PM-PVM: A Portable Multithreaded PVM

C. M. P. Santos* J. S. Aude**
*NCE and COPPE, Federal University of Rio de Janeiro, Brazil, e-mail: claudio@nce.ufrj.br
**NCE and IM, Federal University of Rio de Janeiro, Brazil, e-mail: salek@nce.ufrj.br

Abstract

PM-PVM is a portable implementation of PVM designed to work on SMP architectures supporting multithreading. PM-PVM portability is achieved through the implementation of the PVM functionality on top of a reduced set of parallel programming primitives. Within PM-PVM, PVM tasks are mapped onto threads and the message passing functions are implemented using shared memory. Three implementation appproaches of the PVM message passing functions have been adopted. In the first one, a single message copy in memory is shared by all destination tasks. The second one replicates the message for every destination task but requires less synchronization. Finally, the third approach uses a combination of features from the two previous ones. Experimental results comparing the performance of PM-PVM and PVM applications running on a 4-processor Sparcstation 20 under Solaris 2.5 show that PM-PVM can produce execution times up to 54% smaller than PVM.

1. Introduction

Symmetric multiprocessing (SMP) is currently common in nowadays desktop computers. However, its potential parallelism is still very little exploited. PVM [1], which is widely used in the development of parallel applications for heterogeneous computer networks, can be an adequate environment to encourage and simplify the development and migration of parallel applications on and to SMP platforms. Unfortunately, however, most of the PVM implementations are not easily portable and do not fully use the facilities available in these systems to improve the performance of parallel applications.

PM-PVM (*Portable Multithreaded* PVM) is an efficient and portable PVM implementation for Unix-like SMP environments, which maps PVM tasks onto threads and implements PVM message passing functions using shared memory. Therefore, PM-PVM also supports a hybrid parallel programming model based on message passing and global shared variables.

PM-PVM portability is achieved by implementing the full PVM functionality on top of a reduced set of parallel programming primitives supported by Mulplix, a Unix-like operating system designed to run parallel applications within Multiplus, a DSM multiprocessor [2]. Therefore, to port PM-PVM to different SMP platforms, it is only necessary to implement this reduced set of primitives on such platforms. In fact, this has already been done for *Solaris LWPs*, *Solaris threads* [3] and P*threads* [4].

PLWP [5], TPVM [6] and LPVM [7] are previous efforts to develop PVM-like environments based on threads. PLWP is a non-intrusive implementation of a PVM thread-based system designed to work with a specific thread model. TPVM is also a non-intrusive implementation. It does not support shared memory among threads and its functions for sending or receiving messages are not re-entrant. LPVM implementation modifies data structures used in PVM shared memory versions and provides re-entrant message passing functions. However, the PVM user-interface is modified in order to generate a thread and signal safe environment.

PM-PVM, on the other hand, is not based on previous PVM implementations. Its new code tries to get as much benefit as possible from the use of threads and shared memory. Nevertheless, PM-PVM and PVM user interfaces are very similar, which simplifies the migration of existing PVM applications to SMP platforms.

Section 2 of this paper describes the PM-PVM programming model and discusses the additional facilities offered by its hybrid parallel programming model. Section 3 gives a brief description of the Mulplix primitives used in the PM-PVM implementations. In Section 4, three different PM-PVM implementation approaches are presented. Section 5 discusses the performance results achieved with the use of PM-PVM and PVM on a *SparcStation* 20 with 4 processors. The first group of tests evaluates the performance of the implementations of some PVM operations and Mulplix primitives using *Solaris LWPs* and *Solaris threads*. The second group of tests compares the performance of PVM and the different PM-PVM implementations for four parallel applications. Finally, Section 6 summarizes the main conclusions and presents proposals of future work.

2. The PM-PVM Model

A PVM application consists of a set of tasks running in parallel on a virtual machine, typically a heterogeneous computer network. A PVM task has a unique identification (*tid*) and is mapped onto a process within Unix-like systems. The tasks communicate among themselves through message passing. A message consists of the destination task identification, the message tag and the list of data fragments. The communication is performed with the use of buffers. Each data fragment to be sent is packed according to some kind of codification and stored in a buffer. At the receiving end, it is firstly stored in a buffer and then unpacked. The PVM library provides functions for: buffer handling; sending/receiving messages; defining the virtual machine configuration; etc.

PM-PVM has been designed to be a portable PVM-like environment optimized to run on SMP platforms with support to multithreading. PM-PVM implements the PVM functions with the use of a reduced and simple set of primitives derived from the Mulplix operating system. Instead of being mapped onto processes, PM-PVM tasks are mapped onto *threads*. Hence, a PM-PVM application consists of a single process made up of threads which run in parallel, share the process resources and communicate among themselves using shared memory, but following the message passing model at the user level.

PM-PVM is almost fully compatible with PVM. Basically, only the *pvm_spawn* function has had its user interface changed in order to become compatible with the Mulplix thread model. Due to the compatibility issue, PM-PVM keeps the idea of packing and unpacking message data through buffers, but with no data encoding.

PM-PVM supports global shared variables among its tasks. Global variables can be used to store the initial values of an application data base, eliminating the need to send this information to the tasks through messages. If each task has exclusive access to a given fraction of the data base, the results of the task processing can also be stored directly into the global variable area. This kind of hybrid parallel programming model is specially useful in applications which follow the master-slave approach.

3. The Mulplix Primitives

Mulplix set of parallel programming primitives is used by PM-PVM to implement PVM functions based on threads and shared memory. Therefore, to port PM-PVM to different SMP platforms, it is only necessary to implement this simple and reduced set of primitives on top of some available thread package for such platforms.

The Muplix primitive, "***thr_spawn***", creates a group of threads. Its parameters are the number of threads to be created, the name of the function to be executed by the threads, an optional list of preferential processing elements for the execution of each thread and a common argument. A synchronous creation of threads is performed with the ***"thr_spawns"*** primitive.

Three additional primitives for thread control are: ***"thr_id"***, which returns the unique identification number of a thread; ***"thr_kill"***, which allows any thread to kill another thread within the same process; and ***"thr_term"***, which forces the termination of the thread.

Dynamic shared memory allocation can be performed with the use of the ***"me_salloc"*** primitive. The ***"me_sfree"*** primitive is used to release previously allocated regions of shared memory.

Mulplix supports synchronization mechanisms based on mutual exclusion and partial ordering relations. For the manipulation of mutual exclusion semaphores, primitives are provided for the creation (***"mx_create***"), allocation (***"mx_lock***"), non-blocking allocation (***"mx_test"***), destruction (***"mx_delete***") and release (***"mx_free***") of a semaphore. For partial ordering semaphores, primitives are provided for waiting on the event occurrence ("***ev_wait***") or for creating (***"ev_create***"), asynchronous signalling (***"ev_signal***"), synchronous signalling ("***ev_swait***") and destroying ("***ev_delete***") an event.

4. Implementations of the PM-PVM Model

Within PM-PVM, information on the tasks is stored in the *task control vector*, which is indexed by the task identification (*tid*). As the *tid* is a unique task identifier, no access collision can occur and this data structure can be implemented as a simple global vector. PM-PVM functions call the *thr_id* function to obtain the *tid* before accessing information on a task within this vector.

Three different approaches have been used for the implementation of PM-PVM. In the first one, named PM-PVM1, a message is shared by all destination tasks. There is no message replication. Each task is initially associated with a vector of 4 buffers. Additional buffers are created in groups of 4. A buffer holds the message source task identification, the message tag and a pointer to the message. The message holds a pointer to its list of data fragments. A new fragment is allocated in memory for every packing operation. The message also holds the *reference counter*, which stores the number of references to it. Access to the reference counter is protected by a lock. In addition, each task has a pointer to the received message queue, where message information is stored until a receive operation is performed.

The second approach, PM-PVM2, aims at optimizing the message passing procedure by reducing PM-PVM1 synchronization overhead in handling the reference counter and in allocating memory for the data fragments. Within PM-PVM2, data fragments are stored in a single vector, which is dynamically allocated in 256-byte blocks. Therefore, additional packing operations will only require memory allocation if the previously allocated blocks have no room left to store the new data fragments. In addition, PM-PVM2 eliminates the message reference counter by copying the message to the destination tasks. PM-PVM2 buffer structure includes the data fragment vector and all information held by PM-PVM1 message structure. When a message is sent, the full contents of the buffer is copied to the destination task. The received message queue is implemented through a linked list of buffer structures.

The third approach, PM-PVM3, is a combination of the previous ones. The contents of a message is shared by all destination tasks, as in PM-PVM1, but, on the other hand, the message data fragments are stored in a vector of bytes as in PM-PVM2. PM-PVM1 message structure is preserved, but with a pointer to the data fragment vector. This way, PM-PVM3 eliminates some synchronization needed in PM-PVM1 by not using a fragment list and avoids the PM-PVM2 message replication overhead.

5. Performance Evaluation

PM-PVM implementations have been developed and tested on a *Sparcstation 20* with 4 HyperSparc@100MHz processors and 128 Mbytes of memory. All programs have been compiled using GNU C [8] compiler with all optimization options enabled. PVM version 3.3.11 for *Solaris/Sun* has been used in these experiments. In this version, the communication among PVM tasks is performed using *Unix streams sockets*.

The experimental tests have been performed for PM-PVM implementations based on *Solaris LWPs* and on *Solaris threads*. All results refer to the smallest execution time in seconds achieved in at least 10 executions of the test programs. This minimizes the interference on the measurements that might be caused by other programs that were eventually running on the same machine.

Initially, the performances of the implementations of the basic Mulplix primitives have been evaluated. The experiments have shown that the implementation based on *Solaris threads* of the *thr_id* primitive, which is often used by the PM-PVM functions, is over 6 times faster than that based on *LWPs*. In addition, implementations based on *Solaris threads* perform thread context-switching faster and do not require explicit synchronization for handling dynamic memory allocation, another frequently used operation. However, considering the creation of threads, the implementation based on *Solaris threads* is 7 times slower than that based on *LWPs* and it is also slightly slower when performing semaphore lock/unlock operations.

Experimental tests have also been used to evaluate the performance of the basic PVM operations. These tests have shown that all implementations based on *Solaris threads* have a worse performance than those based on LWPs when creating a large number of tasks, as already expected. However, they are at least 80 times faster than PVM. Implementations based on LWPs can even be 250 times faster than PVM. The tests have also shown that PM-PVM1 performs worse than PVM, PM-PVM2 and PM-PVM3 when packing multiple fragments. The time spent by PM-PVM1 on 1M packing operations of a single integer is at least 4 times longer.

Concerning message transmission, PVM has shown to be at least 40 times slower in sending and 25 times slower in receiving empty messages than any of the PM-PVM implementations. Both PM-PVM1 and PM-PVM3 are nearly twice slower than PM-PVM2 in receiving empty messages. However, PM-PVM2 is almost 50 times slower than both PM-PVM1 and PM-PVM3 in sending messages consisting of a single fragment with 10K integers due to the time spent on memory allocation and message copy.

5.1 Performance of parallel applications

The performances of PVM and the different PM-PVM implementations have also been analysed considering four parallel applications: SOR, Bubble Sort, Gaussian Elimination and VLSI placement using Genetic Algorithm. In all applications, the tasks have been evenly distributed among the four available processors. For the PM-PVM implementations based on *Solaris threads*, four *LWPs* have always been created, one on each processor, with one or more tasks mapped onto each *LWP*. The PM-PVM versions using *Solaris threads* are identified in the tables showing test results by the termination st.

5.1.1 SOR. This application simulates the heat propagation process on a surface described by a 192x192 array and having its borders always at 0° C. Initially the whole surface temperature is set to 0^o C. The center of the surface is then heated to 100^o C for a short time period (one single iteration). At each iteration, the temperature on each position is evaluated as the average temperature on its 8 neighbours. The algorithm runs for 238 iterations until an equilibrium situation is reached. The parallel version of the SOR algorithm uses the master-slave approach and each task processes a set of rows.

# of tasks	2	4	8	12	16
PVM	29.46	28.31	28.44	42.56	47.31
PM-PVM1	23.06	20.48	18.63	19.89	20.51
PM-PVM2	23.61	20.57	18.84	19.79	20.42
PM-PVM3	23.17	20.53	18.61	19.95	20.49
PM-PVM1st	-	19.89	17.26	16.96	17.12
PM-PVM2st	-	19.77	17.27	16.57	16.23
PM-PVM3st	-	19.74	17.26	16.82	16.90
Sequential	40.00				

Table 1: Performance of the SOR algorithm

Table 1 shows that PVM has the worst performance because it processes message passing operations much slower than PM-PVM. With more than 8 tasks, PVM performance gets much worse since the task idle time is too small to make process context switching worth while. However, with PM-PVM, context switching is faster and the performance improves with 8 tasks. Performance also improves with 12 tasks for PM-PVM implementations based on *Solaris threads* and even with 16 tasks when PM-PVM2st is used, since the message length decreases as the number of tasks increases.

With the PM-PVM hybrid parallel programming model, an optimized implementation has been produced by avoiding the master to send sets of rows to the slaves at each iteration. The array representing the surface is defined as a shared variable and, since each slave operates on different rows, they are accessed without mutual exclusion synchronization. Message lengths are reduced, but the number of messages does not change to ensure the correct task synchronization at each iteration.

# of tasks	2	4	8	12	16
PM-PVM1	21.48	18.70	16.48	17.59	18.19
PM-PVM2	21.14	18.33	16.12	17.04	17.19
PM-PVM3	21.47	18.66	16.44	17.47	17.88
PM-PVM1st	-	18.85	15.99	15.50	15.16
PM-PVM2st	-	15.86	13.79	13.22	13.12
PM-PVM3st	-	18.85	15.92	15.33	15.25

Table 2: Optimized SOR implementation

Table 2 shows the results produced with the optimized SOR implementation. The smaller message lengths improve the performances of all PM-PVM implementations. This is more evident with PM-PVM2, which is more sensitive to the message lengths.

5.1.2 Bubble sort. The *bubble sort* parallel implementation uses a master-slave approach and works in two phases. The first one consists of dividing a vector with 96000 integers in sections and sorting them in parallel. In the second phase, the ordered sub-vectors are repeatedly merged together in pairs by a decreasing number of parallel tasks.

Table 3 shows that with 2 and 4 tasks, the algorithm execution time is reduced with the square of the number of tasks, since the *bubble sort* algorithm complexity is $O(n^2)$. With more tasks, the performance improvement is not so big, since the tasks start competing by processors. With 16 tasks PVM performs worse than with 12 tasks and much worse than any PM-PVM implementation. In addition, the message length increase, as the algorithm progresses, also makes PVM performance worse.

# of tasks	2	4	8	12	16
PVM	104.76	26.92	14.58	10.78	15.56
PM-PVM1	104.17	26.23	13.28	8.98	6.84
PM-PVM2	104.15	26.34	13.36	9.10	6.99
PM-PVM3	104.17	26.25	13.28	8.96	6.84
PM-PVM1st	-	26.27	13.27	8.93	6.81
PM-PVM2st	-	26.33	13.38	9.04	6.98
PM-PVM3st	-	26.24	13.25	8.95	6.83
Sequential	466.86				

Table 3: Performance of the Bubble Sort algorithm

5.1.3 Gaussian elimination. This application performs the Gaussian elimination of a 960 x 960 matrix of real numbers and evaluates its determinant. The initial task creates as many other tasks as required and sends to them the range of columns onto which they should work. The range of columns is associated with the tasks according to their *tids*. Every task reads the full matrix from an input file. During the Gaussian elimination procedure, the task holding the current pivot element calculates and sends the corresponding set of multipliers to the tasks with bigger *tids*. Then, each task performs the necessary calculations and sends its partial determinant result to the initial task that finds the final determinant value.

# of tasks	2	4	8	12	16
PVM	44.50	26.92	24.16	25.35	28.50
PM-PVM1	44.11	25.32	22.10	21.27	21.73
PM-PVM2	44.28	25.88	23.92	24.86	26.07
PM-PVM3	44.18	25.24	22.07	21.56	21.54
PM-PVM1st	-	25.26	19.23	18.10	17.78
PM-PVM2st	-	25.87	19.79	19.30	19.92
PM-PVM3st	-	25.27	19.13	18.24	17.70
Sequential	82.24				

Table 4: Gaussian Elimination Performance

Table 4 shows that PVM and PM-PVM have similar performances for up to 4 tasks. From then on, PM-PVM1 and PM-PVM3 perform better, indicating that they benefit more from context switching between tasks. With PM-PVM2, however, performance gets worse with more than 8 tasks because the number of messages increases

when the messages are longer. As expected, implementations based on *Solaris threads* get more benefit from increasing the number of tasks.

5.1.4 Genetic algorithm. This application aims at solving the placement problem in VLSI design and uses a distributed processing model where each processor runs the full genetic algorithm on a fraction of the total population [9]. The original PVM implementation has been ported to PM-PVM using a hybrid parallel programming model in which some of the initialization messages have been eliminated with the use of shared data structures among the tasks.

Two test circuits have been used: Circuit_1 (80 modules and 30 nets) and Circuit_2 (100 modules and 300 nets). Table 5 shows the performance results for the placement of both circuits, considering a population with 1024 individuals. The parallel versions achieve an excellent speed-up because the evolution of populations in parallel finds "fitter" individuals earlier in the process, leading the algorithm to the optimum result faster.

	Circuit_1		**Circuit_2**	
# of tasks	**4**	**8**	**4**	**8**
PVM	14.91	29.53	37.11	74.18
PM-PVM1	22.03	39.15	43.13	76.15
PM-PVM2	8.69	17.45	30.45	61.54
PM-PVM3	14.11	25.54	35.03	69.79
PM-PVM1st	23.00	40.61	43.68	79.41
PM-PVM2st	8.71	17.64	30.41	61.29
PM-PVM3st	15.96	25.84	36.38	70.38
Sequential	128.11		715.52	

Table 5: Performance of the Genetic Algorithm

PVM performs better than PM-PVM1 and slightly worse than PM-PVM3. However, PM-PVM2 performs much better than PVM. PM-PVM3 loses with the small amount of message sharing since much time is spent on allocating and releasing memory for the messages and it is not worth while to maintain the reference counter. With PM-PVM1, there is also loss in performance due to the packing of the several data fragments that make up typical messages in this application. PM-PVM2, on the other hand, can re-use allocated memory regions.

With 8 tasks, all performance results get worse. This shows that the algorithm idle time is negligible and, therefore, no benefit is achieved by using PM-PVM implementations based on *Solaris threads*.

6. Conclusions

In all tests with parallel applications, at least one of the PM-PVM implementations has performed much better than PVM with execution times up to 54% smaller. In applications where the tasks are idle for some time, PM-PVM implementations based on *Solaris threads* normally perform better since context-switching is faster.

PM-PVM2 performs better in applications with a small number of short messages. PM-PVM3 and PM-PVM1 perform better than PM-PVM2 in applications with effective message sharing, such as broadcasting intensive applications. In contrast with PM-PVM2, PM-PVM3 performance is less sensitive to the message size. PM-PVM1, however, does not perform well in applications issuing several messages with multiple fragments.

Future work will mainly focus on the optimization of PM-PVM implementations for cluster-based distributed shared memory architectures such as Multiplus and on the integration of PM-PVM with PVM in order to provide a scalable and efficient environment for use in heterogeneous networks of SMP workstations.

7. Acknowledgements

The authors would like to acknowledge the support given to the development of this research work by FINEP, FAPERJ, CNPq, RHAE and CAPES.

8. References

[1] *PVM - A users guide and tutorial for Network Parallel Computing*, Geist A., et al., The MIT Press, Cambridge, Massachusetts, 1994.

[2] *The Multiplus/Mulplix Parallel Processing Environment*., Aude, J.S., et al., *Proc. ISPAN'96*, Beijing, China, May 1996, pp. 50-56.

[3] *Multithreaded Programming Guide - Solaris 2.5*, SunSoft, Inc, 1995.

[4] *Threads Extension for Portable Operating Systems*, Posix P1003.4a, IEEE, 1994

[5] *PVM Light Weight Process Package*, Chuang, W., Laboratory of Computer Science, MIT, Computation Structures Group Memo 372, December 1994.

[6] *Multiparadigm Distributed Computing with TPVM*, Ferrari, A., Sunderam, V.S., *Concurrency: Practice and Experience*, Vol. 10(3), March 1998, pp. 199-228

[7] *LPVM: A Step Towards Multithread PVM*, Zhou, H., Geist, Al., *Concurrency: Practice and Experience*, Vol. 10(5), April 1998, pp. 407-416

[8] *Using and Porting GNU CC*, Stallman, R. M., Cambridge, USA, Free Software Foundation, 1993.

[9] *Parallelization of Genetic Algorithms Applied to the Placement Problem in Workstation Clusters*, Knopman, J., Aude, J.S., *Proc. VIII SBAC-PAD*, Recife, Brazil, Aug 1996.