(Quasi-) Thread-Safe PVM and (Quasi-) Thread-Safe MPI without Active Polling*

Tomas Plachetka

University of Paderborn, Department of Computer Science
Fürstenallee 11, D-33095 Paderborn, Germany

Abstract. PVM (the current version 3.4) as well as many current MPI implementations force application programmers to use active polling (also known as busy waiting) in larger parallel programs. This serious problem is related to thread-unsafety of these communication libraries. While the MPI specification is very careful in this respect, the implementations are not. We present a new mechanism of interruptable blocking receive which makes PVM and MPI quasi-thread-safe. This mechanism does not require any modifications to the existing semantics of PVM or MPI (we only extend the interfaces with two new functions) and allows writing multi-threaded programs without active polling. Then we sketch how the interrupt mechanism can be hidden in the implementations of PVM and MPI, making both PVM and MPI completely thread-safe without active polling. Results of our experiments promise a significant speedup for all larger communication-intensive parallel applications.

Keywords: PVM, MPI, thread-safety, polling, latency, efficiency

1 Introduction

We define a *non-trivial parallel application* as an application consisting of parallel processes which all perform two independent tasks shown in Fig. 1. Distributed databases, media servers, shared memory simulation libraries and parallel scientific computations using application-independent load balancing libraries are a few examples of important non-trivial parallel applications.

T_1 CPU intensive computation, communicating with other processes
T_2 Fast servicing of requests coming from other processes (in order to provide the other processes with data, or to balance the load, or to handle fatal errors, ...)

Fig. 1. Two independent activities of one process of a non-trivial parallel application

* Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and by the grant VEGA 1/7155/20

Non-trivial parallel applications based on the current implementations of PVM [4] (PVM 3.4.4) or MPI [3] (e.g. MPICH 1.2.3, ScaMPI 1.13.7) are forced to use *active polling*. Active polling means that the heavy computation in T_1 gets interrupted "now and then" to check for incoming requests in T_2. If there is no incoming request, then the computation in T_1 is resumed; if there are some requests, they are serviced, and then T_1 is resumed. Here is why it is bad: if "now and then" means often (say, every 1 microsecond) then CPU cycles are unnecessarily wasted every 1 microsecond when there are no incoming requests to service; if "now and then" means seldom (say, every 1 minute) then incoming requests do not get serviced for a minute in the worst case. An obvious compromise is to use a "more reasonable" time period than 1 microsecond or 1 minute (in order to achieve a balance between the wasted CPU time and the message-passing latency). However, the "reasonable" constant must be tuned for every application and for every system on which the application is supposed to run. (Moreover, even a tuned application can exhibit a nondeterministic behaviour.)

Our goal is to avoid active polling in non-trivial applications, not only inside communication libraries. We propose an interrupt-driven mechanism which allows writing non-trivial applications without active polling. This mechanism can either be implemented as an extension to the PVM and MPI interfaces (we refer to leaving the thread synchronisation to the user as to *quasi-thread-safety*) or it can be completely hidden in the PVM and MPI implementations, yielding the highest level of thread-safety defined by the MPI standard (*complete thread-safety*, see `MPI_THREAD_MULTIPLE` in [3])—multiple threads may call PVM/MPI functions, with no restrictions.

We focus on PVM with socket-based communication in this paper. We are aware of different implementations of PVM and MPI and also of different implementations of the MPI specification itself—however, where a reasoning applies to both PVM and MPI (meaning the implementations of the standards), we write "PVM/MPI" throughout the text. We are convinced that the techniques explained in Section 4 can be applied to any implementation of PVM and any implementation of MPI (no matter whether sockets or shared memory are used for inter-process communication).

Section 2 compares our work to previous efforts. Section 3 discusses options of implementing non-trivial applications using PVM. In Section 4 we explain why non-trivial parallel applications cannot be written without active polling using the current PVM/MPI implementations. We present a new interrupt mechanism which makes PVM/MPI quasi-thread-safe and allows avoiding active polling in non-trivial applications. This mechanism is an extension to the PVM/MPI interfaces (two new functions are added to the existing interfaces). The original functionality and performance of PVM/MPI are fully preserved. In the same section we also sketch how the interrupt mechanism can be hidden inside PVM/MPI, making the libraries completely thread-safe, without active polling. Our theoretical considerations are supported by experiments in Section 5.

2 Related Work

Thread-safety of PVM and MPI has been studied for many years. We mention only a few research papers here, the complete list would be too extensive.

LPVM (Lightweight-process PVM) system was introduced in [7]. LPVM is designed for shared-memory machines. PVM tasks are implemented as threads in LPVM. A definition of thread-safety is given in the paper: “A program is thread-safe when multiple threads in a process can be running that program successfully without data corruption. A library is thread-safe when multiple threads can be running a routine in that library without data corruption.” The authors recognize two main issues that have to be dealt with to make PVM multi-thread-safe: *global state and reentrancy.* LPVM removes global states from the PVM library by assigning receive and send buffers (and other resources) to each task. The user interface of PVM (3.3) is only slightly modified but a major redesign of `libpvm` is needed. Our implementation is simple and does not require a removal of global states.

TPVM [2] is a different approach which assumes a thread to be the basic unit of parallelism in a distributed system. There is a thread server registering all threads running in the system. This fine-grained model is mapped onto the coarse-grained process model of PVM for the purpose of message passing. Rather than going into technical details, we shall explain the problem of TPVM from the point of view of a non-trivial application (see Section 4.1 in [2] for more details). There is a global message queue accessible to all threads in each process. A thread wanting to receive a message follows the following protocol. First it looks for the message in the global message queue. If the message is there, the thread continues; if not, the thread attempts to receive a message from another TPVM task using a nonblocking receive. If a message is there, but it is addressed to another thread, the thread stores the message in the global message queue and retries the nonblocking receive (and later wakes up threads waiting for those messages); otherwise it falls asleep. Here is the weakness of the protocol: When a thread falls asleep, then **another** thread must attempt to receive a message in order to wake up the sleeping thread. If there is no such thread, *the sleeping thread will sleep forever*—even though there might be a message addressed to the sleeping thread sent by some other TPVM process. This situation can be resolved by running a special thread that regularly polls for incoming messages and wakes other threads up. Our mechanism does not require running a polling thread.

A PVM library supporting the concept of active messages is described in [6]. The library uses a signal handling mechanism in order to detect a change on the set of receiving socket file descriptors. This idea is close to what the MPICH’s `ch_p4` driver does. However, it does not solve the problem of non-trivial applications.

MiMPI [1], IBM’s MPI and ScaMPI claim to be fully thread-safe. It is difficult to judge without further information how the problem of thread-safety is solved in the implementations. Active polling is used inside ScaMPI.

3 Programming Non-trivial Applications with PVM

This section discusses different programming techniques for implementing non-trivial applications using PVM 3.4 and explains why active polling cannot be avoided.

3.1 PVM and Thread-Safety

Threads are a natural way of implementing the tasks T_1 and T_2 from Fig. 1 to run simultaneously (concurrently) in the scope of a single PVM task (a single process). Fig. 2 a shows a pseudo-code of a generic non-trivial parallel application. We restrict the communication of thread T_1 to sending messages but this restriction is not important. We implemented the pseudo-code of Fig. 2 a. It does not work. The messages sent by `pvm_send` of the thread T_1 get never delivered. PVM does not allow a thread to send a message while another thread is blocked in `pvm_recv`. This is a consequence of "*PVM is not thread-safe*".

Fig. 2 b shows how this problem can be overcome. Thread T_2 uses active polling to check for incoming messages. Thread T_1 is allowed to call `pvm_send` only during inactive periods of T_2 (when T_2 is blocked in the `sleep` call). This code works; however, it is very poor for the reasons explained in Section 1. Unfortunately, as we show in the following, the pseudo-code of Fig. 2 b is the only way to implement the desired functionality.

3.2 Handlers which Do Not Get Fired Unless They Are Told to Do so

Message handlers have been introduced in PVM 3.4 (see also [5]). They *seem* to provide an alternative way of implementing the scenario of Fig. 2. We briefly explain what message handlers are and what is wrong with them.

```
Thread T1                  Thread T2
while (not_done)           while (not_done)
  compute();                 recv();
  send();                    handle_message();
```

a) Natural implementation, not working

```
                 mutex comm;

                          Thread T2
                          while (not_done)
Thread T1                   lock(comm);
while (not_done)            arrived=probe();
  compute();                while (arrived)
  lock(comm);                 recv();
  send();                     handle_message();
  unlock(comm);               arrived=probe();
                            unlock(comm);
                            sleep(time);
```

b) Active polling, working but poor

Fig. 2. Threaded implementation of one process of a generic non-trivial application

PVM 3.4 users are encouraged to create own message handlers using the function

`pvm_addmhf(int src, int tag, int ctx, int (*func)(int mid))`

which registers a user's message handler `func`. The handler `func` is fired when a matching message arrives (the message header must match the parameters `src`, `tag` and `ctx`).

An elegant implementation of the task T_2 from Fig. 1 would be registering a set of message handlers for T_2, getting even rid of threads (the only running thread would be T_1). However, the manual to `pvm_addmhf` says (note the marked text): "... `pvm_addmhf` specifies a function that will be called *whenever libpvm copies in a message* whose header fields of src, tag, and ctx match those provided to `pvm_addmhf()`". In other words, *the message handlers are called only when the application asks for it* (by calling `pvm_recv` or `pvm_probe`, or `pvm_send`, ...) If the task T_2 is implemented using message handlers registered by `pvm_addmhf`, then T_2 will not receive any message until T_1 wants to communicate!

Using message handlers for implementing non-trivial applications therefore requires running a thread that calls `pvm_probe` (or some other PVM function) at regular time intervals. This equals to active polling.

3.3 Lazy Signaling

PVM allows delivering signals between tasks. Signals, in combination with message handlers, can be used to get rid of the polling thread mentioned in Section 3.2. The scenario of Fig. 1 would work as follows (we present a simplified version here, the actual implementation can be very complex):

1. Process A sends a message to process B
2. Meanwhile, process B runs T_1, not noticing the message
3. Process A sends a signal to process B, saying "Get up, you got a message!"
4. Process B gets the signal and fires a signal handler. The signal handler calls `pvm_probe` which fires a message handler that receives the message and performs an appropriate T_2's action.

Note that there is no active polling in the above scenario. However, technical issues make this approach *inefficient and non-portable* and restrict the use of PVM to *homogeneous parallel machines.*

4 (Quasi-) Thread Safe PVM/MPI without Active Polling

The reasons why the scenario from Fig. 1 cannot be efficiently implemented using the current standards are:

1. Tasks T_1 and T_2 must be implemented as threads.
2. T_2 must run a blocking receive.

3. T_1 cannot send any messages while T_2 is blocked in the blocking receive if the communication library (PVM or MPI) is not thread-safe.
4. It seems to be difficult from the software engineering point of view to implement the PVM and MPI libraries in a reentrant way without active polling.

In the following section we first describe an *interrupt mechanism* which is missing in PVM and in MPI and which *allows* to avoid active polling in non-trivial multi-threaded parallel applications—this is what we call *quasi-thread-safety*. Quasi-thread-safety leaves the correct synchronisation of threads to the user (or to a library built on top of quasi-thread-safe PVM/MPI). Then we sketch *how to make PVM/MPI completely thread-safe*, hiding the interrupt mechanism in the implementations of the standards.

4.1 Interruptable Blocking Receive and Quasi-Thread-Safety

Let us return to one of the reasons of PVM/MPI being thread-unsafe unless active polling is used (point 3): "T_1 cannot send any messages while T_2 is blocked in the blocking `recv` if the communication library (PVM or MPI) is not thread-safe." In terms of concurrent programming, T_2 is waiting inside a critical section of a blocking `recv` which does not allow T_1 to enter *its* critical section of a `send`. Our idea is to let T_1 *interrupt* T_2 for the time necessary for completing the `send` call. T_2 must be blocked outside of its critical section during the interrupt. Care must be taken to not let T_1 enter its critical section before T_2 has left its critical section. At the end of the interrupt (after the `send` call has returned), after T_1 has left its critical section, T_2 must return to exactly the same state in which it was before the interrupt.

To implement the above mechanism, we extended the PVM (3.4) and MPI (MPICH 1.2.3, `ch_p4`) libraries with two new functions. The extensions require only a few changes in the implementation of PVM/MPI (literally, a few lines of code), while the original functionality does not change at all (in case of PVM only the implementation of the library is slightly modified, the PVM daemon is not changed). This is the C interface and semantics of the new functions:

`void interrupt_recv(void)`
Blocks the calling thread until the thread that is blocked in a `recv` has blocked outside of its critical section and then returns the control to the calling thread. (If there is no thread blocked in a `recv`, the control never returns to the calling thread.)

`void resume_recv(void)`
Makes the thread blocked outside of the critical `recv`'s section reenter its critical `recv`'s section and then returns the control to the calling thread.

The extended PVM/MPI remain generally thread-unsafe. A random sequence of concurrent calls to PVM/MPI functions results in an undefined behaviour. We do not intend to call the functions randomly. We restrict ourselves to calling PVM/MPI functions in a well defined order that avoids both thread-safety problems and active polling when writing non-trivial parallel applications. This is possible to achieve with the extended PVM/MPI and this is why we call the extended libraries quasi-thread-safe. To simplify writing of programs that use only the well defined order of PVM/MPI calls, we developed a programming paradigm implemented as a library (TPL, Thread Parallel Library) that uses the strategy described at the end of this section.

Fig. 3 depicts a modified scenario of Fig. 2 a, revealing the integration of `interrupt_recv` and `resume_recv` in PVM/MPI in more detail. This scenario avoids active polling as well as thread-safety problems in PVM/MPI. The trick is based on sending a fake "message" from a process to itself. We assume a socket inter-process communication here (but the same interrupt mechanism can be implemented for shared memory communication as well). The implementation of `recv` in PVM/MPI causes T_2 to block in a `select` call on a set of file descriptors connected to sockets (leading to other processes in the network). We extend this set of file descriptors with the read-end of a synchronous POSIX pipe `intr_fd`. The function `interrupt_recv` called by T_1 simply writes to the write-end of `intr_fd`. This `write` causes T_1 to block until the read-end of the (synchronous) pipe has been read. The read-end of `intr_fd` becomes readable, the `select` in `recv` unblocks and T_2 bails out of its critical section. After that, `recv` reads from `intr_fd` and T_2 gets blocked in the following `read`. Now the thread T_1 gains control and can safely send a message. After the message has been sent, T_1 calls `resume_recv` which is implemented in Fig. 3 in exactly the same way as `interrupt_recv`. `resume_recv` writes to the write-end of `intr_fd`. This unblocks the second `read` in `recv` and T_2 gets eventually blocked in the `select` of its critical section again. (The implementation of "bailing out of the critical `recv` section" is tricky. At the time of writing we have a solution for socket-based PVM 3.4 and MPICH's ch_p4 driver.)

Here is a sketch of how PVM/MPI can be made completely thread-safe, without a loss of efficiency caused by active polling:

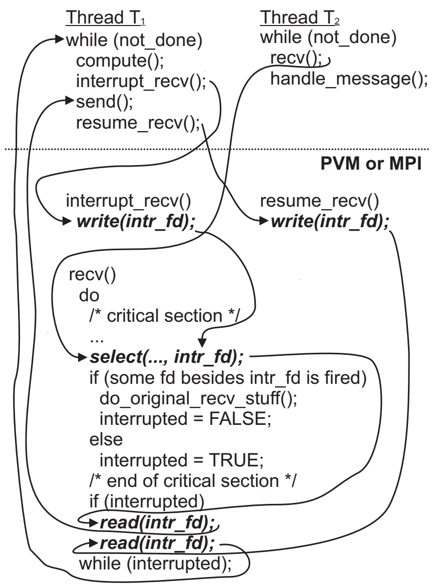
– Each PVM/MPI process runs a thread (let us call it *main thread*) that is (almost always) blocked in a blocking `recv`, listening to all sockets from where a message might arrive. When a message arrives, it is stored by main thread in a global message queue (or message queues).
– The message queue is protected by a mutex.
– Each `send` is preceded by `interrupt_recv` and followed by `resume_recv`.
– All `send` calls are mutually excluded (using a mutex).
– The user's code runs as a thread (or as several threads if the application itself is multi-threaded). User's `recv` calls are implemented as functions that access the message queue, not file descriptors. A `recv` gets blocked on a semaphore when it did not find the requested message.
– All PVM/MPI functions are divided into collision classes. Each class contains functions that internally modify the same global memory structures (or cause some other racing conditions when called simultaneously). Functions of one class are mutually excluded (using a mutex).

– Thread addressing should not be a part of PVM/MPI interface. It is up to the user to develop a thread addressing scheme (if it is needed in the application).

5 Experiments

We used two versions of the pingpong benchmark in our experiments. The pingpong application consists of two processes, P_1 and P_2. In both versions the process P_2 runs a loop in which it first receives a message and then answers it. The two threaded versions of the process P_1 are depicted in Fig. 2 b (*POLL*, the active polling version, based on thread-unsafe PVM/MPI) and Fig. 3 (*QTS*, the event-driven version based on quasi-thread-safe PVM/MPI). In both versions there is no `compute()` call in T_1 and no `handle_message()` call in T_2. We used a `nanosleep` of 50 milliseconds in the active polling version (as we show later, the choice of this constant is optimal—a finer setting has the same effect).

Table 1 shows the time measurements in which the process P_1 sent (and received) 100,000 messages of the length 10,000 Bytes. Each measurement was repeated 10 times. "Maximum deviation" is the absolute maximum deviation from the average time (in seconds). We also observed the number of `nanosleep()` calls which explains the timing differences between the polling versions of the


```
Thread T1
while (not_done)
  compute();
  interrupt_recv();
  send();
  resume_recv();

Thread T2
while (not_done)
  recv();
  handle_message();
```

```
interrupt_recv()
  write(intr_fd);

resume_recv()
  write(intr_fd);

recv()
  do
    /* critical section */
    ...
    select(..., intr_fd);
    if (some fd besides intr_fd is fired)
      do_original_recv_stuff();
      interrupted = FALSE;
    else
      interrupted = TRUE;
    /* end of critical section */
    if (interrupted)
      read(intr_fd);
      read(intr_fd);
  while (interrupted);
```

Fig. 3. Scenario of Fig. 2 a using the interrupt mechanism of quasi-thread-safe PVM/MPI. The synchronisation points (where a thread can possibly block) are typeset in ***slanted boldface***. A synchronous pipe `intr_fd` is used here for the thread synchronisation

Table 1. Pingpong benchmark, 100,000 messages

Communication library	PVM	PVM	MPICH	MPICH	ScaMPI	ScaMPI
Pingpong version	POLL	QTS	POLL	QTS	POLL	TS
Average time (in seconds)	120	**21**	42	**26**	98	339
Maximum deviation (in seconds)	± 66	± **3**	± **1**	± **1**	± 40	± 81
Number of `sleep()` calls	2,370	**0**	583	**0**	1,533	0 (?)

benchmark (see below). The data were measured on a double-processor 850 MHz Intel Pentium 3 running Linux Redhat. We compared PVM 3.4, MPICH 1.2.3 (ch_p4 driver) and ScaMPI 1.13.7. ScaMPI is thread-safe, therefore we could also use the code from Fig. 2 a (*TS*) for a direct comparison with the quasi-thread-safe MPICH's version. All times are absolute execution times (the timing starts right before the receiving loop in the process P_1 and stops right after the loop).

Note that the active polling versions of the pingpong benchmark are *profiting* from not calling `sleep` (`nanosleep`) very often. A continuous stream of messages sent by the process P_2 allows the process P_1 to pull the messages from the network one after another without falling asleep after each `recv`. The deviations in the measured times by the POLL versions are caused by the varying number of `sleep()` calls in the individual runs of the benchmark. If the stream of messages is not continuous, then the latency of each `recv` followed by a `sleep` depends *solely* of the timer's resolution and the process/thread switching overhead in the operating system (not of the speed of the processors or the network connecting them). Without a special tuning of the kernel the latency is 0.02 sec (the `time` argument of the `nanosleep(time)` call is not important). This can be measured by calling `nanosleep(1 microsecond)` many times in a loop.

6 Conclusions

We showed that the current PVM implementation and some of MPI implementations force a whole class of important parallel applications to use active polling. We also showed the limitations of active polling in non-trivial communication-intensive applications. We explained a new interrupt mechanism that makes PVM as well as MPI quasi-thread-safe (quasi-thread-safety means a possibility of writing non-trivial applications without active polling) or, with a little more effort, completely thread-safe. No change in the interfaces of PVM or MPI is needed in order to achieve complete thread-safety of both standards without active polling. We demonstrated the potential of our interrupt mechanism on a pingpong benchmark. The benchmark is an abstraction of all non-trivial parallel applications. Even in a comparison to a highly optimised active polling we measured significant speedups when we used the new interrupt mechanism of quasi-thread-safe PVM 3.4 and quasi-thread-safe MPICH 1.2.3. Moreover, we measured a speedup of 13.5 in a direct comparison to the commercial thread-safe MPI implementation by Scali.

Acknowledgements

We would like to thank Burkhard Monien and all the people in his working group, among them especially to Axel Keller for his perfect technical support and Ulf-Peter Schroeder for his valuable hints. Our thanks go also to Branislav Rovan and Peter Ružička at the Comenius University in Bratislava who many years ago turned our attention to parallel and distributed programming.

References

[1] F. García, A. Calderón, and J. Carretero. MiMPI: A multithread-safe implementation of MPI. In *Proceedings of PVM/MPI 99, Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes on Computer Science*, pages 207–214. Springer Verlag, 1999. 298

[2] A. Ferrari and V. S. Sunderam. TPVM: Distributed concurrent computing with lightweight processes. *Concurrency—Practice and Experience*, 10(3):199–228, 1998. 298

[3] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. 1997. 297

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine (A User's Guide and Tutorial for Networked Parallel Computing)*. The MIT Press, 1994. 297

[5] A. Geist, J. A. Kohl, P. M. Papadopoulos, and S. Scott. Beyond PVM 3.4: What we've learned, what's new and why. In *Proceedings of PVM/MPI 97, Recent Advances in Parallel Virtual Machine and Message Passing Interface, 4th European PVM/MPI Users' Group Meeting*, volume 1332 of *Lecture Notes on Computer Science*, pages 116–126. Springer Verlag, 1997. 299

[6] K. R. Subramaniam, S. C. Kothari, and D. E. Heller. A communication library using active messages to improve performance of PVM. *Journal of Parallel and Distributed Computing*, 39(2):146–152, 1996. 298

[7] H. Zhou and A. Geist. LPVM: A step towards multithread PVM. *Concurrency—Practice and Experience*, 10(5):407–416, 1998. 298