ハイパフォーマンス
コンピューティング 62−24
（1 9 9 6. 8. 2 9）

Implementing MPI in a High-Performance, Multithreaded Language MPC++

Francis B. O'Carroll,[†] Atsushi Hori,[††] Hiroshi Tezuka,[††]
Yutaka Ishikawa[††] and Satoshi Matsuoka[†††]

We have ported the MPICH implementation of MPI to the high-performance, multithreaded programming language MPC++. We discuss our modifications to the design of MPICH to support multiple threads. MPICH now runs experimentally on top of MPC++ on a Sun workstation cluster connected by Myrinet and achieves higher performance than standard MPICH on Myrinet TCP/IP on the same hardware.

1. Introduction

MPI[1),2)] is a standard message passing library which, among many design goals, was designed to be very portable and have a thread safe application programmer interface. Portability in a normal single threaded environment has been demonstrated by the several implementations of MPI available for a wide variety of hardware. However, the implications of a thread safe and reliable implementation of MPI are only now being explored. The MPI-2[3)] document notes a few minor places in the original standard's API that are not thread safe and proposes simple fixes. The type of threading environment envisioned for MPI and the semantics of MPI in such an environment are explored informally in[4)] and more formally in[3),5)]. Thread safe, reliable, multithreaded implementations of MPI for particular environments are only just now being developed[5)]. Furthermore, the wide variety of experimental and production threaded environments now available poses a challenge to creating a portable design that can also be portable across different multithreaded environments.

As a first step to a portable multithreaded implementation of MPI we have implemented MPI in the the multithreaded language MPC++ by porting the MPICH version of MPI, and taken advantage of MPC++'s non-preemptive thread model to design a multithreaded MPI more easily than would be possible in other multithreaded environments.

† System Twenty One, Inc
†† Tsukuba Research Center, Real World Computing Partnership
††† Department of Information Engineering, University of Tokyo

2. Overview of MPICH

We chose the MPICH implementation of MPI because it has a clean, layered design for portability to different systems and there was some documentation[4),6),7)] on its software architecture and how to port it. In addition it has some pseudocode support for threads in the source code.

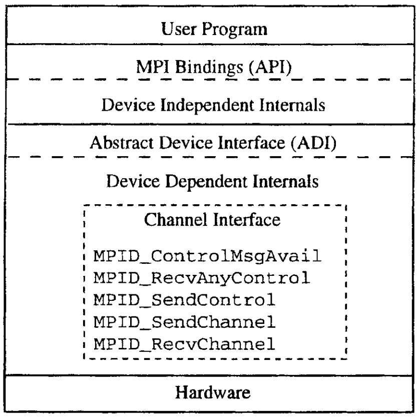

Fig. 1 MPICH software architecture

The software structure of MPICH is illustrated in **Fig. 1**. The user program calls MPI through an application programmer interface, or MPI bindings, which are the C or FORTRAN function entry points (all functions at this level are of the form `MPI_*`). The API calls a device independent layer of internals which implement concepts such as MPI's datatypes, groups, topologies and device independent parts of the communicator concept. This layer communicates with the hardware through an Abstract Device Interface (ADI) which is feature rich abstraction (over 30 functions) of communication device operations modes and protocols. The ADI is implemented with device dependent code. If the underlying hardware provides some of the advanced (high performance) features of the ADI they may be implemented directly; other features of the ADI must be software emulated if the hardware does not support them. These ADI entry points are prefixed `MPID_`.

Each type of hardware needs its own implementation of the ADI, and the highest performance implementations of MPICH on each platform have highly tuned the internals of the ADI. However, MPICH also provides a highly simplified general purpose implementation of the ADI called the *channel device*. The channel device is almost all software emulation and the hardware dependent code has been distilled into the following five functions: `MPID_ControlMsgAvail` is a non-blocking check for any waiting control message. `MPID_RecvAnyControl` is a blocking receipt of any incoming control message. `MPID_SendControl` is a non-blocking send of a short control message. The blocking operations `MPID_SendChannel` and `MPID_RecvChannel` are for large messages and won't be further considered here.

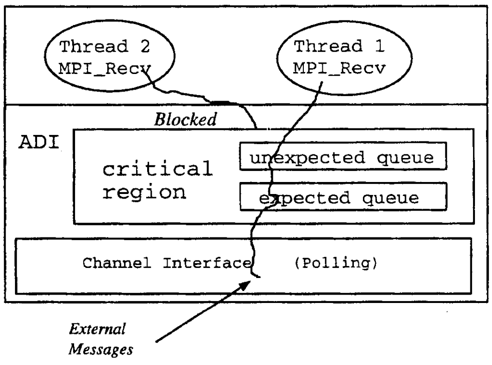

Fig. 2 Operation of MPICH with threads, as envisioned by original pseudocode

The operation of the channel device is shown in **Fig. 2**. The channel device uses polling. From the point of view of a receiving node, messages arrive from other nodes without warning and we have to deliver them to the the matching (source processor number, message tag, and communicator context) calls to an MPI_Recv type function (blocking or non-blocking mode, buffered, synchronized, ready, or part of a collective operation). Messages are regarded as *expected* or *unexpected*. An expected messages is one for which some type of receive has already started executing (on some thread), and an unexpected message is one for which no matching receive has been executed yet on any thread.

The MPICH ADI[6] receives all messages without regard for whether the application is ready for them or not. The ADI is invoked by some form or receive function and then receives any outstanding messages, and either matches them with the expected message handles, or else puts the messages on the unexpected queue. But, for a blocking receive, it will only exit the ADI when the corresponding message arrives.

Since the MPICH ADI is originally written for a single threaded process it uses polling to receive messages. To receive a message, the user calls some form of MPI_Recv. The request is passed down to the device dependent layer which first checks the unexpected queue to see if a matching message has already been received. If so, the message is dequeued and passed back to the user. If not, a message handle recording the request is created and stored on the expected queue. Then the ADI is called which polls for any available messages. As each message is received, it is placed on the expected or unexpected queues as appropriate. Only when a message *matching* the original MPI_Recv is received does the polling stop and the original request is satisfied.

The previous description applies to single-threaded operation. For multiple threads, operations on the queues must be implemented in a critical region. The original MPICH code has some pseudocode to create a critical region around the queue operations. In Fig. 2, thread 1 has locked the queues and is polling for message 1. Thread 2 is blocked trying to access the queues. If thread 1 receives message 2 before message 1, message 2 will be queued on the unexpected queue and thread 2 cannot receive it until message 1 has been received.

3. Interaction of Threads and Communication

MPICH's pseudocode does not indicate what type of threading model it was written for. The term *blocking* for the channel interface functions in the context of other threads, and the operation of the thread scheduler, must be defined.

Considering threading models and their interaction with interprocessor communication, threads can be implemented at the kernel level or the user level. Similarly interprocessor communications can be implemented at kernel level or user level.

Case 1: If the kernel handles both threads and communications, then a thread blocked waiting for communication will only block one thread, but switching to another thread will be expensive. And example of this style is the LWP (light weight process) of SunOS.

Case 2: If threads are implemented at user level but communication is handled by the kernel, then any communication attempt will block all threads. An example of this is POSIX pthreads running in a Unix environment.

Case 3: If both threads and communications are coordinated and done at user level (I/O will have to be handled by other means), then waiting for a communication event will only block a single thread and a thread context switch will occur very cheaply.

Since case 3 applies to our implementation of MPC++ on a workstation cluster, we expect that a multithreaded MPI will achieve high performance.

4. First Implementation

As a first implementation of the channel interface of the MPICH ADI, we need to provide a way to transport messages, a way to synchronize sending and receiving threads.

MPC++ provides an extended syntax[☆] for synchronous and asynchronous local and remote function invocation (creation of threads), and *entry* and *token* types for message passing.

The remote function call was used to transport control messages as function arguments.

☆ Version 2 of MPC++ eliminates language extensions. The same capabilities are provided by the Multi Thread Template Library (MTTL), hence accessible to any C++ program through standard syntax.

We define a class MPID_Sync which can be used for the key operation of synchronizing message exchange between independent threads.

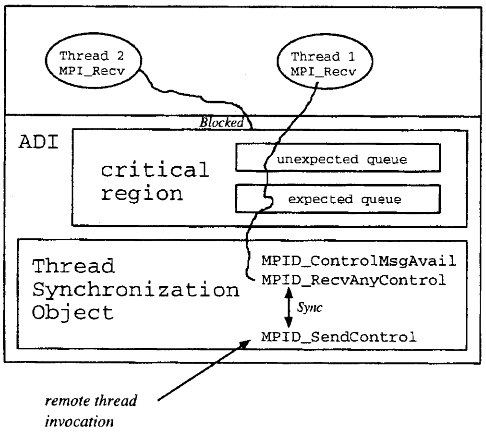

Fig. 3 Operation of MPICH with in MPC++

```
class MPID_Sync{
private:
 int available;
 entry(MPID_PKT_T) ee;

public:
 MPID_Sync(){available = 0;}

 MPID_PKT_T read() {
  MPID_PKT_T vv;
  ee(vv):
    available = 0;
  return vv;
 }

 void write(MPID_PKT_T vv){
  available = 1;
  ee <- [vv];
 }

 int avail() {
  yield();
  return available;
 }
}
```

MPID_Sync defines three member functions, read(), write() and avail(). They correspond to the core of MPID_RecvAnyControl, MPID_SendControl and MPID_ControlMsgAvail.

Suppose that a PE0 wishes to send a message (of type MPID_PKT_T) to PE1. Assume that PE0 knows the location of an MPID_Sync object on PE1. Then PE0 creates a message packet of type MPID_PKT_T and then passes the packet as a function parameter by remotely invoking the write function on PE1's MPID_Sync object. It is invoked asynchronously to satisfy the non-blocking requirements of MPID_SendControl.

Reception of the message works as follows. When PE1 invokes MPID_Sync::read(), it will either return immediately if the message is available, or it will block until the message is received. To avoid blocking the receiving thread, if PE1 wants to do a nonblocking check of message availability on a particular MPID_Sync object, it calls the object's avail() member function.

When avail() is called, there are three possibilities. Either the write() has already been executed, so MPID_Sync::available will be nonzero, or it is possible that there is a remotely invoked thread waiting to execute the write(). However, unless the thread calling avail() yields, such a waiting thread will never have a chance to be executed. Also, even if there are no such waiting threads, there may be other MPI threads which need a chance to run. Hence, avail() first yields, to give messages a chance to arrive. We rely on the scheduler being fair and eventually passing control back to this thread. The code for avail could also be

```
if (!available)
    yield();
return available;
```

Such code would only yield control to another thread if absolutely necessary.

A runtime library remote memory copy (get) was used for bulk data transfer of large messages to implement MPID_RecvChannel and MPID_SendChannel.

5. Performance

We have compared our implementation of the above design with standard MPICH. The code above has been used to implement MPI in MPC++ running on a workstation cluster at RWCP. The workstations are Sun SS20 model 71 equipped with LANai 2.3 interfaces to Myrinet switches, and also 100 Mbps Ethernet, for comparison. Theoretical maximum throughput of Myrinet is 640 Mbps.

In this environment, the MPC++language implementation uses the PM[8] communications library. PM does not use the Myrinet API (nor TCP/IP) and achieves higher bandwidth and lower latency than the Myrient API.

Standard MPICH runs on workstation clusters using Unix sockets, hence it uses TCP/IP for communication. Myrinet supplies a TCP/IP driver for the LANai hardware. We measureed the performance of standard MPICH using Myrinet's TCP/IP driver. **Table 1** shows maximum bandwidth was about 8.7 million bytes per second. Using MPC++ and PM we achieve about 24 million bytes per second. MPC++ makes much better use of the Myrinet bandwidth. Interestingly, MPICH on TCP/IP over Myrinet is only about 35% faster than over 100 Mbps Ethernet.

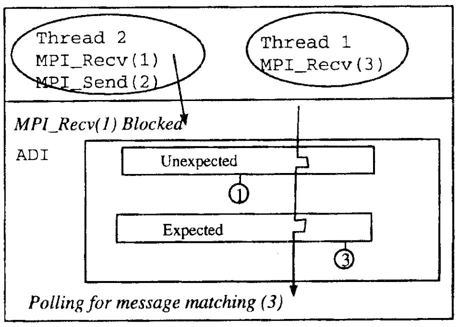

Fig. 4 Deadlock in ADI

6. Deficiencies of First Design

We have already mentioned that MPI blocking calls may block each other even if on different threads since only one thread is in the ADI at any one time, and it polls until it finds

Table 1 Bandwidth (10^6 bytes per second)

Message Size (bytes)	MPC++ Myrinet	TCP/IP Myrinet	TCP/IP 100 Mbps
1	0.005	0.001	0.001
4	0.022	0.005	0.006
16	0.090	0.022	0.025
64	0.353	0.090	0.100
256	0.772	0.353	0.383
1024	2.578	1.198	1.430
4096	6.043	3.025	3.127
16384	15.069	5.454	5.246
65536	22.149	8.148	6.695
262144	24.032	8.688	6.432

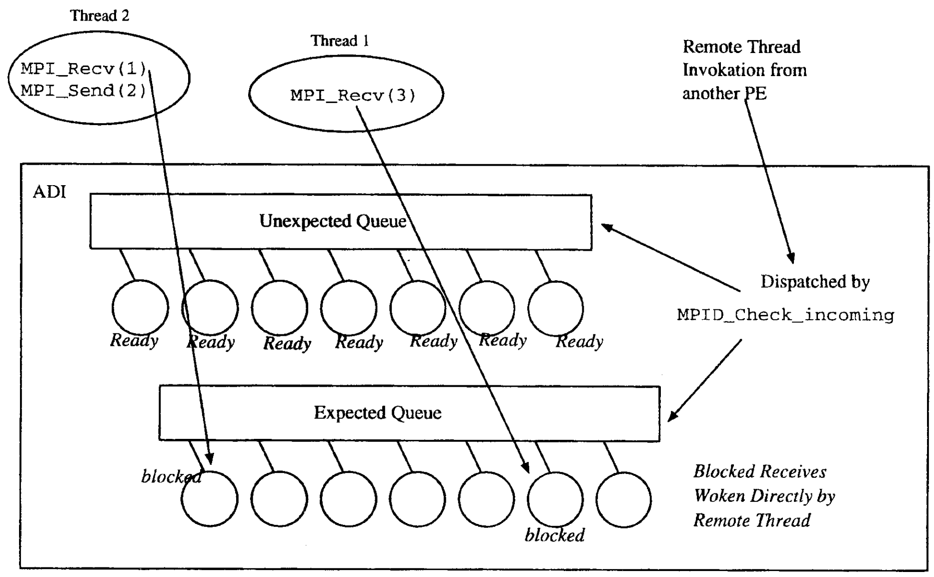

Fig. 5 Better Multithreaded Design

the message matching the original receive which invoked this thread, regardless of any threads waiting to enter the ADI.

Figure 4 illustrates a possible deadlock. Suppose that another PE is executing the sequence `MPI_Send(1); MPI_Recv(2); MPI_Send(3);` Thread 1 is busy polling for a message matching a tag value of 3. It has already received a message with a tag value of 1 from another PE, and placed that on the unexpected list. Thread 2 is waiting for thread 1 to exit the critical section so it can search the queues and retrieve the message with tag 1. The other PE will never send tag 3 to wake up thread 1 because it is waiting on a message from the blocked thread 2.

7. More efficient Multithreaded Design

Figure 5 shows a better design that avoids deadlock. The same example would work as follows. Now, each node on the queue is a synchronization object. Thread 1 executes MPI_Recv(3), locks the queues, does not find a tag of 3, so creates a synchronization object on the expected queue. It then unlocks the queues and calls read() on the new object. Since write() has not been called on the new object, thread 1 blocks, allowing another thread to execute (in the original code, it would keep polling for messages in the ADI). Now thread 2 tries to MPI_Recv(1) and also blocks on a new, different object. It was able to enter the ADI because the queues are only locked for a short duration. Polling has been removed.

Instead of polling, a remote thread from the sending PE will synchronize with and wake up the corresponding receiver. Now the remote PE no longer knows the location of a single synchronization object, so it cannot call write() directly. We implement (and modify) a routine from the ADI called MPID_Check_incoming to act as a dispatcher.

The remote PE asynchronously remotely invokes MPID_Check_incoming, passing the message as an argument. MPID_Check_incoming locks the queues, searches for the synchronization object of a matching receiver. If found it detaches the object from the queue, unlocks the queue and then calls write() on the object. This will unblock the receiving thread (which is blocked on read())and schedule it for execution.

If there is no match on the expected queue, it creates a new object on the unexpected queue and calls write(). When a later receive searches the unexpected queue and finds a match, it will call read() which will not block as the message is already delivered.

This design not only removes deadlock from the previous design, but also may result in lower latency at the MPI user level.

8. Summary and Future Work

We have analyzed the architecture of the MPICH implementation of MPI with respect to a threaded environment and modified the design to support multiple threads in MPC++ running on a workstation cluster. The first design has been implemented and its performance exceeds the standard MPICH on the same hardware. MPC++'s non-preemptive, user level threads and thread-coordinated user level communication library simplified the task of porting and debugging. The next version of MPICH in MPC++ will be pure C++ and use version 2 of MPC++ and its Multi Thread Template Library, and will implement the improved design.

References

1) Message-Passing Interface Forum: MPI: A message passing interface standard, version 1.1 (June 1995).
2) Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W. and Dongarra, J.: *MPI: The Complete Reference*, MIT Press, Cambridge, Massachusetts (1996).
3) Message-Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface (Draft Proposal) (April 1996).
4) Gropp, W., Lusk, E. and Skjellum, A.: *Using MPI*, MIT Press, Cambridge, Massachusetts (1994).
5) Skjellum, A., Protopov, B. and Hebert, S.: A Thread Taxonomy for MPI, *Second MPI Developer's Conference Proceedings*, Notre Dame, Indiana, IEEE Computer Society Press, pp.50–57 (1996).
6) Gropp, W. and Lusk, E.: MPICH Working Note: Creating a new MPICH device using the Channel interface, Technical report, Mathematics and Computer Science Division, Argonne National Laboratory (1995).
7) Gropp, W. and Lusk, E.: MPICH ADI Implementation Reference Manual, Technical report, Mathematics and Computer Science Division, Argonne National Laboratory (1995).
8) Tezuka, H., Hori, A. and Ishikawa, Y.: Design and Implementation of PM: A Communication Libaray for Workstation Cluster, *JSPP'96*, pp. 41–48 (1996). (In Japanese).