Process Groups for Distributed Computing

Adam Beguelin * Philip M. Papadopoulos †

December 15, 1994

Abstract

PVM (Parallel Virtual Machine) assigns a single task identification (tid) to each process running on the distributed virtual machine. The tid encodes a machine identification and in the case of a multiprocessor machine, a node identification. Tid's are assigned in such a way that processes may be spawned locally with the local message routing process (pvmd) notifying the other pvmd's after the process addition. While this is a useful and convenient method from a systems view, it does not provide a sufficiently high level of abstraction for programmers who often prefer a $0 - N$ naming scheme for spawned processes. PVM3 uses the notion of process groups to simplify the naming problem. Groups are implemented in a central server process that maps group names to (instance,tid) pairs and provides a mechanism for collective communications. Group membership is allowed to undergo changes during a program run and are termed dynamic. This ability can be especially important for long term calculations where individual computers (and hence processes) may appear and disappear because of network and/or node failures. For all their flexibility, dynamic groups (with only a single central database) may unnecessarily reduce program performance when a group is truly static in nature. We investigate methods for improving group communication performance such as caching group membership information on local hosts. In this paper we will explain the group mechanism available in PVM, compare it to some other grouping methods such as those employed in MPI, investigate the performance and fault tolerance characteristics of the dynamic group server and measure the impact of caching group information.

1 Introduction

When writing parallel programs it is often useful to have a notion of process groups to ease the task naming problem. A group is simply a set of processes attached to a particular name and usually some ordering is provided on this set. Depending on the system, group membership can be static where the group and its members are defined only once and membership is not allowed to change during execution or membership can be dynamic where changes may occur at anytime. Often, groups are used for performing collective communication such as participation in a broadcast or barrier synchronization. Several existing systems use process groups for slightly differing purposes. In PVM2 groups are the sole naming facility for processes [2]. Isis [3] uses groups for naming and fault tolerance. Bala et al. [1] use groups for collective communication. Groups in PVM3 [6] provide a dynamic naming facility for PVM tasks.

In this paper we discuss the PVM3 group facility in detail, how it is used, how it is implemented, and its performance. Since MPI (Message Passing Interface) [5] and PVM are software systems that enhance portability of parallel code, we will point out the salient differences between MPI groups and PVM groups. Also included are possible avenues for enhancing performance and results from a cached version of the group server are given.

*Carnegie Mellon University, School of Computer Science and Pittsburgh Supercompting Center

†Oak Ridge National Laboratory, Mathematical Sciences Research Section

2 Groups in PVM

In PVM2, tasks belong to exactly one group. When a task is created it is placed in a group based on a string indicating both the task's executable file name and the group name. Each task in the group is given a unique integer identifier, called a pid, upon task creation. Thus a task can be uniquely identified by the group name and pid pair. The pids for a group always start at zero and are monotonically increasing. PVM2 insures no two tasks in the same group have the same pid and group membership is static. A task is a member of exactly one group for its entire lifetime. Groups in PVM2 are global. Any PVM2 task can access a task based on its group name and pid.

The fact that task creation and group membership are intimately connected in PVM2 leads to scalability problems. When tasks are created they must, in a sense, compete for global names. A single point of control must handle the assignment of pids. There are various voting algorithms which could also be used to agree on pids but these require many more messages and is expensive in a network environment. In a voting scheme the voters must be reachable so they can participate in the election. This precludes the voting scheme as the sole method of naming because of possible network failures which can lead to deadlock. One solution is to divorce process creation from group membership. This is done in PVM3.

In PVM3, the tasks are named by using a combination of the Unix process id and the machine address to form a task id or tid. This naming scheme does not require any cooperation among existing tasks at process creation time. Advertising a process's name however is now more problematic. When new tasks are created only the parent task and the newly spawned task know the new task id. In order for tasks in the system to communicate they must know each other's tids. PVM provides special call, pvm_parent() which returns the tid of a tasks parent. This of course does not provide a simple scheme for finding the names of other tasks in the system. Groups in PVM3 provide this naming service as well as group broadcast and barrier synchronization.

The group facilities in PVM3 are similar to those in PVM2. A group is named using a string, such as "workers". When a task joins a group it is given a group id or instance. The first task to join a group will be group id zero. Subsequent tasks to join the group will have group ids one, two, and so on. PVM3 insures no two tasks have the same group id at any one time. Groups in PVM3 differ from groups in PVM2 in several ways: PVM3 tasks are not automatically in any group, tasks can be members of more than one group, and tasks can dynamically join or leave groups at any time.

The group mechanism proposed by Bala et al. [1] requires a global synchronization to create a group. All processes are inherently a part of a global group and all processes must participate in the partitioning of the global group into subgroups. Further partitioning only requires participation by the processes in the group being partitioned. The requirement that all processes participate in group creation is impractical in a system such as PVM where multiple programs can be executing in the same virtual machine. For instance, requiring the climate modeling program to synchronize with the molecular dynamics code will not do. Further complications arise upon process failure. With the PVM3 group implementation processes can fail and new processes may take their place. With the synchronous partitioning method of group creation it is unclear how failures would be handled.

The dynamic nature of groups in PVM3 makes them flexible, yet the separation of groups from process creation keeps PVM3 scalable.

2.1 Example group program

We will now present an example program to illustrate the group facilities in PVM3. In this example each task joins the "worker" group. The first task to join this group will have the group id zero. This group will spawn $NTASK - 1$ other tasks. The tasks wait on the barrier for the "worker" group. The second argument to the barrier is the number of tasks in the group that must reach the barrier before continuing. After passing the barrier, tasks broadcast a message to all other tasks in

```
#include <pvm3.h>
#include <assert.h>
#define     MTAG    1963
#define     NTASK   10
int
main()
{
    int mytid, mygid, i, tmp; int sum, tids[NTASK];
    mytid = pvm_mytid(); mygid = pvm_joingroup("worker");
    if (mygid== 0)
        pvm_spawn("ex1", 0, PvmTaskDefault, 0, NTASK-1, tids);
    pvm_barrier("worker", NTASK);
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&mygid, 1, 1);
    pvm_bcast("worker", MTAG);
    sum = mygid;
    for (i = 0; i < NTASK-1; i++) {
        pvm_recv(-1, MTAG);
        pvm_upkint(&tmp, 1, 1);
        sum += tmp;
        }
    assert(sum == ((NTASK*NTASK - NTASK)/2));
    pvm_lvgroup("worker");
    pvm_exit();
}
```

Figure 1: A simple group example: ex1.c.

the group. This message contains the group id of the sending task. Note that the pvm_bcast() call does not send a message to the calling task even when that task is part of the destination group.

Each task then computes the sum of all the group ids in the group. According to the formula $\sum_{i=0}^{n-1} i = (n^2 - n)/2$ an assertion is made to check that the sum is correct. Finally the tasks leave the group and exit.

One of the main objectives for groups is to make the programming interface easier to use. If one rewrites the previous example without using groups the code is one third longer. (See Figure 2.) Admittedly, the code size increase as a percentage would probably be smaller for a longer program, but the programmer would have to still have to manually manage the tid to an integer conversion.

There is a subtle synchronization issue that must be addressed in ex2 that is simplified in ex1 through the use of groups. In ex2 the message tag for the first multicast (pvm_mcast()) must differ from the tag for the second multicast. If these tags do not differ, a race condition will exist. If both messages use the same tag, it is possible that the first pvm_recv() may actually receive its first message from a sibling task rather than from the parent task. This problem is essentially eliminated by the use of groups and the pvm_barrier() call in ex1.

The next question to ask concerns efficiency. What is the efficiency trade off for using groups versus handling the group functionality manually? Before addressing this question, a description of the PVM3 group implementation is in order.

3 Implementing Groups in PVM3

In PVM3, groups are implemented via a group server. This server maintains group memberships and serves as a control point for barriers. The group server, pvmgs, is a normal PVM task. When a PVM task makes a call such as pvm_joingroup(), the library function sends a message to the group server. The group server performs the join group by adding the task to a list of tasks in that group and then sending the group id (or error code) back to the PVM task to be returned by the

```
#include <pvm3.h>
#include <assert.h>
#define     MTAG1   1963
#define     MTAG2   1964
#define     NTASK   10
int
main()
{   int mytid, myindx, i, tmp; int sum, tids[NTASK];
    mytid = pvm_mytid();
    if (pvm_parent() == PvmNoParent) {
        tids[0] = mytid;
        pvm_spawn("ex2", 0, PvmTaskDefault, 0, NTASK-1, tids+1);
        pvm_initsend(PvmDataDefault);
        pvm_pkint(tids, NTASK, 1);
        pvm_mcast(tids, NTASK, MTAG1);
        }
    else {
        pvm_recv(pvm_parent(), MTAG1);
        pvm_upkint(tids, NTASK, 1);
        }
    for(myindx = 0; myindx < NTASK; myindx++)
        if (tids[myindx] == mytid) break;
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&myindx, 1, 1);
    pvm_mcast(tids, NTASK, MTAG2);
    sum = myindx;
    for (i = 0; i < NTASK-1; i++) {
        pvm_recv(-1, MTAG2);
        pvm_upkint(&tmp, 1, 1);
        sum += tmp;
        }
    assert(sum == ((NTASK*NTASK - NTASK)/2));
    pvm_exit();
}
```

Figure 2: A simple example without groups: ex2.c.

pvm_joingroup() call. If multiple tasks call pvm_joingroup() at the same time for the same group, group ids will be assigned based on the arrival order of the pvm_joingroup() messages at the group server.

All of the group calls essentially follow this pattern, a message is sent to the group server and a result is returned to the calling task. The notable difference is the pvm_barrier() call. For the barrier, a message is sent to the server but a response is not received until the barrier is complete. Once enough tasks have reached the barrier, the group server sends a message (using pvm_mcast) to all the waiting tasks allowing them to continue.

For the pvm_bcast() call, a message is sent to the group server. The group server responds by returning an array of task ids currently in the group. The task calling pvm_bcast() then performs a pvm_mcast on this task id array. It would be possible to eliminate the return message by simply sending the contents of the broadcast message to the group server and allowing it to do the multicast directly. The problem with this approach is that upon receipt of a pvm_bcast() message, the sending process would appear to be the group server and not the actual sender.

We can therefore conclude that the cost of PVM3 group calls are essentially the cost of a round trip message. Is this cost too high? Could it be reduced by using caching and distributed algorithms? We argue that the cost is reasonable in light of the ease of programming objective for groups in PVM. Further, most of the cost can be eliminated either by careful placement of group calls or replacement of group calls with a combination of other PVM calls as demonstrated in the ex2 example of Figure

2.

One of the design criteria for groups in PVM3 was to implement the group calls entirely in terms of the other existing PVM calls. This allows the group code to exist independently from the PVM daemon, keeping the daemon scalable. Instead of using a group server, multicast messages could be used. This would entail multicasting messages to all known tasks when group membership changed. Besides the large number of messages involved, it is possible that a user might inadvertently receive the group control messages. Thus, a server approach was taken because it is simpler and less intrusive than a broadcast mechanism.

3.1 Groups in MPI

The MPI (Message Passing Interface) forum consisting of representatives from industry, universities and the national laboratories[5] released their final standards document in April of 1994. In this subsection, we will compare and contrast the semantics of groups in MPI and PVM.

MPI only allows static groups where PVM allows dynamic groups. Both models allow groups to be formed at anytime. PVM uses a user-specified string to identify a group where MPI has the concept of a communicator with no explicit name attached. In MPI, every process belongs to the default group (communicator) MPI_COMM_WORLD and all subsequent groups must be formed from existing groups. MPI attempts to address the very difficult issue of safety in communication space. By this, we mean that library writers can be guaranteed a safe communication environment where only messages meant for the library routine are sent to and received by the library itself. In MPI, the concept of safe communication space is termed "context" and a set of processes are termed a group. A communicator is a collection of processes operating within a certain message space or context. PVM, on the other hand, makes no such provisions for communication safety making generic libraries difficult to write. PVM programs always operate in the context of MPI_COMM_WORLD, to use MPI terminology.

The inherent design of MPI requires two classes of communicators – intra-communicators for collective operations among members of the same group and inter-communicators for communications between groups. If two groups exist and communication is desired between them, then the programmer has two choices – form a third group made up of members from both groups and use intra-communication or use an inter-communication to allow messages to flow from one group to the other. A caveat exists with inter-communicators, a process may not be a member of two groups that have been bound together by an inter-communicator or deadlock may occur. Collective operations, such as broadcast, are not allowed for groups bound together by an inter-communicator.

The most pronounced difference between MPI groups and PVM groups is illustrated by the broadcast mechanism. PVM takes the view that a broadcast is a message that is sent to all members of a group and is caught by a regular receive. Any process may broadcast a message to a group and this feature allows PVM groups to act as a pool-of-servers and still provide collective operations. MPI, however takes the view that a broadcast is only a method for distributing data among the group members and has the same semantics as a PICL [7] broadcast. Therefore, processes outside of a group may not broadcast a message to that group with a single instruction. Table 3.1 summarizes some of the key features in PVM and MPI.

Feature	**PVM**	**MPI**
Group Structure	Dynamic	Static
Context	No	Yes
Overlapping Groups	Yes	Yes. (Inter-communication forbidden)
Who may broadcast	any process	only processes within the group

Table 1: Comparing some features of groups in PVM and MPI

Skjellum has proposed the MPIX (MPI-eXtension) [9] library to eliminate some of the restrictions

Type	Time	Vlen	# loops	# Tasks
Sequential	4.10 s.	16000	100	1
No Group	2.36 s.	16000	100	4
Group	3.63 s.	16000	100	4

Table 2: Timings for the dot product program on Alpha workstations.

and add new functionality to include collective operations (barrier, scatter, gather, reduce, broadcast) when using inter-communication. Providing safe communication spaces for library routines written in PVM is clearly important and must be included in future versions for PVM to mature as a production message passing environment. An effort is underway to address these issues in PVM.

4 Performance of Groups in PVM3

In this section we examine the performance of the group calls in detail. As discussed previously, most group calls involve a round trip message with the group server. The cost of such a call can be high, particularly when the call is inside an inner loop. As an example we present three versions of a dot product program: a sequential version and two parallel versions, one using groups and one without groups.

The programs generate two vectors of length `vlen` and then perform `nloop` dot products on those vectors. For the parallel versions the vectors are generated in parallel, one block per task. The dot product is calculated by computing a dot product for the local portion of the vectors and then broadcasting the partial dot product to all the other tasks. Tasks then collect the partial dot products and sum them to obtain the final dot product. This method assures each task will have the overall dot product. Rather than broadcasting the partial sums we could have sent them to a single task that would calculate them and broadcast the result. This would probably be more efficient but we are interested in comparing the group method of programming to the non-group method. For timing purposes we calculate `nloop` dot products rather than a single dot product.

As Table 2 shows[1], the non-group program was slightly faster than the group program. This program is in some sense a worst case scenario. The group program heavily relies on the pvm_bcast() call. At each iteration of the loop all four tasks are communicating with the group server in order to do a broadcast.

4.1 Methods to Enhance Performance

The group server implemented in PVM3 has several advantages including simplicity and a modular design that does not require changing the PVM kernel. There are some disadvantages to the current design: lack of scalability and lack of fault tolerance in the server itself. The scalability issue is that a single process handles all group references and can become a communications bottleneck when using a large number of processors. The lack of fault tolerance can be efficiently addressed by using backup servers or a small collection of peer servers.

Although dynamic groups are flexible, their utility comes at a high price and begs the issue of performance. Many applications, most notably those written for use on homogeneous multiprocessors, need only static groups where membership is defined once and is not allowed to undergo change during the course of a program. The advantage of static groups is increased performance since individual processes can hold the group information and access local storage to determine group characteristics such as instance maps and group size. Synchronization primitives such as barriers

[1] The times shown here are for the entire program, including initialization of vectors to random values.

still need global communication among the group members. We investigate a compromise solution that caches group information. Caching retains all the advantages of dynamic groups and comes close to the performance of static groups but with increased complexity and overhead. Performance results for a parallel Cholesky factorization on a cluster of workstations connected by ethernet are given to show the relative value of caching.

The chief difficulty in distributed systems is maintaining cache coherency so that cached and non-cached versions of the group server give the same guaranteed ordering properties. Since we cannot rely on special hardware, the caches must be maintained exclusively by using messages within PVM. Before proceeding, it is essential that "proper group operation" be defined so that the cached version logically operates identically to the non-cached server.

Definition 1 *Proper group operation – A process P_1 will be included in (excluded from) all future operations on group G_1 when notification has been sent to P_1 that it has been added to (deleted from) group G_1.*

This definition applies equally to both the cached and non-cached PVM group servers. Notice that process P_1 is considered part of a group by *all* group members only *after* acknowledgment of the addition has been sent to P_1. Process P_1 is suspended until it receives this acknowledgment.

Caching information is not a cure-all for group performance, but it can give a significant performance boost when the virtual machine consists of workstations connected by a relatively slow network. Unfortunately, caching group information makes no impact on massively parallel processors because the cached data usually resides on a front-end processor and therefore becomes a central communication bottleneck for the MPP.

4.1.1 Design of the Cache Server

Caches are kept for every host in the virtual machine rather than on every process. The latter is more desirable, but impossible to achieve in the current PVM implementation. Some assertions in Section 4.1.2 are made as to why per-process caching is hard to achieve in PVM.

Two processes make up the cached version of the group server: a single master process and a cache process on each host. The master process holds the complete database for all defined groups and keeps track of which caches have current information for individual groups. Invalidation of a cache is done on a per-group basis. For example, if a cache process holds information for groups G_1 and G_2, then changes in G_1 will not affect the cache's ability to responds to requests regarding G_2, and vice-versa. The graphs in Figure 3 show a message cycle for the cache process and master process. Notice that both processes will block only when waiting for a message at the beginning of a logic cycle. Once a message is received, a complete logic cycle is completed. Queues to hold outstanding requests are utilized so that these processes may function in a non-blocking mode.

A single cache process is spawned on every host that requests group information. Furthermore, the cache keeps only the information that is necessary to meet requests from its local processes. A key design goal was to minimize master-cache communication. Since the master server is (for most processes) located on another machine, any messages to the group server must run "on-wire" to the remote host. When running in an ethernet environment it is possible for the on-wire communications to be serialized due to the bus-type nature of the network. By reducing the master-cache communication, we cut down on serialization by the physical media, the on-wire time for messages, and the total message volume. An example of reducing message volume can be illustrated by noting following formula:

$$T_{add} = 2(T_{onhost} + T_{offhost} + N_{valid}T_{offhost})$$

where T_{add} is the time to add the group member, T_{onhost} is the time for processess to communicate when on the same host, $T_{offhost}$ is the time for two processes to communicate when on different hosts, and N_{valid} is the number of valid caches. The first term measures the communication time between process and cache, the second between the cache and the master and the third is the

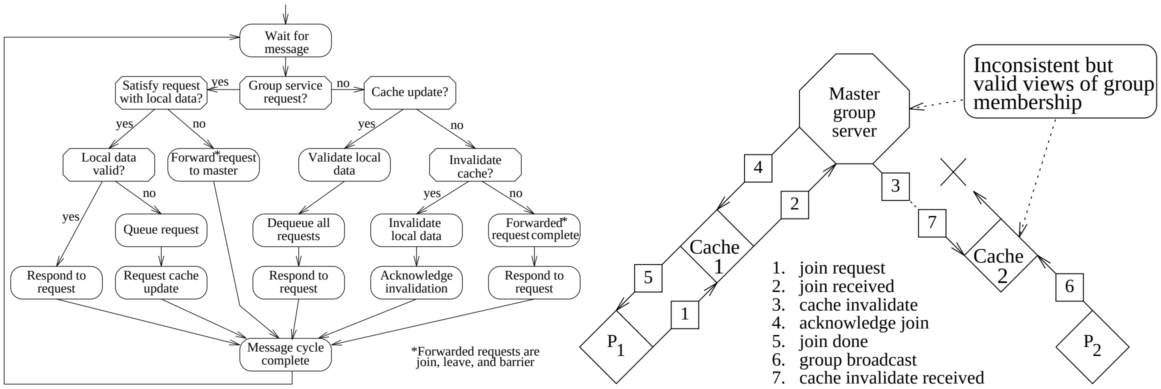


Figure 3: Cache process message flow diagram and possible race condition when cache invalidation is not acknowledged

total time to invalidate all caches. The master operates in a non-blocking mode and it is possible to receive overlapping add requests, cache invalidation acknowledgements, and cache (group list) update requests. Minimizing the number of cache invalidation cycles can be accomplished by having the master order its response to these requests – once all remote caches are invalidated for the first add request, all process adds are acknowledged and then all update requests are granted.

A second design goal was to have the cache be the only group "server" that a process sees. Hence, some requests (join, leave, barrier) that would naturally be handled directly by the master process are passed through the cache processes. Replies to these requests are passed back through the caches. This means some extra message traffic but has the benefit that the interface library to the existing group server is mostly unchanged with requests being routed to the cache instead of to the central master server. The failure modes for this system are more complicated. By allowing processes to see only a single (cache) server, the programmer need only worry about their local server failing and would handle a cache failure in exactly the same way they would handle a non-cached server failure. On the other hand, the master server should not hang waiting on an invalidation acknowledgment for a failed cache. Using PVM's notify mechanism, the current group server correctly handles group member (task) failure by removing these task from all groups to which it belonged. Cache failures can be handled in a similar way by removing failed caches from the list of validated caches.

Figure 3 shows the logic flow diagrams for the cache process. Notice that the left side of the flow diagram for the cache deals with user requests and the right side handles all system communication with the master server.

4.1.2 Why Per-Process Caching is Hard

The ideal solution for groups would be to cache information on a per-process basis instead of per-host. Although this prospect is tenable on many platforms that PVM supports, it is not possible on all. To meet the requirements of Definition 1, *all* up-to-date copies of the group information must be invalidated and a positive response to this invalidation must be received by the master process before a join or a leave request may be acknowledged.

The real question is how an invalidation message is acknowledged. Figure 3 shows a race condition if an invalidation is merely sent to a process and not acknowledged. In that figure, it is assumed that process Cache_2 holds a valid copy of the group list. The difficulty illustrated here is that the broadcast in step 6 is sent after the group join acknowledgment has been sent in step 4, but the broadcast does not include the process P_1. This means that the master server has one valid view and Cache_2 has a different valid view of group membership. This race condition does not arise in the non-cached version.

The invalidation acknowledgment may be either explicit or implicit. Our implementation uses explicit acknowledgment. However, it would be sufficient to know that a cache invalidation message had arrived at the destination process (implicit acknowledgment) and that it would be processed before the next group service call. The PVM interface guarantees that a message will be reliably delivered to an active destination process, but does not give any guarantee as to the timeliness of the message. Hence, the sending process only knows that a message will be delivered but gets no feedback as to when it gets delivered.

A second difficulty arises from the fact that a process only makes calls to the PVM library when a PVM function is needed and the PVM library has no mechanism to update its internal state variables unless the user program has called a library routine. The effect of this is that a message can get to the local memory of a process only when a PVM library call is made. Therefore a cache invalidation message may not be quickly (or ever) acknowledged by a process that is busy computing "outside of PVM." This is analogous to interrupts being handled on a polling basis. The combination of polling-only interrupts and unknown time-of-arrival for messages means that without an explicit acknowledgment from a process, the master server can not reliably know if the group data has been properly invalidated.

An obvious solution would be to use a Unix signal [8] to a process to notify it of a group invalidation by interrupting its processing. If signals were reliable and portable, not all signals are supported on the nodes of an Intel hypercube, for example, it would be possible to cause a process to immediately acknowledge an invalidation and thereby realize per-process caching.

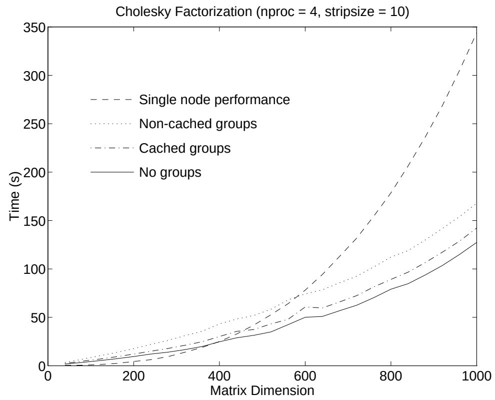
A second solution would be to use threads. One thread could wait on messages from the master server and process acknowledgments and bypass the problems with signaling mechanisms. There are two difficulties with this approach. First, a portable thread package is still unavailable for all (or even most) of the systems that PVM supports. Second, the PVM library is not thread-safe and a significant re-design of PVM would be needed.

4.2 Some Performance Results

This section gives some performance results that compare an optimized serial computation to 3 parallel versions: a version with no group server, a version using the cached server, and a version using the default group server. The example code is a parallel block version of a Cholesky factorization. The machines tested were 4 SparcClassics connected by standard ethernet and are on our section's heavily utilized main network.

The parallel implementation of the Cholesky factorization of a symmetric positive definite matrix divides the matrix into strips of equal size. See [4]. The overall message traffic in this scheme increases linearly with $4N$ group service calls being made for a matrix dimension of N. The timing curves in Figure 4 were generated by running the factorization twelve times at each matrix size, eliminating the longest and shortest times and averaging the remaining ten observations. The first graph in Figure 4 simply shows the runtimes, while the second graph shows the overhead of using a group server over that of the non-server version. The overhead demonstrates the time cost of using group service calls for broadcast and barrier instead of having the program define its own static naming scheme.

Notice that the code that does not utilize the group server is always faster because fewer messages (no round trip messages to the server) are being sent. It is interesting to note that the overhead for using the group server without caching starts at nearly 100% and drops to almost 30% when the matrix size is 1000. The cached server shows a similar drop from 36% to 11%. It is clear that caching does much better than the standard server for this range of matrix dimensions. However, as the dimension increases, the advantage of caching becomes diminished. The reason for this is that the ratio of computation to communication is increasing as the problem size increases. Therefore, optimizing the group service calls with caching or even resorting to static groups does little to change the overall run time. It is this type of observation that bolsters the case for the simple centralized group server: as the problem size increases into the regime where a parallel program outperforms a serial code, the cost of allowing dynamic groups relative to cached (or even static) information diminishes.



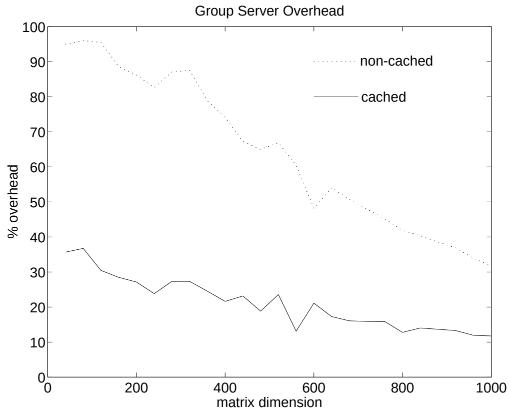


Figure 4: Cholesky factorization performance and overhead.

5 Conclusions

The group construct in PVM provides a useful and simple method to name processes and defines basic synchronization and collective operations. The group server can incur large overhead and caching group information can help alleviate some of this burden. However, the payoff of caching is not large enough to warrant making it part of the main PVM distribution. We are exploring other methods, including allowing the declaration of static groups, to increase program performance.

From the timings presented in Section 4 we can conclude that the performance cost of the group calls in a worst case scenario is not overly restrictive. In light of the ease of programming added by the group calls as shown in Section 2.1 we conclude that the group calls are a useful part of the PVM API but possibly the first place to look when attempting to tune a program for performance.

One should note that the group calls add a layer of abstraction that can be useful for fault tolerance purposes. If a task fails it is possible for another task to join the group, taking the place of the failed task. The specification of the group calls states that when a task joins a group its group id is the lowest available group id. This means if a failed task leaves the group, another task can rejoin and obtain the failed tasks group id. When tasks fail the group server automatically removes them from any groups of which they may be members. If all communication is done by group id rather than task id, then messages will be routed to the correct tasks, even upon failure and restart. Of course, this scheme assumes the programmer has taken measures to notice the failure (which can easily be done in PVM) and is able to restart a new task which can take over for the previously failed task, a difficult endeavor in itself.

References

[1] Vasanth Bala, Shlomo Kipnis, Larry Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. In Richard F. Sincovec, David E. Keys, Michael R. Leuze, Linda R. Petzold, and Daniel A. Reed, editors, *Proceedings of Sixth SIAM Conference on Parallel Processing for Scientific Computing*, volume 2, Philadelphia, 1991. SIAM.

[2] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. A users' guide to PVM parallel virtual machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[3] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[4] J.J. Dongarra and D.W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, page to appear, 1995.

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

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

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

[8] Brian W. Kernighan and Rob Pike. *The UNIX programming Environment.* Prentice Hall, Engelwood Cliffs, New Jersey, 1984.

[9] Anthony. Skjellum, Nathan .E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) library. *Integrated Computer-Aided Engineering (ICAE) Journal*, submitted, 1994.