported. It was created to study potential performance improvements and implementation issues required by multithreaded message-passing systems. The experimental package is called LPVM and due to its thread-safe design it is not compatible with standard PVM. Our tests have shown that some operations such as task creation can be a 1000 times faster using LPVM but communication in the present implementation is about 50% slower in LPVM.

Steve Moyer and Vaidy Sunderam have created a parallel I/O package called PIOUS on top of PVM. The design is both portable and extensible. It can even be used for wide-area transaction processing. The PIOUS software is available from `ftp.mathcs.emory.edu` Other groups around the world have also worked on parallel I/O the works with PVM programs. One example is the PFSLib project sponsored by Intel in Germany.

We are beginning to explore security issues related to PVM. A project is being funded to add to PVM the ability to send data with different levels of security. The intent is to be able to decide on a per message basis whether the message can flow over the network without any security (present situation), whether the data only needs to be authenticated, (encrypted checksums guarrantee the message hasn't been modified), or whether the data is sensitive and can't be viewed (entire message is encrypted using a session long key). The later case becomes important as more business, financial, and medical programs get written in PVM.

9 Conclusions

As computer and network technology change, new distributed computing research areas open up. Similarly, as users get more comfortable with distributed computing, the more capabilities, such as scalability and fault tolerance, are expected from the distributed computing packages. This paper shows the research nature of the PVM project and the many areas that the PVM team is exploring to push the envelope of distributed computing technology. Many of the features described here will move from experimental versions to the standard release version of PVM, while others will prove to be unportable or of not enough benefit to warrant inclusion.

Further information about the PVM project can be found at the Web site: http://www.epm.ornl.gov/pvm.

References

[1] G. A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. PVM: A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, 1994.

[2] MPI Forum, "MPI: A Message-Passing Interface Standard", *International Journal of Supercomputer Application Vol. 8 No. 3/4*, 1994.

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