A High-Performance MPI Implementation on a Shared-Memory Vector Supercomputer

William Gropp and *Ewing Lusk**
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439

Abstract

In this article we recount the sequence of steps by which MPICH, a high-performance, portable implementation of the Message-Passing Interface (MPI) standard, was ported to the NEC SX-4, a high-performance parallel supercomputer. Each step in the sequence raised issues that are important for shared-memory programming in general and shed light on both MPICH and the SX-4. The result is a low-latency, very high bandwidth implementation of MPI for the NEC SX-4. In the process, MPICH was also improved in several general ways.

1 Introduction

MPI [3, 6] is a portable message-passing library specification. MPICH [4] is a portable MPI implementation in the sense that it can be adapted relatively easily to a new machine, and a high-performance implementation in the sense that MPICH enables such an adaptation to take full advantage of high-performance hardware. The NEC SX-4 [7] is a parallel vector supercomputer with shared memory. The presence of shared memory means that message transfer will be done by `memcpy`, and the vector units enable `memcpy` to be very fast. Since the SX-4 supports System V shared memory, MPICH could be ported immediately to the SX-4 because it has already been ported to the System V shared-memory environment in other contexts. However, realizing the potential peak performance of the SX-4 required studying and eliminating several performance bottlenecks. Some of these were specific to the SX-4, and some apply to other machines as well. This paper describes the process of achieving high performance of MPICH on the SX-4 and the issues this process raised. In particular, in the presence of high bandwidth, the cost of locking for shared memory access becomes critical; but reducing the cost of locking introduces other concerns, such as the precise behavior of the memory system, caching strategies, and instruction ordering in a multiprocessor system.

This paper is organized around the sequence of versions of MPICH that we built for the SX-4. For each one, we describe the issues raised, both for MPICH and for the SX-4,

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

and how they were resolved. For each version, we report latency and bandwidth results on the SX-4.[1] We note that we used a very newly-installed version of the system hardware and software. Our performance results are valid for the DLR machine as configured in May 1996. Upgrades to both the hardware and software are expected. We hope that it is easy to see from the results we present here how such upgrades will affect MPICH performance on the SX-4.

2 Shared Memory, Semaphores, and Locks

Each Unix process has its own separate address space. The programming model targeted by the MPI standard requires separate address spaces for the application's MPI processes. However, many Unix-based systems provide mechanisms by which some memory can be shared among processes, and this memory can be used by the MPI implementation library, even if it is not seen by the user application program. One of the most widely available mechanisms is System V shared memory (identified by the presence of the `shmat` system call). Others include the use of `mmap` to produce a shared-memory area, available on a wide variety of Unix-based systems, and vendor-specific mechanisms such as SGI's shared arenas. The SX-4 uses System V shared memory for sharing memory among Unix processes, although its primary parallel programming model is *pthreads* (see Section 9.2 for more on this model).

In order to coordinate access to shared memory, a locking mechanism is required. System V environments provide such a mechanism through System V *semaphores*, identified by the presence of `semop`. Most modern CPUs also have hardware instructions from which locks can be synthesized, such as a test-and-set instruction or conditional load and store instructions. Locks can even be synthesized entirely in software [5]. Various vendors provide proprietary locks of various kinds in their libraries. The SX-4 software environment provides System V semaphores. The machine has a test-and-set instruction, but access to it is only through assembler language.

3 MPICH Message-handling Protocols

To understand our results, one must have some familiarity with the structure of MPICH and the message protocols it uses to obtain robust, high performance. Details of the MPICH architecture can be found in [4]. The fundamental concept is that of the *abstract device interface*. Almost all of MPICH is completely portable, with the crucial set of environment- and vendor-dependent functions captured in a collection of implementations of the abstract device interface routines. The basic abstract device implements point-to-point message passing; collective operations can also be implemented by the abstract device. At the simplest level, the abstract device provides routines to send information from one processor to another. This information may be a complete MPI message, or information used by the MPICH implementation to send a message.

[1] We gratefully acknowledge NEC for allowing us to use the SX systems at Houston and at the German Aerospace Research Establishment (DLR) at Göttingen, where our experiments were carried out. We also acknowledge Rolf Hempel of the NEC Computation and Communication Research Laboratories at Bonn, Germany.

An MPI message consists of two parts: the *data* that makes up the messages that user wishes to transmit and the *envelope* that contains information about the message, such as its source, length, and tag.

MPICH has three different protocols for transferring messages. These are currently selected based on the length of the message,[2] but they can be modified when MPICH is configured for a given system.

short The message is short enough to be included with the envelope (default is less than 1 Kbyte)

eager The message is short enough to be sent immediately (without waiting for a posted receive), with the assumption that it can be buffered on the receiving side. The upper limit of the lengths that use this protocol is very system-dependent. The default is 128K bytes; this is chosen to optimize performance over economy of memory utilization.

rendezvous The message is long and will not be transferred until the receive is posted, in order to eliminate the necessity of buffering.

In the MPICH abstract device, the envelope and a small amount of data are combined in a *control packet*. Additional control packets are used to coordinate the transmittal of data, for example, in the case where data is not delivered until requested. For example, in one implementation of the rendezvous protocol, a control packet is used by the sender to announce the availability of a message; the destination process sends another control packet when it is ready to receive the message. The data is actually transferred in an additional step (without a control packet).

Each of these protocols is implemented in a different way by the various devices. For the shared-memory device (`ch_shmem`) on which our SX-4 implementation is based, each process maintains a separate queue for receiving control packets from any process (thus there are p separate queues if the size of `MPI_COMM_WORLD` is p). These queues are kept in shared memory and are guarded by locks. Each queue has multiple writers (any process sending to the process that owns the queue) but only a single reader (the owner of the queue). Sending a control packet involves allocating shared memory for the packet, filling in the information, and appending it to the end of the destination process's queue. In the general case, appending the control packet to the end of a shared queue requires a lock to guarantee that only one process changes the queue at a time. The data for eager and rendezvous messages are transferred by copying them into and out of the shared memory.

4 The Instantaneous Port of MPICH to the SX-4

Since MPICH had already been ported to the System V environment, and the building of MPICH uses `configure` to identify the capabilities of the environment it is running in and construct `Makefiles` accordingly, only two commands should have been needed for the first port.

[2] The length may be taken relative to the amount of space being used to hold unreceived eager messages.

```
configure
make
```

In practice, this strategy almost succeeded. A few small problems (with NEC's `include` files, the default implementation of `MPI_Address`, and the need always to link with the Fortran linker) were easily resolved. The resulting version, which used System V shared memory and semaphores, passed the extensive test suite for a complete MPI implementation. We tested its performance using `mpptest`, a sophisticated benchmarking tool provided in the MPICH distribution. The results for the simple "ping-pong" test are shown in Figure 1.

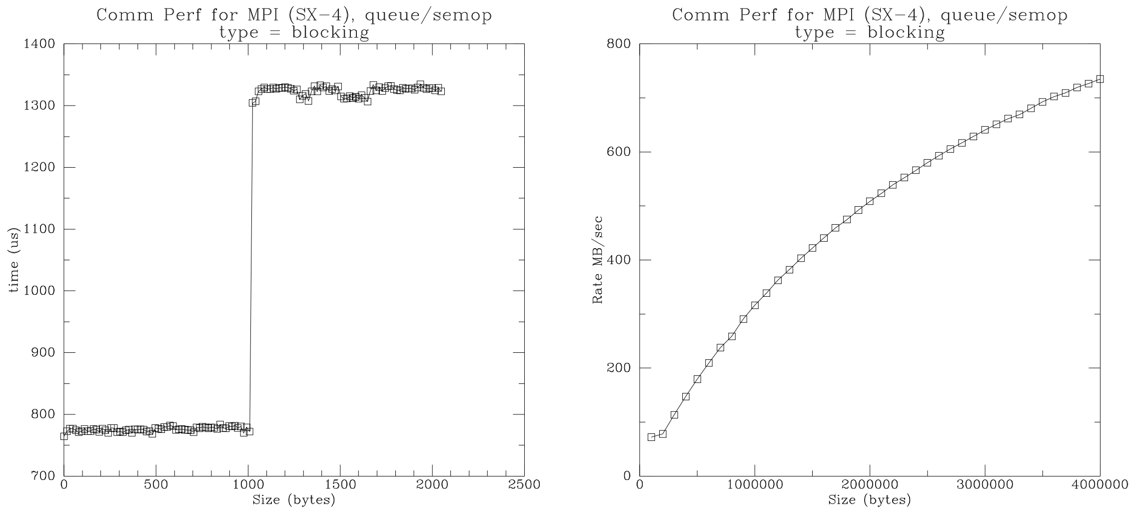


Figure 1: Performance of MPICH with System V shared memory and semaphores

If we consider the latency graph on the left side of Figure 1, two features stand out:

- The latency is high, starting with 750 microseconds and changing to over a millisecond.
- The transition from the short message protocol (message included in packet) to the eager protocol (message short enough to be sent immediately but will not fit in a single packet) at 1024 bytes is quite noticeable, since two locks are required instead of one.

This experiment tells us that the cost of using System V semaphores on the SX-4 is too high to permit low latency. On the other hand, bandwidth is quite promising in the sense that it is still increasing at messages with lengths of 4 megabytes. The SX-4 has special vector move instructions and provides access to the memory bandwidth to users through the standard library version of `memcpy`. The latency cost of the locks is so high, however, that it restricts the bandwidth even for large messages. Therefore we decided to focus on alternative strategies for the use of locks.

5 Lock-free Packet Queues

We followed two approaches: reducing the number of times that we had to perform a lock operation, and replacing the System V locks with less expensive ones. In this section we discuss our experiences with the first approach. In Section 6 we discuss experiences with the second one. Finally, in Section 7 we describe the combination of these two strategies, which yields the best implementation.

We introduced into MPICH a lock-free mechanism for delivering control packets and short messages (short enough to be included in the packet itself). The idea is borrowed from MPICH's T3D abstract device implementation [2], and generalizes to any system with one-sided `get/put` operations, as well as to other shared-memory machines. The central idea is to define in shared memory (System V shared memory on the SX-4) an array of packet slots, one slot for each pair of processes that we expect to communicate. For example, on a 32-processor SX-4, one might make this an array of length 32×32. Optimizations to reduce the size of this array can be made, because in a scalable computation it is unlikely that each process will communicate with each other process; however, we have not made such optimizations yet.

When process i wishes to send a packet to process j, it looks at the packet slot dedicated to messages from i to j. The slot contains a bit indicating whether it is occupied or not. If the slot is unoccupied, process i puts the packet (header, perhaps plus part or all of a message) there and sets the bit. If the slot is occupied, process i chooses one of a number of back-off strategies, and tries again later (perhaps only a microsecond later, or even sooner). Meanwhile, process j can check for an incoming packet by checking the bit. When the bit is set, process j copies the packet out of the slot and clears the bit.

One way to view this is that we have switched from p multiple-writer/single reader queues to $p(p-1)$ single-writer/single-reader slots. By using single writer queues, we avoid the need for a lock. The cost is the need for each process to check $p-1$ slots instead of a single queue. This introduces a scalability issue that we discuss in Section 9.1, along with some variations of this approach.

This algorithm depends on a model of memory and instruction execution (see [1] for an excellent tutorial) that cannot be relied on in today's high-performance machines, which depend heavily on the use of cache and the ability to modify the order of instruction execution (including, perhaps especially, memory reads and writes) for greater performance. Two separate issues are involved:

- Some parallel machines, including the SX-4, are not cache coherent. That is, data written to a memory location by one process does not necessarily invalidate the cache of another processor that may have cached that location. For example, a process may be spinning on a memory location, have it cached, and not see a change made in the value of that location by another process. This is the *cache coherence* problem.

- Many machines, including the SX-4, reorder the order of instruction execution when there is a perceived benefit without a change in semantics. In particular, the order of writes to separate locations might be altered from the order specified in the source code. Thus the assumption that one can write data to one location and then set a bit in another location to indicate that the data is ready to be read is invalid. The

CPU may be allowed to write the bit before writing the data, even if the compiler has been instructed not to reorder instructions at compile time. This is the *sequential consistency* problem.

Both problems can be overcome with assembler-language instructions that force the appropriate type of synchronization. NEC supplied us with C-callable functions to do this. Note that these routines are not coded in assembler language for speed but in order to ensure correctness. There is no mechanism in C for dealing with either the cache coherence or the sequential consistency problem; the `register` declaration of C addresses the issue of *register* consistency. We solve the cache coherency problem by using vector move instructions on the SX-4, which bypass the cache. We solve the sequential consistency problem by using the NEC routines for the critical operations of setting and testing the bit indicating whether a packet slot is full (ready to be read) or not.

The use of these instructions is straightforward. In Figures 2 and 3 we list an abbreviated version of the critical code for manipulating the lock-free queues. Assume that `slots` is the two-dimensional array of packet slots, one for each ordered pair of processes.

```
int ReadControl( pkt, size, from, is_blocking )
MPID_PKT_T **pkt;
int        size, *from;
int        is_blocking;
{
    while (1) {
        for (i=0; i<num_processes; i++) {
            if (PKT_READY_IS_SET(slots[i][myid].ready)) {
               *from = i;
               *pkt  = &slots[i][myid];
               return 0;
            }
        }
        if (!is_blocking)
            return 1;
        else
            /* execute backoff strategy */
    }
    return 0;
}
```

Figure 2: Code for reading a control packet

Macros are used to encapsulate the instructions needed for the critical memory operations. In Figure 2, the `PKT_READY_IS_SET` macro is used to make sure that this routine reads the value of the bit as set by another process, not its own copy in cache.

In Figure 3, the `MPID_PKT_READY_IS_SET` macro is used again to avoid looking at a (stale)

```
int SendControl( pkt, size, dest )
MPID_PKT_T *pkt;
int        size, dest;
{
    if (MPID_PKT_READY_IS_SET(slots[dest][myid].ready)) {
        while (MPID_PKT_READY_IS_SET(slots[dest][myid].ready)) {
             /* execute backoff strategy */
        }
    }
    pkt->ready = 0;
    PKT_COPYIN( &slots[dest][MPID_myid], pkt, size );
    PKT_READY_SET(slots[dest][myid].ready);
    return 0;
}
```

Figure 3: Code for sending a control packet

copy of the ready bit in cache. In addition, the `PKT_COPYIN` and `PKT_READY_SET` are used enforce sequential consistency, i.e., to make sure that the packet data is indeed written before the bit indicating that it is ready is set.

In any particular version of MPICH, the macros are defined to be the instructions needed to perform the operations (setting and testing the "ready" bit and copying a packet into a slot) in a correct way. On the SX-4, these call the assembler language routines given us by NEC, which use special synchronization instructions to provide sequential consistency for the memory operations. On a PA-RISC machine, we would use a `sync` instruction to flush memory writes, and might depend on cache-coherent hardware.

On the SX-4 there is no instruction to flush a single cache line, only to flush the entire cache. Therefore we bypass the cache altogether with the vector move instructions for copying a packet. This ensures that the data will be visible to all processors when the instruction completes on one processor.

The need for these routines also illustrates why locks are often so expensive. A general purpose lock must ensure that the memory satisfies the user's expectations of sequential consistency and cache coherence; this may involve a significant overhead beyond the cost of the lock operation itself. For example, on the SX-4, because of the cost of flushing the cache, a general lock operation will always be relatively expensive.

Performance of the resulting MPICH version is shown in Figure 4. Eliminating locks for short messages reduced the latency to around 40 microseconds. On the other hand, we still need one lock for medium-length messages, and this is still expensive (the lock is used in the allocation of shared memory for the message data; there is a single shared pool of shared memory in the `ch_shmem` implementation). One can also notice the effect of using a large amount of memory for the lock-free queues ($32 \times 32 \times 1024$ bytes). Since only a limited amount of System V shared memory can be defined on the SX-4 (between 8 and 16 megabytes on the system we used), very long messages have to be transferred through

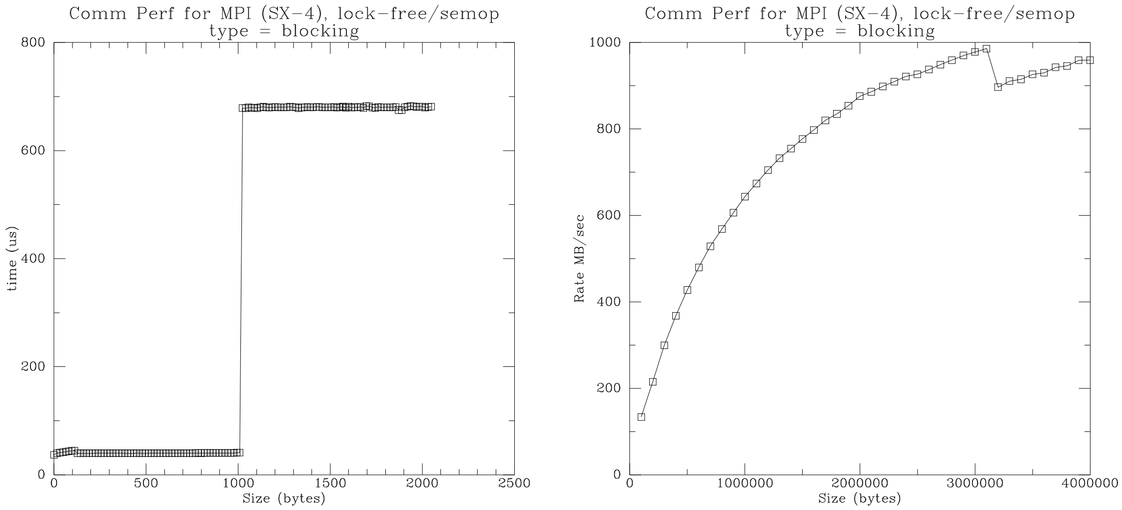
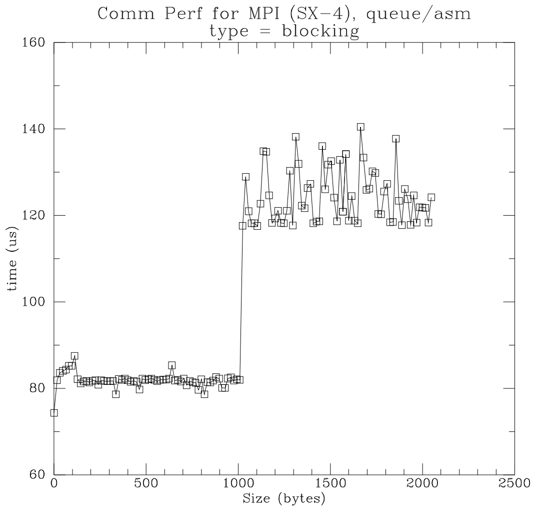


Figure 4: Lock-free Queues plus System V Semaphores

shared memory in multiple segments. The length at which we switch from one segment to two is indicated by the dip in bandwidth at about 3 MB.

6 Replacing System V Semaphores with Fast Locks

The SX-4 instruction set includes a test-and-set instruction, from which a lock can be constructed that does not require a system call. In addition, use of this instruction automatically synchronizes the processor with its memory, so that one need not worry about sequential consistency issues (cache consistency is still an issue, but by using vector instructions to read and write data, we avoid the cache entirely). Therefore the simplest way to improve performance over the "instantaneous port" version is to replace the System V semaphores with such locks. On request, NEC provided us with C-callable assembler-language functions that implement locks using the test-and-set instruction. Replacing the System V semaphores with these locks was not completely straightforward, since MPICH bound together the System V locking scheme (which we wished to replace) with the System V shared-memory scheme (which we wished to keep). A small amount of work on MPICH's abstract device for shared memory (`ch_shmem`), however, made this possible, and MPICH is now the better for it. The results of replacing the System V locks with the test-and-set-based locks is shown in Figure 5. Latency for short messages is about 80 microseconds, and it jumps to about 120 microseconds when the second lock is required. The much lower locking cost improves the bandwidth, which is reaching 1.2 GB/second for large messages.



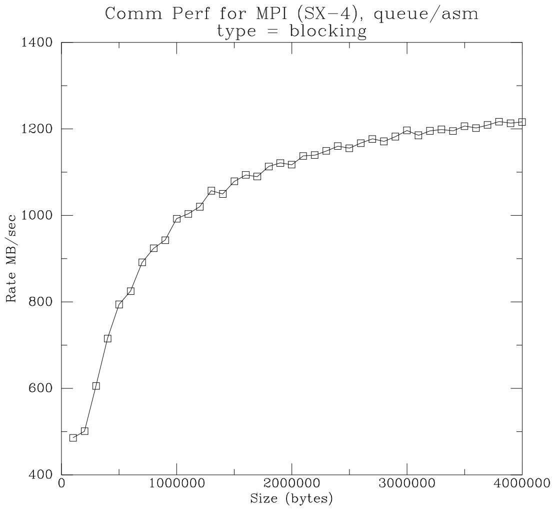


Figure 5: Using assembler-language locks

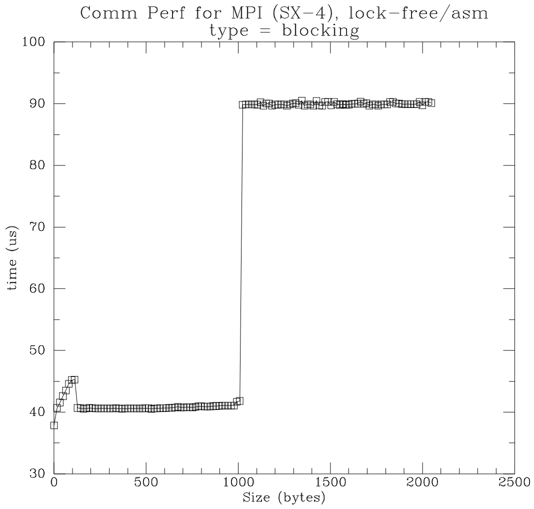
7 Putting It All Together: Lock-free Packet Queues and Fast Locks

The assembler-language locks turned out to be significantly slower than we expected, so the lock-free mechanism for packet delivery is preferred. In Figure 6 we see the result of combining the System V shared memory, lock-free queues, and assembler-language locks where locks are required. Latency is now a respectable 40 microseconds (dipping to 37 microseconds for 0-length messages, and jumps to only 90 microseconds for messages of length 1 kilobyte. Bandwidth is still 1.2 GB/sec., limited only by the speed of `memcpy` (see Section 8) and the amount of shared memory available. This is now the current version of MPICH for the SX-4. It passed all of the acceptance tests that we ran, which included all of the extensive MPICH tests except the collective tests, because they assume the existence of a greater number of processors than we had available.

8 Can We Do Better?

All of the above modifications to MPICH were made during a five-day period while we were visiting the NEC Computation and Communication Research Laboratories at Bonn, Germany. MPICH was proved to be easy to port to this new environment and easy to tune for high performance. An interesting question is, how much of the potential message-passing performance of the SX-4 did we achieve in this short time? If much more effort were to be invested in an MPI implementation for the SX-4, could latency and bandwidth be improved?

Our answer is, Maybe, but not much. First, let us consider the bandwidth. Using



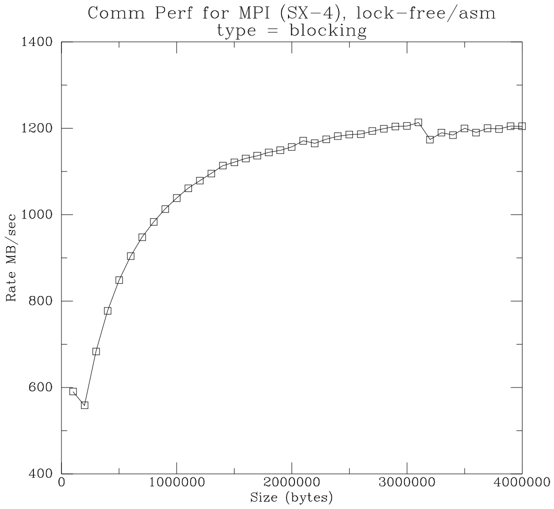


Figure 6: Lock-free queues plus assembler-language locks

separate Unix processes and shared memory, a message must be transferred by two `memcpy` operations, first into and then out of the shared memory. We measured the performance of `memcpy` on the SX-4 using `copytest`, a program distributed with MPICH that we use to measure memory bandwidth *as delivered by* `memcpy`. In this case it is just copying memory within a single process.

As we measured it on the machine we were using (at DLR), `memcpy` delivers about 2.5 GB/second for large moves. Since message passing between processes requires two copies, our 1.2 GB/second bandwidths are quite close to peak. Note that other machine configurations might produce greater bandwidths; we report here just on the experiments we did on the DLR machine. The point is that MPICH does not have much measurable overhead (beyond the cost of `memcpy`). Future hardware upgrades to this machine can be expected to improve the bandwidth of `memcpy` and therefore of MPICH.

Studying latencies caused us to look at the assembler code generated by the compiler for the critical routines that implement the lock-free queues. We did make some changes to the C code that saved a few microseconds, and these optimizations are reflected in the results we have given. It would also be possible to remove a few more microseconds by removing all debugging capabilities. No doubt careful study would reveal a few more corners to be trimmed, and of course the entire calling sequence down from `MPI_Send` could be recoded in assembler, but our examination of the generated code leads us to believe that even major changes, seriously impacting portability, would not take the latency much below 30 microseconds. The issues of cache coherency and sequential consistency in this section of the code (either with or without locks) ensure that in this case the cycle time of the machine is not a good guide to the time needed to execute this code.

9 Further Work

Although the existing implementation of MPICH is complete and efficient as it stands, further work could be done in a number of areas. In this section we describe some directions that the MPI implementation on the SX-4 might take.

9.1 Refining the Lock-Free Queues

The major addition to MPICH that this work created was the lock-free queue mechanism. While what we have done so far is adequate for the benchmarks we ran and demonstrated the utility of the concept, we can envision two related refinements useful for application programs.

In the current data structures, each ordered pair of processes has only one packet slot. If a process is sending multiple messages in rapid succession to another process, it may need to back off repeatedly if the receiving process is not keeping up by posting the appropriate receives. A longer queue for each ordered pair of processes can be created by having an array of packet slots that are used cyclically. Then a sending process need back off only if all the slots for the pair are in use.

Lengthening the queues uses up shared memory, however, which could be a limited resource when the potential number of processes as large, and we have seen (in Figures 4 and 6) that this has a (small) negative effect on the bandwidth. In addition, latency is adversely affected when the number of queues is large, since a receiving process must check all of its queues for incoming messages (the `for` loop in Figure 2). In a truly scalable computation, it will not be the case that every process will communicate with every other process. Therefore, it will have little impact on performance to limit the lock-free queue data structures so that each process has only a relatively small number of queues for incoming messages from the processes that it communicates the most, while messages from other processes are handled by a single queue (per receiving process, as in the `ch_shmem` implementation) guarded by a lock. The processes that "deserve" lock-free queues can be identified dynamically as the communication pattern of the application is recognized.

9.2 Replacing Processes by Threads

The NEC SX-4 system software included a subset of the POSIX *pthreads* library. The primary (far from the only) difference between expressing parallelism with multiple threads in a single process rather than multiple processes is the difference in the memory model. In a *thread* model all static variables are implicitly shared, although it is possible to allocate thread-local memory and access it through special calls in the *pthreads* library. This is in contrast to the Unix *process* model, in which address spaces are implicitly separate, although mechanisms (such as `mmap` or `shmat`) often exist for allocating memory visible to multiple processes. This difference in the memory model typically makes it difficult to port an application program from one model to the other. Hence in this work so far we have considered only the process model as a foundation for MPI implementation.

On the SX-4 it is possible to declare at compile time that static variables are to be kept in thread-local memory, although accessed in the normal way (by `load` and `store` instructions. It is also possible to declare some variables to be shared among threads, using the `_pthread shared_begin`—`_pthread shared_end` compiler pragma. These two features make it possible to replace processes by threads on the SX-4.

Using threads offers two advantages. First, since the process is the unit of scheduling in the operating system, all application threads will be swapped in at the same time. This accomplishes a form of "gang scheduling," which is not supported for groups of processes on the SX-4. Second, since threads can directly access the address space of other processes, the message delivery mechanism we have used with processes, in which messages are copied into shared memory by the sending process and out of it by the receiving process, can be replaced by a single-copy mechanism. The use of a single `memcpy` from one thread-local address to another to complete a receive operation would double the bandwidth of the MPI implementation.

We emphasize that parallel threads (instead of parallel processes) are not visible to the application program, only to the MPI implementation. Therefore message-passing programs, which typically assume that static variables are not shared, should be portable to this scheme. Even so, this is not necessarily a desirable strategy. The bandwidth increase is unlikely to benefit most applications, since the 1.2 GB/second bandwidth of the process model is already so high. It is unlikely that the latency will be affected, since *pthread* locks cannot be much more efficient than the combination of lock-free queues and test-and-set locks that we are already using (and in fact may be slower if they force cache flushes). Finally, since the interaction between threads and system calls has not been POSIX-standardized as much as the *pthreads* interface itself, user programs may not prove as portable as one might hope. (For example, if one thread reads a file, it might block all other threads because the process is blocked, whereas this situation would not occur with separate processes.) Whether these potential drawbacks are outweighed by the benefits of maximum-bandwidth and gang scheduling remains to be determined by implementations of, and experience with, applications.

9.3 Using Lock-Free Queues on Other Architectures

The lock-free queues described in Section 7 were invented to facilitate lock avoidance, not only on shared-memory machines, but also on machines like the Cray T3D and the NEC Cenju-3. These machines, while not supporting true shared memory, do provide the one-sided remote memory access functions `put` and `get`. Using these functions, one process can directly access the address space of another. Therefore, the lock-free queue data structures can be distributed among the private address spaces of the processes. For example, the packet slot (or array of slots) by which process i sends messages to process j can be stored in the address space of process j and accessed via a `put` operation by process i. The one case in which process i would want to do a `get` would be to read the bit that indicates that a slot is empty. For efficiency reasons, this bit should be kept in the address space of process i, which can read it locally, and it should be cleared by process j with a `put` operation when the slot is emptied. One additional change is to keep separate locations for "slot is full" and "slot is empty" to allow purely local memory reads (as opposed to remote memory reads) to detect when a slot contains a message (when checked by a receiver) or when a slot is free to accept a new message (when checked by a sender).

An implementation based on this design has been done for the NEC Cenju-3 by Hubert Ritzdorf of the NEC Computation and Communication Research Laboratories.

10 Summary

We have described the results of porting MPICH to the NEC-SX-4. The modular structure of MPICH enabled a number of distinct implementation strategies to be explored in a short time, particularly since NEC was able to quickly supply special SX-4-specific functions that we needed. The results are summarized in Figure 7. (On the left half of the left side of Figure 7, the two "lock-free" curves coincide, since the code is the exactly same for the short protocol in those two cases.) The "default" version, using standard System V

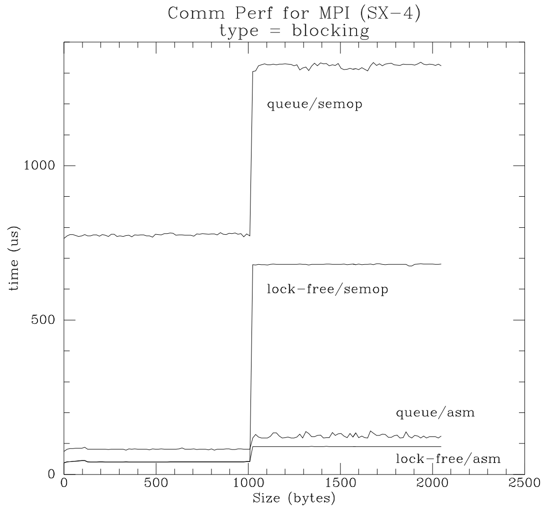


Figure 7: Superimposed Performance Graphs

shared memory and semaphores, while it did provide a complete implementation of MPI on the SX-4 with no additional work, did not have acceptable performance. The semaphores were such expensive system calls that they not only made the latency unacceptable but also significantly impacted the bandwidth (see the lower two curves in the right half of Figure 7). We note that a future release of the operating system is expected to have more efficient System V semaphores. Switching to assembler-level locks pushed the bandwidth close to the maximum available (constrained by the use of two `memcpy`'s). To reduce the latency further, we developed a lock-free queuing mechanism for packets, producing the lowest curve in the left half of Figure 7. The result is a low-latency (38 microseconds), high-bandwidth (1.2 GB/second) complete implementation of MPI on the NEC SX-4.

There were a number of lessons learned that apply to any parallel program. The strategy of replacing general locks with special lock-free data structures points out a way to significantly reduce the cost of coordinating access to shared memory. Of particular interest was the need for assembly language to obtain correct behavior of the memory system;

this suggests the need for language features, much like the `register` and `volatile` of C, to express the memory access relationships.

MPICH, the portable MPI implementation that served as our starting point, gained two general, permanent improvements. First, the lock-free queuing mechanism was encapsulated in a new MPICH "device" (`ch_lfshmem`), which can be implemented on other shared-memory and pseudo-shared-memory machines. Second, the reorganization of the existing `ch_shmem` device, necessitated by our desire to use the assembler-language locks with System V shared memory, will allow greater flexibility in configuring for shared-memory machines in the future.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Technical Report 9512, Rice University ECE, September 1995. (also DEC Western Research Laboratory Research Report 95/7).

[2] Ron Brightwell and Anthony Skjellum. MPICH on the T3D: A case study of high performance message passing. (preprint), 1996.

[3] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report Technical Report No. CS-93-214 (revised), University of Tennessee, April 1994. Available on `netlib`.

[4] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message-passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[5] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987. Also SRC Research Report 7, November 30, 1985.

[6] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[7] N. Nishi, S. Habata, M. Inoue, H. Matsumoto, and T. Kondo. SX-4 architecture for scalable parallel vector processing. In *Proceedings of the International Symposium on Parallel and Distributed Supercomputing*, pages 45–50, September 1995. (Fukuoka, Japan).