Adding Context and Static Groups into PVM

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

July 6, 1995

Abstract

This paper describes how the concept of a communication context can be added to the PVM system. Context is a way to partition the communication space so that a message sent within one context cannot be received in a different context. This partitioning is essential for the design of robust parallel software libraries. Context can be added to PVM in such a way that existing PVM applications need not be changed.

A potential bottleneck of communication, especially for massively parallel computers, is PVM's central group server. This paper also describes a method for adding *static groups* to PVM. Allowing a user to specify a group with a static membership list eliminates the central bottleneck for many collective operations. The static group functions proposed here incorporate context to provide safe, collective operations.

1 Introduction

PVM (Parallel Virtual Machine) [2] is a byproduct of an ongoing Heterogeneous Distributed Computing research project at Oak Ridge National Laboratory and the University of Tennessee. The PVM software, which allows a collection of UNIX computers connected to a network to be used as a single, large, parallel distributed memory computer, continues to evolve and improve as new research ideas are tested and incorporated.

About a dozen different distributed computing research ideas are being investigated at this time in our project. An overview of most of these ideas and research results can be found in [1]. They include ideas such as low level ATM network interfaces in PVM, parallel I/O, task migration, buffered tracing, network multicast, faster heterogeneous data encodings, and lightweight threads. In this paper we will describe the ideas of communication context and static groups. Schemes for incorporating communication context and static groups into the PVM environment have been explored over the past several months and a proposed implementation is presented here.

Over the past two years the Message Passing Interface (MPI) Forum [4] has defined a standard specification for message-passing. While much of the specification can be described as "standard message-passing practice" a few ideas that where not found in standard practice where incorporated. Perhaps the most notable are context and derived datatypes.

*This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, 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.

PVM 3.3 already contains the capability of derived datatypes, but does not have the concept of a communication context.

In PVM and other message-passing interfaces, each message is labeled with a user supplied tag before sending. The receive function matches on this tag (and on the source of the message). This allows the user to discriminate between multiple messages arriving at the same time. In PVM the tag is a single positive integer. Message passing systems also allow the receiver to specify "wildcard" values for matching. If the source is a wildcard then matching is only done on the tag. If the tag is a wildcard then matching is only done on the source. And if both are wildcards then any incoming message matches. Such "promiscuous" posting of receives is a common programming technique.

As parallel programming grew and applications became more sophisticated it became apparent that message tags and source 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 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.

One way to contrast message tags and context is that message tags differentiate messages in time while context differentiates messages in space. With context, tasks inside a group can send messages to each other, but these processes can not receive messages from outside tasks. Thus, the *space* of all tasks is partitioned by different contexts. Message tags are user chosen and are associated only with the message and not the task that sends it. By using unique message tags in a "recv", a user can guarantee that messages are received by a task in a particular order regardless of the order that they were sent. Thus, message tags help control the *time* order that messages are received. While related, context and message tags are really different concepts.

In PVM there is the concept of a user defined "group" of tasks that the user may wish 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 defined. 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 highly parallel systems the server can become a serial bottleneck that degrades performance.

An example of the bottleneck that the central dynamic group server can incur is in the broadcast mechanism. Before a task actually broadcasts a message to the group of destination tasks, it must first get the the list of recipients from the group server. This roundtrip message is unnecessary when groups are known to be static and can severely degrade the overall performance of a program that relies on collective operations. The problem is especially acute on homogeneous MPPs where the server resides on the front-end. With static groups, the roundtrip message is eliminated and a significant performance boost can be realized.

In static groups the members of the group can never change once formed. This paper proposes adding the capability to form a static group in PVM and give it a unique context. We think the combination of the safety of context and the performance of static groups will make these new group operations very popular.

In the next section we propose four new routines for PVM that add the concept of static groups and context to the PVM system. Section 3 describes how context can be implemented inside PVM. Section 4 describes a generalized name server to be integrated into the PVM daemon that among other things will allow other PVM tasks to find out about the static groups and their members.

2 Context

Context provides a mechanism that allows tasks to choose arbitrary message tags and still have the guarantee that messages with these tags will not be received inadvertently by a task executing in a different thread or routine. One way to visualize context is to consider two tags associated with a message – a user tag and a context tag. The user tag is arbitrarily assigned by the programmer, while the context tag is assigned by the system. If, for example, a routine does a wildcard receive on the user tag, then only incoming messages with the same context tag will get matched. This has the effect of dividing messages into separate communication spaces. The concept of context is critical to library writers because they must guarantee that their messages will not be intercepted, especially in an environment where they cannot know *a priori* a "safe collection" of message tags to use. Context gives library writers the encapsulation that they need.

We take the view that context is merely a system tag that goes along with the users ordered pair of (message tag, destination). The proposed interface provides a mechanism for a task to allocate a system-wide unique identifier for use as a context tag. Two routines make up the allocation mechanism:

```
int context = pvm_newcontext()
       return a unique context tag to the calling task

int info = pvm_freecontext(int context)
       free a context for future re-allocation
```

The routine pvm_newcontext() returns a unique context tag to the caller. It is left to the caller to distribute the context to other tasks for communication to be made possible in the new context. One natural idea is to associate a context with a static group of tasks. A later section describes new static grouping calls that automatically incorporate context for safe collective communication.

The send and receive calls in PVM do not have an explicit argument for a context tag. Instead, we associate with each tasks a *current operational context* that provides the context tag for all communications. Once set, the current operational context stays in effect until it (re)set to a different value. Two routines are defined for manipulating the operational context.

```
int context = pvm_getcontext()
       return current operational context

int oldcontext = pvm_setcontext(int context)
       set the current operational context and return
       the previous operational context
```

The first merely retrieves the current operational context of the caller. The second routine sets the operational context and returns the previous operational context. There is a base context that all tasks know about and is the context that tasks operate in if an operational context has never been set. Spawned tasks start up in the current operational context of their parent or in the base context if they have no (PVM) parent.

For context to be practical, an implicit or explicit collection of tasks must know about and use a particular context tag. MPI, for example, requires the group of tasks to be explicit. Implicit groupings have desirable properties in the dynamic environment of PVM. For example, a task that collects trace output data does not need to explicitly know all that tasks that will be sending it data. The sending tasks need only know the proper context with which to mark trace messages for the receiver. Tasks may join and leave the virtual machine and continue to relay trace data without interfering with other (unrelated) tasks on the virtual machine. For example, a climate simulation and molecular dynamics simulation may run on the same virtual machine and send trace data to the same process. By allowing an implicit grouping, the two simulations are never required to synchronize with each other or with the trace output collection task. In this implementation, it doesn't make sense to "send a message to all tasks in context X." Any number of tasks may have their current operational context set to X.

2.1 Inheritance

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

```
master:
            context_1 = pvm_getcontext();
```

```
            pvm_spawn("workers_1");
            context_2 = pvm_newcontext();  /* context allocated */
            pvm_setcontext(context_2);
            pvm_spawn("workers_2");
            pvm_recv(-1,-1);  /* receive any message from workers_2 */
            pvm_setcontext(context_1);
            pvm_recv(-1,-1)   /* receive any message from workers_1 */

workers_1:
            mycontext = pvm_getcontext();  /* context_1 from parent */
            pvm_send(pvm_parent(),1);

workers_2:
            mycontext = pvm_getcontext();  /*context_2 from parent  */
            pvm_send(pvm_parent(), 1);
```

2.2 Ownership of Context

Context is allocated by the local pvmd. The number of available context tags is a limited resource and different management schemes for this resource require tradeoffs. In this design, context is not "owned" by any particular task. In other words, if the task that allocated the context dies, the context is not automatically recycled. Contexts are recycled if a task explicitly calls pvm_freecontext or if the entire virtual machine is reset. The danger here is that the context resource may simply run out. A clear advantage comes about in fault tolerance: a group of tasks may replace a failed task and continue to use the extant context tag, even if the failed task was the one to allocate the context. The remaining tasks may use the context to collectively decide how and where the replacement task should be spawned. If the context automatically goes away, then the remaining tasks have no safe communication space for making this decision.

The non-ownership of context represents a clear tradeoff. On the one hand, context tags may simply run out if not automatically recycled. On the other, remaining tasks still using a context would be put into an indeterminate and unsafe state, if the context were automatically recycled on a particular task exit.

3 Name Service

PVM is completely dynamic. Hosts may be added to and deleted from the virtual machine. Processes my start, run to completion and then exit. 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. This means that two mailing daemons may not run simultaneously on the same host. The name server provided here associates a user-defined string with a message. Like Unix's insistence that only one mail daemon bind to socket 25, only one PVM task may insert a specific string in the database. When another 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. The message could, for example, contain a list of task id's and an associated context, or just a task id, or simply initialization data. There are three basic calls for the name server

```
int info = pvm_put ( char *key, int bufid, int owner_tid)
     insert the packed message in bufid into the central
     name database and reference it by the user-defined
     key.

     The key is automatically deleted if the owner_tid exits
     the machine.

     An error is returned if key already exists

int bufid = pvm_get ( char *key )

     return a message buffer id that contains the message
     associated with the key.

int info = pvm_del (char *key)
     delete the key from the database. Caller must either
     be the owning tid or a pvmd.
```

The call pvm_put() inserts a (name, message) pair into the name server. The message is explicitly packed into a message buffer before pvm_put() is called. The owner of the key is the tid specified in the owner_tid argument. If the owner tid exits the virtual machine, then the key and associated data is removed from the database. Two task 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_get() looks up a key, and copies the registered message into the buffer labeled by bufid. It is up to the receiver to unpack and interpret this message.

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

4 Grouping Mechanisms

The previous two sections provide the basic mechanisms to build safe collective communication mechanisms. Previous to this release, PVM has provided only dynamic groups with membership kept in a single central server. The group server is a probable communications bottleneck when group operations are relied upon heavily. Performance suffers and the end result is that dynamic groups are too slow to use in most instances. The advantage of dynamic groups is that it is possible to build some form of fault-tolerance. Nevertheless, it is clear that most of today's programs use groups that are static. The user pays all of the performance penalty for dynamic groups but gains none of the benefits. The routines presented here provide a mechanism for statically grouping tasks. The current group server is set aside into its own library so that dynamic groups are still 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, but the operational context is not changed. Groups may overlap. Group formation and destruction are both synchronizing routines.

```
int gid = pvm_staticgroup(int * tidlist, int ntids, int *characteristic)
          form a group from the tidlist.
          The characteristic is reserved for future use and
          could, for example contain information about the
          underlying topology

          Synchronizing call where all tids in tidlist
          make the subroutine call with identical tidlist's.

int info = pvm_freegroup(int gid)
          synchronous call to delete the group.
          The contexts used for the group are
          freed so that they may be recycled.
```

The rank of the group is implied by the listing in tidlist, tid i in the tidlist will become rank i of the group. For fortran programmers, the first entry in the list is considered to be rank 0. During the group formation phase, rank 0 of the group requests a new context and distributes it to all the members of the group. The current operational context of the caller does not change after a call to pvm_staticgroup().

Group send and receive calls take a gid, rank and tag as arguments and route messages to/from the appropriate tid(s). On the sending side, the rank may be wildcarded and this corresponds to a broadcast to the group. On the receiving side, a wildcard rank corresponds to any source that has sent a message in the context associated with the group. This slightly queer syntax means that a broadcast can be made from outside the group without adding additional calls. The basic send and receive functions are

```
int info = pvm_gsend(int gid, int dest, int tag, int bufid)
           gid      - group identifier
           dest     - group rank of the receiver.  -1 means to
                      broadcast
           tag      - user-defined message tag
```

```
          bufid    - buffer id of the send buffer

          The message is sent in the context assigned to the group
          gid. The current operational context is not changed.

int bufid = pvm_grecv(int gid, int source, int tag)
          gid      - group identifier
          source   - group rank of the sender. -1 means
                     any tid (even outside the group).
          tag      - user-defined message tag. -1 wildcard means
                     to match any tag.
          bufid    - buffer id of the receive buffer.

          Only receives messages in the context assigned to the group
          gid. The current operational context is not changed.
```

These routines embody context and grouping of tasks into one entity which is analogous to a communicator in MPI. The central difference is that the gid, can be sent in a message to any task, and that task (presumably, one outside the group) can now send or broadcast to the group. The context assigned to the group is guaranteed to be unique across the entire machine.

There exists a set of inquiry routines for tasks to gather information about a group. These routines are local and are summarized as follows

```
int size = pvm_size(int gid, int tid)
         return the size of the group referenced by gid

int rank = pvm_rank(int gid, int tid)
         return the rank of the tid in the group

int tid = pvm_tid(int gid, int rank)
         return the tid of the group member with rank
```

A group-wide synchronization (barrier) may be called as

```
int info = pvm_synch(int gid, int msgtag)
          synchronize all members of the group.
          use msgtag for the messages to implement
          underlying barrier messages.
```

4.1 Context Details for Groups

It is natural to assign a context to a group so that messages may be encapsulated. To make the group formation safe when overlapping groups are being formed, a single reserved context is needed to encapsulate the actual messages that are used to form the group. This reserved context is known at compile time. When instance zero of any group starts the formation task, it listens for a particular tag in the reserved context. Groups are distinguished from one another by the tidlist so that a single tid may be instance 0 of any number of groups. If the exact same task are forming two groups with identical tidlists, the synchronizing feature of pvm_staticgroup() means that the first group will complete its formation task before the second one starts. Member 0 of the group is tasked with allocating a new communication context for the group. In actuality, two contexts are allocated for every group – the context for user-defined messaging and a context for barriers and other collective operations such as gather and scatter.

4.2 The Default Group

An often heard complaint from the PVM community is that tasks have no knowledge of how they came into existence. It is up to the user to determine how many tasks were started in a single spawn. Programs may be started from within another PVM program, from the PVM console (which is just a PVM program), or from the operating system prompt. The first two cases use pvm_spawn to start the new tasks. For SPMD-type programs, it would be very convenient if spawn provided information to the spawned tasks regarding the number and list of tids that were spawned together. Currently, users must create their own mechanisms for disseminating this information and usually send this information in a message to the spawned tasks from a "host" program. The following call provides this information in a single call.

```
int gid = pvm_myspawngroup(int size, int myrank)
          size - number of tasks spawned
          myrank - rank in the returned gid
```

This call is synchronous and must be called by all (or none) of the tasks started in the spawn call. The returned gid can be used in the group operations discussed previously. For example, the actual tids could be determined by iteratively calling pvm_tid. The pvm_myspawngroup call provides information and serves as a synchronization point.

4.2.1 Implementation Details for the Default Group

Once the spawning task has received return values from the remote pvmds, a contiguous list of valid tids is created and sent to the successfully spawned tasks. The spawned tasks will wait for this list of tids on the first call to pvmbeatask. Pvmbeatask is an internal routine that is called by every PVM library function to insure that a task is properly enrolled in PVM. The spawned tasks use the tidlist that has been sent by the spawner to form a static group. Pvm_myspawngroup automatically supplies the tid list to form a static group.

4.3 Group Failure

Static groups and locally-cached information introduce some coherency problems if one or more tasks within a static group fail. PVM provides a notify mechanism that informs a task if another task has failed. The task that is to be notified must explicitly "ask" for this service. We propose to allow the user to set an option that allows this notification to be turned on for static groups so that the group will be automatically dissolved if any member fails. To keep local memory requirements to a minimum, each group member i requests a single notify system to be informed if member $i-1$ exits the virtual machine. Member 0 requests this notify for member $N-1$, where N is the size of the group. If any group member receives a task exit notification, a message is sent to its right neighbor (with wrap at instance $N-1$) to propagate the error condition. The task then frees any memory used for caching the group information. The penalty for using this notification is that each group call will call pvm_probe to check for the notification and error message.

An additional notify event can be added to PVM so that any task can be informed when a context has been freed. For tasks outside of the group, this notify would set for the context associated with a gid. If the context is freed, then the group becomes invalid and local memory used for caching is freed.

4.4 Comparison to Context in MPI

Several points differentiate this context proposal with the implementation of context in MPI. First, context in MPI is generated collectively and the context tag is guaranteed to be unique only among the group and not across the entire machine. This presents no problems if communication is desired only among the group of tasks in the collective (so-called intra-communication in MPI). Some difficulty arises when a task outside of a group wants to communicate with a group (so-called inter-communication). This would occur when a group of tasks represents a collection of servers or when news tasks are spawned. Suppose two non-overlapping groups form in the same parallel program. Call these group A and group B. In MPI, it is quite possible that group A and group B will use the same context tag. In fact, if a task in group A wants to refer to group B as a collective, both groups must synchronize and form an inter-communicator. This is really a mechanism to form a new context that is unique across A and B. PVM contexts are unique and therefore eliminates the need for inter-communicators. Second, context in MPI is explicitly associated with a group of tasks. For any particular task, any context that it knows about is explicitly associated with a known group of tasks. MPI provides a multitude of functions to create new groups from old ones such as group intersection, group union and translating ranks of an existing group. However, before any communication may take place, a new group must be bound to a (new) context to form a communicator. The design presented here does not rule out or discourage this practice. It also does not require this to be the case.

5 Summary

We have outlined the concepts of context and static groups and described how they would fit into the existing PVM framework. These features can be added to the PVM in such a way that existing PVM applications are not affected.

References

[1] G. A. Geist, J. Kohl, P. Papadopoulos. Visualization, Debugging, and Performance in PVM. Proceedings of Visualization and Debugging Workshop Oct 1994. ,

[2] 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. 1991.

[3] A. Beguelin and P. M. Papadopoulos. Process Groups for Distributed Computing. Technical Report ORNL/TM-????? *In Preparation*, September 1994.

[4] Message Passing Interface Forum. MPI: A message-passing interface standard. "International Journal of Supercomputer Applications and High Performance Computing", International Journal of Supercomputer Applications and High Performance Computing, Volume 8, Number 3/4, 1994.