M-PVM: An Implementation of PVM for Multithreaded and Shared-Memory Environments

CLÁUDIO M. P. SANTOS*
e-mail: claudio@nce.ufrj.br

JÚLIO S. AUDE**
e-mail: salek@nce.ufrj.br

*Federal University of Rio de Janeiro - COPPE and NCE
**Federal University of Rio de Janeiro - IM and NCE

Abstract

M-PVM is an implementation of PVM designed to work efficiently in parallel architectures supporting multithreading and the shared memory model. In particular, the current M-PVM implementation is running within MULPLIX, a Unix-like operating system designed to efficiently support parallel applications running on MULTIPLUS, a distributed shared memory parallel computer under development at the Federal University of Rio de Janeiro. M-PVM is a parallel programming library built on top of the MULPLIX parallel programming primitives. Within M-PVM, PVM tasks are mapped onto MULPLIX threads. Two approaches have been adopted for the implementation of the message passing primitives. In the first approach a single copy of the message in memory is shared by all destination M-PVM tasks. The second approach replicates the message for every destination task but requires less synchronization. M-PVM is not totally compatible with standard PVM but offers an environment which simplifies the portability of PVM applications to multithreaded shared memory platforms, in most cases, with performance improvements. Experimental results comparing the performance of M-PVM and PVM applications running on a 4-processor Sparcstation 20 under Solaris 2.5 are presented. These results show that M-PVM can produce speed-up gains in the range of 5 to 10% in relation to PVM.

Resumo

O M-PVM é uma implementação do PVM projetada para trabalhar eficientemente em arquiteturas paralelas que aceitam o conceito de multithreading e o modelo de memória compartilhada. Em particular, a implementação atual do M-PVM executa no MULPLIX, um sistema operacional Unix-like, desenvolvido para executar de modo eficiente aplicativos paralelos no MULTIPLUS, um multiprocessador paralelo de memória compartilhada distribuída em desenvolvimento na Universidade Federal do Rio de Janeiro. O M-PVM é uma biblioteca de programação paralela construída sobre as primitivas de programação paralela do MULPLIX. No M-PVM, as tasks PVM são mapeadas em threads do MULPLIX. Duas abordagens foram adotadas para a implementação das primitivas de troca de mensagens. Na primeira abordagem, uma única cópia da mensagem é compartilhada por todas as tasks destinos. A segunda abordagem replica a mensagem para cada task destino, mas exige menos sincronização. O M-PVM não é totalmente compatível com o PVM padrão, mas oferece um ambiente que simplifica o transporte dos aplicativos PVM para as plataformas de memória compartilhada multithreaded, na maioria dos casos, com melhorias de desempenho. São apresentados alguns resultados experimentais comparando o desempenho dos aplicativos M-PVM e PVM, sendo executados em uma Sun Sparcstation 20 de 4 processadores sob Solaris 2.5. Estes resultados mostram que o M-PVM pode produzir ganhos de velocidade na faixa de 5% a 10% em relação ao PVM.

1. INTRODUCTION

Parallel programming is a powerful technique to improve the performance of non-strictly sequential applications. The use of parallel programming within networks of computers is becoming more widespread in recent years due to the low cost and availability of such platforms. With the use of PVM [Geis94], the network of computers is presented to the user as a single Parallel Virtual Machine. The tasks which make up a parallel application are processed concurrently by the computers connected to the network, which communicate using the message passing model. A library of functions is responsible for providing the communication facilities among the computers and the environment for programming the parallel virtual machine.

M-PVM aims to provide an efficient PVM-like environment within MULTIPLUS [Aude96], a distributed shared memory parallel architecture under development at the Computer Center of the Federal University of Rio de Janeiro. With M-PVM, it becomes very simple to port PVM applications to MULTIPLUS and to obtain performance improvements with the porting. M-PVM is currently implemented on top of MULPLIX, a Unix-like operating system designed to efficiently support parallel applications within the MULTIPLUS platform. The MULPLIX operating system offers a parallel programming environment based on the shared memory model, the use of threads, mutual exclusion and partial ordering synchronization.

Previous efforts to develop PVM-like environments based on threads have been reported: TPVM [Ferr94], PLWP [Chua94] and LPVM [Zhou95]. PLWP and TPVM are "non-intrusive" implementations of a thread-based system for PVM. Since these systems do not change the underlying PVM system, they are not thread-safe. In addition, PVM functions are not re-entrant. Therefore, multiple threads cannot do multiple "sends" and "receives" at the same time. LPVM, on the other hand, is still in an initial stage of development and emphasizes portability aspects. LPVM tries to overcome the limitations found in TPVM and PLWP by modifying the data structures used within the shared memory version of PVM. Preliminary results on the performance of basic operations within LPVM in comparison to PVM have been presented in a recent paper [Zhou95].

In contrast with previous developments, M-PVM has been implemented without using any previous PVM implementation. The code is completely new and uses the MULPLIX native parallel programming environment to get as much benefit as possible from the use of threads and shared memory. Nevertheless, M-PVM user interface is still very similar to that of PVM. Modifications in this interface has been kept to a minimum. On the other hand, M-PVM is also quite easily portable to different platforms, since it only requires the implementation of a few MULPLIX parallel programming primitives as a library on top of the native threads environment. As a matter of fact, this porting exercise has been thoroughly carried out in a few days for Solaris LWP's and Solaris threads [Sun 95].

Section 2 briefly reviews the PVM parallel programming model. In Section 3, the main features of the MULTIPLUS architecture and of the MULPLIX operating system are presented. Section 4 describes the M-PVM environment, pointing out the major

differences to the standard PVM. Section 5 discusses the M-PVM implementation and the main data structures and algorithms used. Finally, in Section 6, performance results achieved with the use of M-PVM are presented and compared with those achieved with PVM on a SPARCStation 20 with 4 processors. Conclusions and directions for future work are also presented in this section.

2. THE PVM MODEL

PVM is an environment for parallel programming based on the message passing model. PVM allows the implementation of a virtual parallel computer from the interconnection of heterogeneous computers through a network. This feature is very attractive because low cost parallel machines can be made available very easily. The PVM processing unit is a task, which corresponds to a Unix process. Within PVM, a task can be created in any processing node of the virtual machine and the communication is performed through messages.

An important component of PVM is the *pvmd*, a program which works in background and is responsible for building the virtual computer. Each computer used in the virtual machine must have a *pvmd* process running. The user can interact with the *pvmds* to alter the configuration of the virtual computer or to follow the execution of any particular application. The PVM tasks communicate through the corresponding *pvmds*. The message sent by a particular task goes at first to the *pvmd* which is running in that computer. This *pvmd* is responsible for sending the message to the *pvmd* of the computer where the destination task is running. PVM messages are based on the use of buffers. Each piece of information is packed, according to a particular type of codification, and stored in a buffer. When the message is ready to be sent, its information is transferred to the *pvmd*. The codification of the message information allows problem free communication among tasks running in heterogeneous computers.

PVM functions are available through a library which must be linked to the user program. The main functions available within the PVM library are the following ones: buffer handling; message sending and reception; monitoring and configuration of the virtual computer; operation with task groups; barrier and global sum synchronization.

To illustrate the use of PVM, the inner product between two vectors is calculated by four processors using the master-slave approach. Each slave performs the summation of values corresponding to vector elements within a pre-defined interval. This summation is sent back to the master process which prints the total value.

```
#include <stdio.h>
#include <pvm3.h>
#define TAG 1

int     vect1 [12] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
int     vect2 [12] = { 4, 7, 8, 1, 3, 6, 5, 2, 9, 13, 17, 21};
int     tids [3], parent;

void master (void);
void slave (void);
```

```
void main (void)
{
        if ((parent = pvm_parent ()) == PvmNoParent)
                master ();
        else
                slave ();
}

void master (void)
{
        int     total, i, num;

        pvm_spawn ("slave", NULL, 0, "", 3, tids);
        for (i = 0; i < 3; i++)
        {
                pvm_initsend (PvmDataRaw);

                num = i * 4;
                pvm_pkint (&num, 1, 1);
                pvm_send (tids [i], TAG);
        }
        for (i = 0, total = 0; i < 3; i++)
        {
                pvm_recv (-1, -1);
                pvm_upkint (&num, 1, 1);
                total = total + num;
        }
        printf ("Inner Product = %d\n", total);
        pvm_exit ();
}

void slave (void)
{
        int     partial, i, start;

        pvm_recv (parent, TAG);
        pvm_upkint (&start, 1, 1);

        for (i = start, partial = 0; i < start + 4; i++)
                partial = partial + vect1 [i] * vect2[i];

        pvm_initsend (PvmDataRaw);
        pvm_pkint (&parcial, 1, 1);
        pvm_send (parent, TAG);
        pvm_exit ();
}
```

The master creates three slaves and sends them a message informing the vector interval on which they should calculate the inner product. After receiving the message, each slave starts the calculation of the partial inner product. When the calculation is complete, the slave sends a message to the master containing the partial result. The master receives the messages, performs the sum and prints the final result.

The sending of a message is carried out by the functions *pvm_initsend* (initializes the buffer), *pvm_pkint* (packs an integer value) and *pvm_send* (sends a message). The message reception is performed by the functions *pvm_recv* (receives a message and places it in a buffer) and *pvm_upkint* (unpacks an integer value). The constant *PvmDataRaw* is used to indicate that no codification needs to be used in the message. The *tids* vector stores the identification of each task created by the master.

3. THE MULTIPLUS / MULPLIX PARALLEL PROCESSING ENVIRONMENT

MULTIPLUS is a distributed shared-memory high-performance computer designed to have a modular architecture which is able to support up to 1024 processing elements and 32 Gbytes of global memory address space. The processing elements use SuperSPARC modules with 1 Mbyte of cache and up to 128 Mbytes of memory belonging to the global address space. Figure 1 shows the MULTIPLUS basic architecture. Within MULTIPLUS, up to eight processing elements can be interconnected through a 64-bit double-bus system making up a cluster. Each bus follows a similar protocol to the one defined for the SPARC MBus, but is implemented as an asynchronous bus.

The MULTIPLUS NUMA architecture supports up to 128 clusters interconnected through an inverted n-cube multistage network. Through the addition of processing elements and clusters, the architecture can cover a broad spectrum of computing power, ranging from workstations to powerful parallel computers.

As shown in Figure 1, MULTIPLUS uses a distributed I/O system architecture. It is possible to assign all processing elements within a cluster to a single I/O processor which is responsible for dealing with all I/O requests to or from mass storage devices started by these processing elements.

Some design decisions have been taken to simplify the problem of maintaining consistency among the private caches of the processing elements within the MULTIPLUS NUMA architecture. The first one is to have in every cluster one bus dedicated to instruction and data access operations and the other one dedicated to block transfer operations which occur in I/O or in memory page migration or copy operations. Only the instruction/data bus needs to be "snooped" by the cache controller and, as a result, the cache consistency problem can be solved within a cluster with the methods usually adopted in bus-based systems. In addition, a software approach based on the work presented by Kontothanassis and Scott [Kont95] has been adopted to keep cache consistency between clusters.

MULPLIX [Aude96, Azev93] is a UNIX-like operating system designed to support medium-grain parallelism and to provide an efficient environment for running parallel applications within MULTIPLUS. In its initial version, MULPLIX will result from extensions to Plurix, an earlier Unix-like operating system developed to support multiprocessing within SMP architectures [Fall89].

A major extension to Plurix included in the MULPLIX definition is the concept of ***thread***. Within MULPLIX, a thread is basically defined by an entry point within the

process code. A parallel application consists of a process and its set of threads. Therefore, when switching between threads of a same process, only the current processor context needs to be saved. Information on memory management and resource allocation is unique for the process as a whole and, therefore, remains unchanged in such context-switching operations. In relation to synchronization, MULPLIX makes available to the user synchronization primitives for the manipulation of mutual exclusion and partial order semaphores.

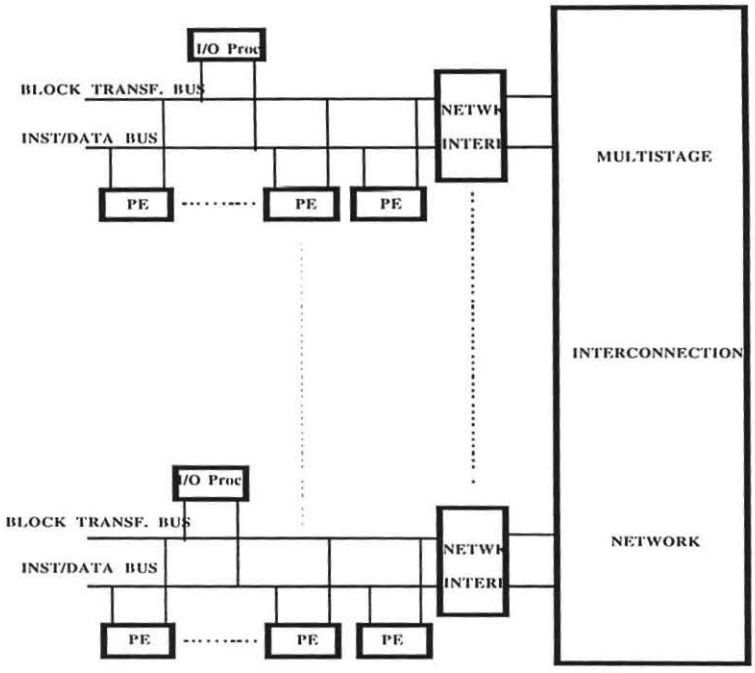

Figure 1: The MULTIPLUS Architecture

The MULPLIX system provides a set of system calls for the development of parallel programming applications within the MULTIPLUS architecture [Azev93]. These primitives deal with the following aspects: the creation of threads; memory allocation; and synchronization.

The system call, "***thr_spawn***", is provided for the creation of a group of threads. The number of threads to be created, the name of the procedure to be executed by these threads and a common argument are the basic parameters of this system call and the ones which are supported by the MULPLIX current implementation. However, one optional parameter can be added to this system call to define preferential processing elements for the execution of each thread to be created. A second version of this system call, ***"thr_spawns"***, allows the creation of threads in synchronous mode. If the thread

creation is synchronous, the parent thread will suspend its execution until execution completion by all the children threads it has started

Three additional primitives for thread control have also been made available within MULPLIX. The first one is "***thr_id***" which returns the identification number of a thread, ***tid***, within MULPLIX. The second one is "***thr_kill***" which allows any thread to kill another thread within the same process. All the descendants of the killed thread are also killed. The only parameter of this system call is the ***tid*** of the thread to be killed. The last primitive is ***"thr_term"*** which allows a forced termination of the thread.

The memory allocation primitives can perform shared and private data allocation. For shared data, the primitive "***me_salloc***" offers two options: a concentrated and a distributed memory space allocation. In the first case, it is expected that most of the accesses to the memory space to be allocated will be performed by the thread which has performed the system call and, therefore, all memory space is allocated within the local memory of the thread preferential processing element. The distributed allocation is used when a uniformly distributed access pattern among the threads is expected. The primitive which performs private memory allocation is "***me_palloc***".

The MULPLIX operating system offers two explicit synchronization mechanisms. The first one is used for mutual exclusion relations and the second one is employed when a partial ordering relation is to be achieved. For the manipulation of mutual exclusion semaphores, primitives are provided for creating ("***mx_create***"), allocating ("***mx_lock***"), extinguishing ("***mx_delete***") and releasing ("***mx_free***") a semaphore. In addition, the primitive ***"mx_test"*** allows a thread to allocate a semaphore if it is free without causing the thread to wait if the semaphore is still occupied. Simple and multiple mutual exclusion synchronizations are supported. With multiple mutual exclusion, a maximum of a given number of threads can execute the critical region simultaneously.

For partial ordering semaphores, which implement barrier-type synchronization, primitives for creating ("***ev_create***"), asynchronous signaling ("***ev_signal***"), waiting on the event occurrence ("***ev_wait***"), synchronous signaling ("***ev_swait***") and extinguishing ("***ev_delete***") an event are provided. The primitives ***"ev_set"*** and ***"ev_unset"*** have also been implemented to allow unconditional setting and resetting of an event. This may be useful in test, debugging or in error recovery procedures.

The same inner product algorithm shown in Section 2 is now presented considering the use of the MULPLIX parallel programming primitives.

```
#include <stdio.h>
#include "thread.h"

void slave (int, int);

int         vect1 [12] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
int         vect2 [12] = { 4, 7, 8, 1, 3, 6, 5, 2, 9, 13, 17, 21};
int         total;

MUTEX lock;
EVENT end;
```

```
void main (void)
{
        int     arg = 0;
        int     *map = NULL;

        total = 0;
        lock = mx_create (1);
        end = ev_create (3, 1);
        thr_spawn (3, slave, arg, map);
        ev_wait (end);
        mx_delete (lock);
        ev_delete (end);
        printf ("Inner Product = %d\n", total);
}

void slave (int arg, int ord)
{
        int     partial, i;

        for (i = ord * 4, partial = 0; i < ord * 4 + 4; i++)
                partial = partial + vect1 [i] * vect2[i];
        mx_lock (lock);
                total = total + partial;
        mx_free (lock);

        ev_signal (end);
}
```

4. THE M-PVM MODEL

The motivation to implement M-PVM within the MULTIPLUS/MULPLIX platform is rooted in the need to provide the users with a high performance and familiar parallel programming environment which would simplify the porting of parallel applications to the platform. M-PVM has been implemented using the MULPLIX parallel programming primitives and with a PVM-like user interface.

The fundamental difference between M-PVM and PVM is that the first one implements a task as a thread while the other implements it as a Unix process. This difference is the source of main incompatibilities between the two environments, because a thread is more like a C function and the automatic conversion of a Unix process into a function is not an easy task.

Within PVM, tasks share the same source code but they have different data areas in memory and different execution contexts. Within M-PVM, the tasks (threads of a same process) share the global variables of the process to which they belong.

Figure 2 shows a generic example of an application using the master-slave approach. The *status* variable is a global variable, defined both in the master and in the

slave processes. Each process has its own instance of this variable, because the data areas in memory are different. In Figure 2, the function *func_z* within the slave processes is emphasized because it handles the *status* variable.

Within M-PVM this application consists of a single process with several threads. The *status* variable belongs to the master and the slave codes. If the same name for the variable is used in these codes, consistency problems will occur if the *status* variable is a global variable within the process.

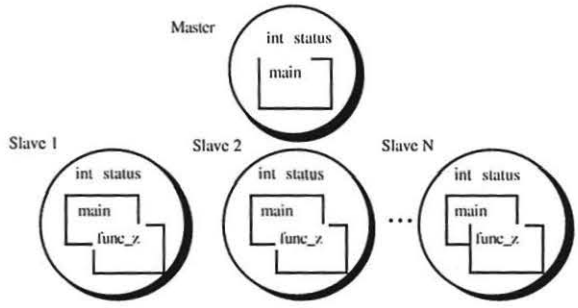

Figure 2: The PVM Algorithm

The solution presented in Figure 3 consists of creating two new global variables, *status_m* and *status_s*, to represent the states of the master and the slave processes, respectively. However, during the execution of the algorithm, several slaves will be processing concurrently (one thread per slave). Since the variable *status_s* is modified by the function *func_z* and several slaves may be running concurrently this function, the variable value may become inconsistent. Ideally, each slave should have its own *status_s* variable declared within the slave function. The problem is that C has only global or local scope. Variables declared within a particular function are not seen by other functions called by the first function. Therefore, *func_z* would not be able to access the variable *status_s* if it were declared within the slave function. Due to these problems, M-PVM cannot be totally compatible with PVM from the user point of view.

An M-PVM application consists of a single process with one or more concurrent threads. PVM *pvmd* and *console* have been eliminated within M-PVM. Message passing among threads is implemented through the process global shared memory and is managed by the M-PVM library of functions.

Since the threads share the resources within the process, as, for instance, file descriptors, the M-PVM programmer should use the multithreaded safe C library [Barr96] to avoid unexpected errors. In addition, the programmer should not use jointly the MULPLIX primitives and M-PVM.

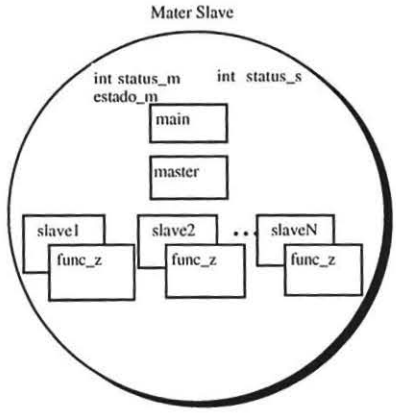

Figure 3: The M-PVM Algorithm

Within M-PVM, global variables can be used to share data among threads. This feature can be used by the programmer to avoid in some cases the need for packing and sending messages. The following example shows the M-PVM implementation of the inner product algorithm.

```
#include <stdio.h>
#include "mpvm.h"
#define TAG 1

int     vect1 [12] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
int     vect2 [12] = { 4, 7, 8, 1, 3, 6, 5, 2, 9, 13, 17, 21};
long    tids [3];
void    slave (int, int);

void    main (void)
{
        int     total, i, num;

        mpvm_spawn (slave, 0, 3, tids, NULL);
        for (i = 0, total = 0; i < 3; i++)
        {
                mpvm_recv (-1, -1);
                mpvm_upkint (&num, 1, 1);
                total = total + num;
        }

        printf ("Inner Product = %d\n", total);
        mpvm_exit ();
}

void slave (int arg, int ord)
{
        int     partial, i, parent;

        parent = mpvm_parent ();
```

```
    for (i = ord * 4, partial = 0; i < ord * 4 + 4; i++)
        partial = partial + vect1 [i] * vect2[i];

    mpvm_initsend (PvmDataRaw);
    mpvm_pkint (&partial, 1, 1);
    mpvm_send (parent, TAG);

    mpvm_exit ();
}
```

As with the PVM implementation, the master (main function) receives the partial sums through messages sent by the slaves. However, it is not necessary to send messages to inform the vector interval where each slave should work, since the algorithm uses the *ord* parameter, taking advantage of a facility provided by the MULPLIX environment. This parameter tells the order of that particular thread within a group of threads. It is also interesting to note that even if *vect1* and *vect2* were read by the master, it would not be necessary to send them to the slaves, because they are global vector variables. This would not be true within the standard PVM environment.

5. THE M-PVM IMPLEMENTATION

The M-PVM library has been implemented only for C programs. In general, the M-PVM functions display the same user interface as the PVM functions. The only exception is the function *mpvm_spawn* which has been adapted for the use with threads.

As the MULPLIX operating system is still not available within the MULTIPLUS architecture, it has been developed with the use of Solaris 2.5 LWPs and Solaris threads [Sun95], libraries of functions which implement the MULPLIX parallel programming primitives to validate the M-PVM environment. Then, M-PVM has been transported, without any alteration in its code, to a SPARCStation 20 with 4 processors.

The PVM original functions had to be completely rewritten to work using shared memory and the concept of threads. The information on M-PVM tasks is stored in a global process area to which the tasks have access through their *tid* (task id). This avoids consistency problems, since one task does not interfere with information belonging to another task because they have different *tids*.

Two approaches have been adopted for the implementation of M-PVM in relation to the way messages are stored and exchanged between threads. The first approach follows the usual trend for the implementation of message passing environments within shared memory platforms [Shek96]. This approach consists of sharing the message in memory among threads in such a way that the sending of a message to several threads does not imply the replication of the message information in memory. Within the message body a reference counter is used. The message can only be eliminated if no pending references are left.

The implementation of this first approach is very advantageous for sending messages, since instead of copying the message it is only necessary to create in the destination task waiting list a reference to the message. It is necessary, however, to use

operations protected by mutual exclusion to access the message reference counter and the task waiting list.

Another important issue is the way messages are built. Every message consists of a list of fragments. Each fragment is individually allocated in memory according with the packing procedure. In the reception of a message this list is traversed and the information is unpacked.

Such implementation, however, has shown poor performance results on the Solaris environment for some applications which demand synchronization operations very often, since the cost of synchronization at the Light-Weight Process level within Solaris is very high [Vaha96]. By analyzing the execution time of each M-PVM library function, it was possible to verify that memory allocation and synchronization operations had a large contribution to the total execution time of some applications.

Shared memory allocation within multithreaded environments is an operation which has to be protected by mutual exclusion synchronization. Therefore, such operations may imply a considerable overhead since, in this first approach, memory fragments are dynamically allocated. Accesses to the message reference counter are another overhead source since they also require synchronization operations.

To overcome these problems, a second approach has been adopted for the implementation of M-PVM. Within this approach the minimization of situations requiring synchronization has been set as a target. In addition, memory space has been dynamically allocated in larger blocks. The M-PVM library has assumed the management of such blocks in order to avoid the frequent and unnecessary operations of releasing and acquiring memory space by reusing previously allocated memory blocks. The size of such blocks has been determined through experimental tests considering different situations of use of the M-PVM library.

Within this second implementation approach, the replication of messages has been adopted in the sending procedure. Therefore, the reference counter could be eliminated reducing the need for synchronization. In addition, the fragment list has also been eliminated since the message is packed in a single buffer to facilitate the copy process.

With these modifications, in this second approach it is only necessary to have synchronization within message passing operations when the task waiting list is accessed.

6. EXPERIMENTAL RESULTS AND CONCLUSIONS

M-PVM is already operational and initial evaluation tests have been performed using the implementation for a SPARCStation 20 with four HyperSPARC 100MHz processors. The experimental results have been derived from the use of the M-PVM implementation based on the MULPLIX parallel programming primitives available as a library of functions on top of Solaris LWPs.

Tables 1 and 2 compare the performance of PVM and the two implementation approaches of M-PVM for a Gaussian elimination algorithm applied to a 100x100 and a 1000x1000 array, respectively. The application has been programmed with 1 to 4 tasks, which are implemented as processes within PVM and as threads within M-PVM. For the 1000x1000 array a test with the use of 5 tasks has also been performed to evaluate the overhead caused by the necessary context-switching that will take place in the processor which has two tasks assigned to it.

The master-slave approach has been used for the parallelization of the application. The array has been divided into groups of columns assigned to different tasks. The master task also participates of the computation. Only the time spent on the reading of the data by the master task has been considered because otherwise there would be a big disadvantage for the PVM solution, since, in this case, the array data would have to be explicitly sent to the slaves by the master. Within M-PVM the data is shared among the threads and, therefore, there is no overhead in sending data from the master to the slaves. This data sharing, however, is not a source of conflict since each thread works with a different set of array columns.

In Tables 1 and 2, the first approach used for the implementation of M-PVM, as described in Section 5, is referred as M-PVM 1 and the second approach as M-PVM 2. Execution times in seconds representing the average value for five runs of the application are displayed in both tables. The results achieved with the sequential version of the algorithm are also shown in the Tables for comparison.

Number of tasks	PVM	M-PVM 1	M-PVM 2
2	0.401	0.338	0.343
3	0.396	0.340	0.346
4	0.493	0.358	0.356
Sequential		0.419	

Table 1: Gaussian Elimination for a 100x100 array - time in seconds

Number of tasks	PVM	M-PVM 1	M-PVM 2
2	84.84	78.54	79.31
3	69.71	63.53	65.71
4	61.65	58.01	59.37
5	66.64	58.45	59.23
Sequential		120.55	

Table 2: Gaussian Elimination for a 1000x1000 array - time in seconds

The results in Table 2 show that for a larger problem, all three environments have benefited from increasing the number of tasks up to a limit of 4, which is the number of processors available. With the use of 5 tasks, Table 2 shows that PVM has had a poorer performance due to the overhead caused by context-switching at the process

level. This overhead has had a much smaller impact on M-PVM performance, since context-switching occurs at the thread level.

In general, the results in both tables show that M-PVM can produce a reduction of the application execution time in the range of 5 to 10% when compared to that achieved with the use of PVM. This reduction can potentially be much more significative when an implementation of M-PVM within MULPLIX is available for experimental tests, since, in this case, the overhead for the use of the MULPLIX parallel programming primitives will be considerably reduced. M-PVM 1 has shown a slightly better performance than M-PVM 2 in all cases.

The second application used for performance evaluation has been the parallelization of a Genetic Algorithm when applied to the placement of cells in a VLSI circuit [Knop96]. This application has been originally developed for an Ethernet cluster of IBM 25 T workstations using PVM. Then, the application has been ported by its own developer to PVM and M-PVM running on the SPARCStation 20. This exercise has shown that in a couple of days it has been possible to complete the porting of the application to M-PVM by an experienced PVM programmer who had no previous experience of using M-PVM.

Table 3 shows the results achieved for the execution time of this application when four tasks are used. As the application uses broadcast operations very heavily and the messages consist of a big number of fragments within M-PVM 1, the results achieved with M-PVM 1 are very poor. This is mainly due to the overhead in synchronization and memory allocation operations as described in Section 5. In fact, the results achieved with this experiment have been the motivation for the implementation of M-PVM 2 under a different approach.

Number of tasks	PVM	M-PVM 1	M-PVM 2
4	10.26	22.37	9.68

Table 3: Execution Time in seconds for the Genetic Algorithm

Tables 4 and 5 show the results achieved with the SOR algorithm (Splash Benchmark) when applied to arrays with sizes 200x200 and 300x300, respectively. In both cases, the initial temperature assigned to all elements of the array is set zero. To the element sitting on the center of the array a temperature value equal to 100 is applied. This temperature value is propagated through the array by evaluating the temperature on each element as the average of the temperature values assigned to its 8 neighbours. The elements on the borders of the array have their temperatures fixed to zero. Therefore, the high temperature in the center of the array initially spreads over a region of the array, but, in the end, the temperature on all the elements in the array goes back to zero.

For this application, M-PVM 1 displayed slightly better results (around 5%) than M-PVM 2. M-PVM 1 is approximately 10% faster than PVM for this application considering the use of 2 or 4 concurrent tasks.

Number of tasks	PVM	M-PVM 1	M-MPVM 2
2	28,35	25,75	26,28
4	24,50	21,78	22,28
Sequential		48,88	

Table 4: Execution Time in seconds for the SOR Algorithm with a 200x200 array

Number of tasks	PVM	M-PVM 1	M-MPVM 2
2	167,03	156,73	163,31
4	124,40	114,58	122,07
Sequential		309,51	

Table 5: Execution Time in seconds for the SOR Algorithm with a 300x300 array

Table 6 shows the results achieved with the parallelization of the formal proof procedure applied to the comparison of behavioral descriptions of logic equations given by Binary Decision Diagrams (BDD's). Circuits given in the MCNC benchmark have been used. The algorithm has been implemented with the use of the master-slave approach. The master sends to three slaves the logic equations describing the circuit behaviour and only these three slaves take part in the computation of the formal proof procedure. For the largest circuit (imec7), both M-PVM 1 and M-PVM 2 displayed an execution time nearly 20% smaller than PVM. For the small circuits, the performances of PVM, M-PVM 1 and M-PVM 2 were similar. This result shows that the performance of PVM is very sensitive to the size of the transmitted messages since the sizes of the logic equations are much bigger in the imec7 circuit than in the other two cases. On the other hand, both M-PVM implementations do not have their performance affected by the message size because of the use of shared memory.

	Sequential	PVM	M-PVM 1	M-PVM 2
apex1	17,90	11,98	11,86	11,90
apex4	16,26	13,14	12,80	12,94
imec7	517,95	478,44	401,23	401,60

Table 6: Execution Time in seconds for the Formal Proof Algorithm

The experiments shown in Tables 1, 2 and 3 have also been performed for an implementation of M-PVM using MULPLIX primitives built as a library on top of Solaris threads [Sun95]. In all cases, the performance results have been considerably worse than those achieved with the use of Solaris LWP's.

Current work with M-PVM is focused on the optimization of the system for NUMA architectures such as MULTIPLUS. In addition, we intend to compare the performance of M-PVM with LPVM [Zhou95] and to enhance the portability of M-PVM by making an implementation available with the use of Pthreads [IEEE94].

7. ACKNOWLEDGMENTS

The authors would like to thank FINEP, CNPq, RHAE and FAPERJ, in Brazil, for the support given to the development of this research work.

8. REFERENCES

[Aude96] Aude, J. S., et. al., The MULTIPLUS/MULPLIX Parallel Processing Environment", Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks - I-SPAN 96, pp. 50-56, Beijing, China, May 1996

[Azev93] Azevedo, R.P., Azevedo, G.P., Silveira, J.T.C, Aude, J.S., "Parallel Programming Primitives within MULTIPLUS (in Portuguese)", Proceedings of the V Brazilian Symposium on Computer Architecture, Florianópolis, pp. 761-775, September 1993

[Barr96] Barros, M. O., Aude, J. S., "Implementation of Multithread Libraries in the Mulplix Operating System (in Portuguese)", Proceedings of the VIII SBAC-PAD - Recife, PE - August 1996

[Chua94] Chuang, W., "PVM Light Weight Process Package", Laboratory of Computer Science, Massachussets Institute of Technology, Computation Structures Group Memo 372, December, 1994

[Fall89] Faller, N., Salenbauch, P., "Plurix: A multiprocessing Unix-like operating system", Proceedings of the 2nd Workshop on Workstation Operating Systems, IEEE Computer Society Press, Washington, DC, USA, pp. 29-36, September 1989

[Ferr94] Ferrari, A., Sunderam, V.S., "TPVM: A Threads-Based Interface and Subsystem for PVM", Computer Science Technical Report CSTR-940802, Ermory University, August, 1994

[Geis94] Geist Al, Beguelin A., Dongarra J., Jiang W., Mancheck R., Sunderam V., "PVM - A users guide and tutorial for Network Parallel Computing", 1994, The MIT Press, Cambridge, Massachusetts.

[IEEE94] Institute of Electrical and Electronics Enginners, POSIX P1003.4a, "Threads Extension for Portable Operating Systems", 1994

[Knop96] Knopman, J., Aude, J.S., "Parallelization of Genetic Algorithms Applied to the Placement Problema in Workastation Cluster", (in Portuguese) - Proceedings of the VIII SBAC-PAD, Recife, PE , August 1996

[Kont95] "High Performance Software Coherence for Current and Future Architectures", L.I. Kontothanassis, M.L. Scott, Journal of Parallel and Distributed Computing, Vol. 29, No. 2, September 1995, pp. 179-195

[Shek96] Shekhar, S., Chubb, D., Turner, G., "Parallelizing GIS on a Shared Address Space Architecture", IEEE Computer, Dec. 1996, pp.42-48

[Sun 95] Sun Microsystems Inc., "Multithreaded Programming Guide - Solaris 2.5", 1995

[Vaha96] Vahalia, U., "Unix Internals - The New Frontiers", Prentice Hall, 1996

[Zhou95] Zhou, H., Geist, Al. "LPVM: A Step Towards Multithread PVM", Technical Report, Oak Ridge National Laboratory, June 1995