Thread-Safe Message Passing With P4 and MPI*

Aswini K. Chowdappa Anthony Skjellum Nathan E. Doss†

Department of Computer Science &
NSF Engineering Research Center for Computational Field Simulation†
Mississippi State University
PO Box 6176
Mississippi State, MS 39762

Abstract

P4 (Portable Programs for Parallel Processors) is a popular message passing system. The Pthreads library is a POSIX-standard implementation that supports multiple flows of control, called 'threads' within a process. MPI(Message Passing Interface) is the emerging message passing system which will soon be the industry standard system. This paper illustrates using multiple threads within the P4 processes and thread-safe message passing. It also describes the various issues that have to be looked into when dealing with the two packages (P4 and Pthreads). We demonstrate thread-safe message passing by means of some test programs. Finally we identify areas where MPI is potentially unsafe in a multithreaded environment. We delve into the details of these issues and discuss introducing multi-threaded message passing into the MPICH implementation in the near feature.

1 Introduction

The multiprocessing paradigm is a widely used computational model that helps in fast and efficient solutions for algorithmic implementations. With the advent of multiprocessing, whether as a network of workstations or a massively parallel multiprocessor, intense research efforts are being concentrated on developing a wide variety of tools to help support programming these multiprocessors. These tools mainly help increase the ease and flexibility of use of these machines. P4 (Portable Programs for Parallel Processors) and MPI (Message Passing Interface) are such parallel programming tools [2, 7, 10].

*Work supported by the NSF Engineering Research Center for Computational Field Simulation, Mississippi State University.

A thread can be broadly defined as a single flow of control within a process. Using multiple threads within a process can enhance performance in many situations, particularly to mask latency. When the importance and effectiveness of making applications multithreaded became apparent, a wide variety of thread libraries became available. Pthreads library is a IEEE-POSIX standard implementation [5].

We give a brief introduction to P4, MPI and Pthread libraries. We describe the desirable features of a thread-safe environment. The potential nonreentrant functions are identified and appropriate fixes to enforce thread-safety are discussed. Each of these changes to P4 have been enumerated. Finally we discuss the same issues as applied to MPI and suggest possible fixes. Since the MPICH implementation uses P4, the work on P4 will speed on work with the MPICH implementation [6].

1.1 P4

P4 is primarily a library of subroutines. P4 was developed at Argonne National Laboratory and supports both shared memory and message passing operations, besides XDR support for programming heterogeneous networks. P4 also has features to support error/interrupt handling [2, 3]. This work uses version 1.3.

1.2 MPI

MPI (Message Passing Interface) was defined and developed to fill the need for a standard, portable message passing system [7]. Its development was largely influenced by the fact that, numerous message passing systems are in vogue with similar goals and end-user audience. But most lack at least a few of the features that applications demand. A standard high level interface such as MPI greatly enhances the ease of writing portable message passing code without compromising on functionality and efficiency [7, 10]. This work utilizes the MPICH implementation carried out jointly at Mississippi State University and Argonne National Laboratory [6].

MPICH is implemented on the lower level message passing system P4, and thus facilitates the usage of uniform message passing calls. Currently there are few implementations available [4, 8, 9, 13]. The first consideration to ensure thread safety in MPICH is to make sure that the lower level message passing system on which MPICH is implemented (P4) is thread-safe.

1.3 Threads and Thread Libraries

A thread, as defined earlier is a single flow of control within a process. A multithreaded process has independent threads of control that share the same address space, file descriptors, IPC ports etc., but each thread has its own thread id, stack, local variables, scheduling priority and policy and program counter. Since the threads share the same address space the context associated with the thread is small, threads are sometimes also called light-weight processes [12].

Writing multithreaded applications provides better performance in many situations some of which are as follows:

- when computing can be overlapped with I/O: By using two threads we can perform I/O as well as continue computation at the same time.
- When computing jobs are frequently created and destroyed: Since threads are light-weight processes, context switches between threads is possibly much less expensive when compared to context switches between processes [1].

Currently, thread libraries provide the means for writing multithreaded programs. Using thread libraries is advantageous in many respects. User-level thread libraries provide synchronization primitives that help synchronize access to shared data. These libraries also provide machine-independent calls thus facilitating portable codes. Some of the earlier thread libraries include Cthreads, Brown threads from Brown university and Mach threads supported by the Mach operating system [1, 11, 15]. The IEEE POSIX committee introduced a standard for the thread libraries, which defined some basic operations that any thread library needed to support. These operations included thread management (initialize, create, join, destroy), synchronization (mutual exclusion, condition variable, semaphores), and signal handling. An implementation that conforms to the POSIX standard is known as Pthreads library [5]. This work used version 2.1 of the implementation carried out at Florida State University [12].

2 Objectives

This work mainly has the following objectives:

- To identify and correct the non-reentrant functions in P4.
- To propose a reliable 'thread-safe model', that is to guarantee atomic access to these functions by means of locking mechanisms.
- To demonstrate thread-safe message passing by means of some simple test programs.
- Finally, to identify the potential problem areas in MPI relating to thread-safety and discuss possible fixes. Also, to discuss changes to the programming model needed for effective use of threads.

3 Concept of thread safety

For an application to be thread-safe it should use reentrant functions; that is, a function that may be safely invoked by multiple threads concurrently without leading to erroneous results, race conditions or deadlocks. Multiple threads executing within a process may access data pertaining to its parent process. Uncontrolled access to this shared data may lead to any of the conditions mentioned above. This problem is similar to the concurrency control problem in database systems.

Threads within the processes of a message passing system such as P4, execute the code of the user application as well as the functions of the message passing system. Hence we would have to ensure atomic operations on the data that is local to the process. Thus P4 must become knowledgable about its thread system.

Since we also have chosen to work with a version of MPICH that is implemented on P4, the first objective is to ensure that P4 is thread-safe. Figure 1 shows the hierarchy of layering of these systems.

4 Implementation Issues

This section discusses the technical details of the modifications needed for P4 and MPI to handle

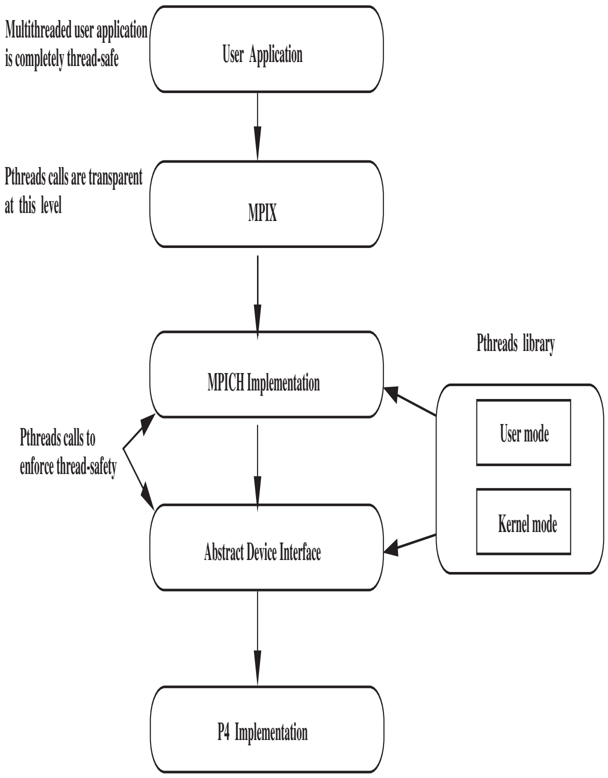

Figure 1: Message passing system hierarchy

thread-safe message passing.

4.1 P4

Our initial task was to make the system calls such as `malloc` and `free` safe. The version of P4 maps the `malloc` call to the `malloc` system call. This works fine for the p4 processes without multiple threads. When multiple threads execute `malloc` they might be allocated the same address space. So this call had to be made thread-safe. Version 2.1 of the Pthreads implementation provides thread-safe malloc [12]. We mapped the original `p4_malloc` and `p4_free` to the thread-safe versions of these calls `pthread_malloc` and `pthread_free`. Similarly the mutual exclusion locks were mapped to the thread-safe locks provided by `pthread_mutex_lock` and `pthread_mutex_unlock`. All these mappings take effect only when a new symbol `P4_MULTI_THREADED_ENV` is defined. This symbol is defined when we specify a new virtual architecture `SUN_PTHREADS` during compilation[1] . The code segment below illustrates these issues.

[1]For this initial work only, we have restricted ourselves to SunOS 4.1.3

```
/*define symbols for the new virtual
  architecture type               */
#if defined(SUN_PTHREADS)
#ifndef SUN
#define SUN
#endif
#undef P4_MACHINE_TYPE
#define P4_MACHINE_TYPE "SUN"
#define P4_USING_PTHREADS
#define P4_MULTI_THREADED_ENV
#endif

/*map pthread mutex lock calls to the
  native P4 machine dependent locks*/
#if defined(P4_USING_PTHREADS)
#include <pthread.h>

#ifndef LINT
#define MD_lock_t       pthread_mutex_t
#define MD_lock_init(l)
          pthread_mutex_init((l), NULL)
#define MD_lock(l)
                pthread_mutex_lock((l))
#define MD_unlock(l)
              pthread_mutex_unlock((l))

extern  MD_lock_t       p4_malloc_lock;
#endif
#endif P4_USING_PTHREADS

/*map malloc and free calls to the
  thread-safe versions provided by
  pthread library                 */
#ifndef P4_MULTI_THREADED_ENV
#ifndef LINT
#define p4_malloc malloc
#define p4_free free
#define p4_clock MD_clock
#endif
#else
#define p4_malloc pthread_malloc
#define p4_free pthread_free
#define p4_clock MD_clock
#endif
```

As part of the local data structure of each process P4 uses a linked list to queue incoming messages. The process adds and deletes from this message queue.

P4 system routines `queue_p4_message` and `search_p4_queue` were made re-entrant by protecting them with the mapped Pthreads locks to ensure atomic access by threads executing within the same process. The following code segment illustrates the implementation.

```
struct p4_msg *search_p4_queue(req_type,
                                 req_from, deq)
#ifdef P4_MULTI_THREADED_ENV
    p4_lock_init(&p4_local->queue_lock);
    p4_lock(&p4_local->queue_lock);
#endif

. . . . . . .
. . . . . . .
    free_quel(qp);

#ifdef P4_MULTI_THREADED_ENV
    p4_unlock(&p4_local->queue_lock);
#endif
    return (tqp);
    qp = p4_local->queued_messages->
                    first_msg;
```

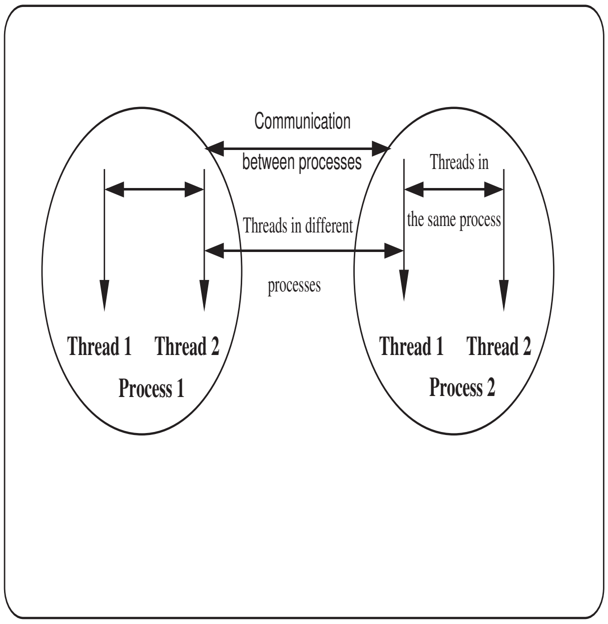

Figure 2: Message passing model

The arrows in figure 2 show message passing at different levels. As indicated in the figure, this model incorporates features to pass messages between threads of the same process (intraprocess), threads executing in different processes (interprocess), and the processes themselves (the usual case). This enhances the amount of concurrency. As is well known messages between processes are exchanged by means of the process id's. Almost all message passing systems provide a query call to the system which returns the id of the calling process. The P4 call that supports this feature is p4_get_my_id(). However it should be noted that threads are not addressable units. To get around this problem the user may assign id's to the threads when the threads are spawned. This is done by assigning numbers to the threads, as they are spawned and specifying this as the argument in the pthread_create() call. The chronology of the calls is as follows:

- main() initializes the P4 environment along with the Pthreads environment (note that Pthreads initialization call is embedded within P4).
- Process 0 reads in the number of threads to be created within each process and passes on the information to the other processes. It is up to the user to decide to choose the process/processes in which he wants to execute the threads.
- All the processes execute the monitor_init() call and the threads register themselves with the monitor. This is a generic barrier for synchronization implemented in terms of Pthreads calls.
- Each thread within a process is assigned a unique id and the threads are spawned by means of the pthread_create call. The name of the worker function and the thread id are arguments to this call.
- The worker function code depends upon the application. Explicit message passing between processes can be accomplished by means of the process id's and that between threads by means of the user assigned thread id's.
- Before quitting the worker function the threads synchronize by calling the monitor_wait() call. And the threads quit by calling the pthread_exit() call.
- The `main` process threads synchronize in the main() routine by means of a monitor_wait() call. Then they execute p4_wait_for_end(). The pthread_exit() is embedded within the p4_wait_for_end().
- It should be noted that if the `main` process threads are not synchronized then one of the threads might terminate destroying the signal environment leading to faulty program environment.

We will forward our suggestions and share our experience with the authors of P4 and discuss permanent inclusion of some the fixes in the future versions of P4.

4.2 MPI

In MPI point-to-point communication the send and receive buffers will have to be thread-safe. In a call such as the MPI_BSEND explicit buffer management becomes crucial. Temporary locking of the buffer during enqueueing and dequeuing should be considered. In MPI collective communication, if two processes are within the same communicator, and each has multiple threads executing within, then the threads cannot participate in a collective operation on the same communicator concurrently.

Alternately, the threads may be involved in different collective operations, see figure 3. Thread A of both the processes are involved in MPI_Reduce, and threads B are involved in MPI_Allreduce. Such a scenario would be possible only if the implementation can provide tags associated with each collective operation. This will ensure that threads would not confuse messages generated on behalf of some other collective operation. Alternately, for global reduce type operations, the reduce operation argument 'op' may be used as a tag with the restriction that threads within the same process may not be involved in the same reduce operation at any given time.

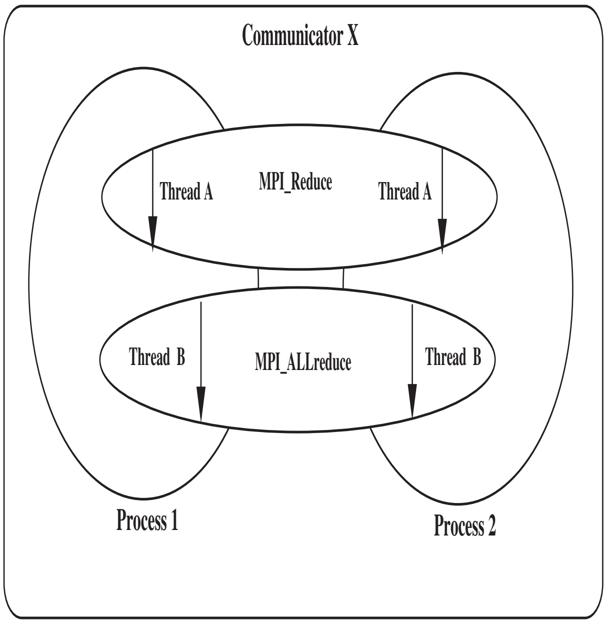

Figure 3: Threads in different processes executing different collective operations concurrently.

Further synchronization primitives can be provided, so that calls like MPI_Comm_dup would not block, see figure 4. This would be transparent to the user. These features can be extended to MPI extensions such as MPIX [14].

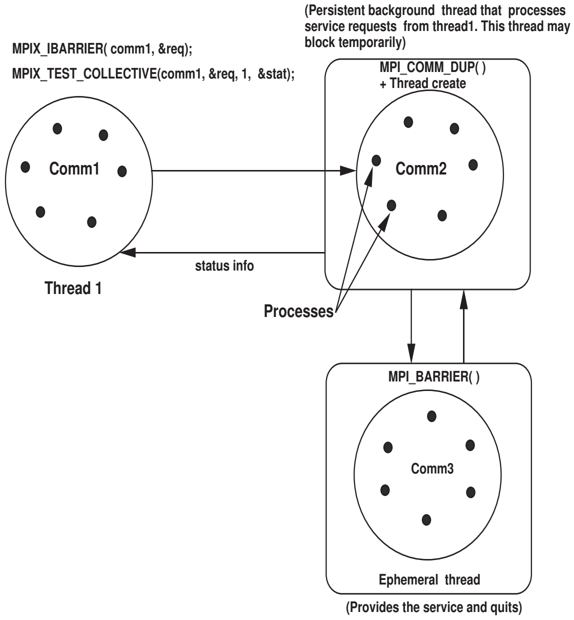

Figure 4: MPIX perspective of thread-safe features of MPI.

5 Conclusions and Future Work

We will make modifications to the current MPICH implementation to incorporate thread-safe features. This new version will be tested to ensure that features of thread-safety enforced at the different levels hold. Further extensions to this model to match with MPIX will be considered.

Multi-threaded message passing plays an important role in exploiting multiple levels of concurrency and computational efficiency. Thread-safety can be guaranteed to support multi-threaded programming.

Acknowledgements

We would like to thank Patrick Bridges for helping us with the Pthreads package. The monitor code is mainly a modified version of the barrier synchronization code that he had written. We also thank Ralph Butler for answering some of the questions concerning the P4 implementation.

References

[1] Joseph Boykin, David Kirschen, Alan Langerman, and Susan LoVerso. *Programming under MACH*. Addison Wesley Publishing Company, 1993.

[2] James Boyle, Ralph Butler, Terrence Disz, Barnette Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processor*. Holt, Rinehart and Winston Inc, 1987.

[3] Ralph Bulter and Ewing Lusk. *A User's Guide to the P4 Parallel Programming System*. Argonne National Laboratory, October 1992.

[4] Greg Burns, Raja Daoud, and James Vaigl. Lam: An open cluster environment for MPI. 1994.

[5] Technical Committee on Operating Systems and Application Environments. *Draft Standard for Information Technology - Portable Operating Systems Interface (POSIX)*. IEEE Computer Society, April 1993.

[6] Nathan Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. A model implementation of MPI. Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993. in preparation.

[7] MPI Forum. *Final Form of Standard - Document for a Standard Message-Passing Interface*. Message Passing Interface Forum, April 1994.

[8] Hubertus Franke, Peter Hochschild, Pratap Pattnaik, and Marc Snir. An efficient implementation of MPI on IBM-SP1. In *Proceedings of the 1994 International Conference on Parallel Processing*, August 1994.

[9] Hubertus Franke, Peter Hochschild, Pratap Pattnaik, and Marc Snir. An efficient implementation of MPI. In *IFIP WG10.3 Working Conference on Programming Environments for Parallel Distributed Systems*, April 1994. also IBM Research Report: RC 19493(84718) 3/25/94.

[10] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[11] T. W. Doeppner Jr. Threads - a system for the support of concurrent programming. Technical report, Brown University, Dept of Computer Science, 1987.

[12] Frank Mueller. A library implementation of posix threads under unix. In *Proceedings of the USENIX Conference*, January 1993.

[13] David Sitsky. Implementation of MPI on the Fujitsu AP1000: Technical details, September 21 1994. Release 1.1.

[14] Anthony Skjellum, Nathan E. Doss, and Kishore Vishwanathan. Inter-communicator Extensions to MPI in the (MPIX MPI eXtension) Library. Submitted to the ICAE Journal, July 1994.

[15] A. Tevanian, R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young. MACHthreads and the UNIXkernel: The battle for control.