# Low-Latency Communication on the IBM RISC System/6000 SP*†

*Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, and Thorsten von Eicken*
Department of Computer Science
Cornell University
Ithaca, NY 14853
{*chichao,grzes,hawblitz,tve*}*@cs.cornell.edu*

**Abstract**

The IBM SP is one of the most powerful commercial MPPs, yet, in spite of its fast processors and high network bandwidth, the SP's communication latency is inferior to older machines such as the TMC CM-5 or Meiko CS-2. This paper investigates the use of Active Messages (AM) communication primitives as an alternative to the standard message passing in order to reduce communication overheads and to offer a good building block for higher layers of software.

The first part of this paper describes an implementation of Active Messages (SP AM) which is layered directly on top of the SP's network adapter (TB2). With comparable bandwidth, SP AM's low overhead yields a round-trip latency that is 40% lower than IBM MPL's. The second part of the paper demonstrates the power of AM as a communication substrate by layering Split-C as well as MPI over it. Split-C benchmarks are used to compare the SP to other MPPs and show that low message overhead and high throughput compensate for SP's high network latency. The MPI implementation is based on the freely available MPICH version and achieves performance equivalent to IBM's MPI-F on the NAS benchmarks.

# 1 Introduction

The IBM RISC System/6000 SP has established itself as one of the most powerful commercial massively parallel processors (MPPs) because of its fast Power2 processors, high-bandwidth interconnection network, and scalability. Nevertheless, the SP's network latency is 2 to 4 times higher than that of older MPPs such as the TMC CM-5 or Meiko CS-2, mainly due to overheads in the communication software and in the network interface architecture.

Active Messages (AM) provide simple communication primitives that are well suited as building blocks for higher layers of software such as parallel languages and complex message

*Formerly known as the IBM Scalable POWERParallel systems 9076 SP2, or simply the IBM SP2
†On-line version at http://www.cs.cornell.edu/Info/Projects/CAM

interfaces. Originally developed for the CM-5 [12], implementations are also available for the Meiko CS-2[10], HP workstations on FDDI ring [9], Intel Paragon, and the U-Net ATM cluster of Sun Sparcstations[13]. All the implementations are based on the Generic Active Message Specification Version 1.1 [3].

Message passing is the most widely used communication model in parallel computing and is now standardized in the Message Passing Interface (MPI)[5]. It supports blocking and nonblocking sends and receives, collective communication, noncontiguous messages, and contains facilities for dealing with groups of processes and libraries. Since much of MPI's functionality is machine-independent, a freely available MPICH [7] implementation of MPI was developed to take care of the upper layers of MPI while providing an abstract device interface (ADI) to the machine dependent layers.

This paper investigates the use of AM communication primitives as an alternative to message passing on the SP in order to reduce communication overheads and deliver high bandwidth with small messages. The first part of this paper (Section 2) describes the IBM SP implementation of AM (SP AM) which is layered directly on top of the SP's network adapter (TB2) and which does not use any IBM software on the Power2 processor. SP AM achieves a one-word message round-trip time of $51\mu s$ which is only $4\mu s$ higher than the raw application-to-application round-trip latency, and 40% lower than the $88\mu s$ measured using IBM's message passing library (MPL). SP AM bulk transfers achieve an asymptotic network bandwidth of 34.3 MBytes/s which is comparable with the 34.6 MBytes/s measured using IBM MPL. Furthermore, SP AM has a message half-power point ($n_{\frac{1}{2}}$) of only 260 bytes using asynchronous bulk-transfers.

The second part of the paper demonstrates the power of AM as a communication substrate by porting Split-C, a split-phase shared-memory extension to C, and implementing MPI over SP AM. Split-C benchmarks are used in Section 3 to compare the SP to other MPPs and show that low message overhead and high message throughput compensate for the SP's high network latency.

The MPI implementation (Section 4) is built entirely over SP AM and is based on MPICH. The goal is to demonstrate that the communications core of MPI can be implemented over AM in a simple fashion and still provide very high performance. This simplicity eases portability and eases optimizations that might otherwise be unwieldy. The implementation focuses on the basic point-to-point communication primitives used by MPICH's ADI and relies on the higher-level MPICH routines for collective communication and non-contiguous sends. Extending the implementation to specialize these functions to use AM more directly would be straight-forward. The current MPI over SP AM matches MPI-F's performance for very small and very large messages and outperforms MPI-F by 10 to 30% for medium size (8 Kbyte to 20 KByte) messages. The NAS benchmarks (Section 6) achieve the same performance using MPICH over SP AM as using MPI-F.

## 1.1 Active Messages background

AM is a low-latency communication mechanism for multiprocessors that emphasizes the overlap of communication and computation [12]. Messages are in the form of requests and matching replies and contain the address of a handler that is invoked on receipt of the message along with up to four words of arguments. The handler pulls the arguments out of the network, integrates them into the ongoing computation and potentially sends a reply back. Large messages are sent using bulk transfer operations called stores and gets which transfer data between blocks of memory specified by the node initiating the transfer.

Message delivery is generally done by explicit network polling in which case each call to *am_request* checks the network and explicit checks can be added using *am_poll*. Interrupt-driven reception is also available but not used in this analysis of SP AM. AM guarantees reliable delivery of messages but does not recover from crash failures, network partitions, and other similar kinds of failures. The interface is summarized in Table 1.

| *Function* | *Operation* |
|---|---|
| `am_request_M` ($M = 1, 2, 3, 4$) | Send an $M$-word request |
| `am_reply_M` ($M = 1, 2, 3, 4$) | Send an $M$-word reply |
| `am_store` | Send a long message – blocking |
| `am_store_async` | Send a long message – non-blocking |
| `am_get` | Fetch data from remote node |
| `am_poll` | Poll the network |

Table 1: Summary of AM interface. *Am_request_M* functions invoke the *handler* on the remote node with $i_1, .., i_M$ as parameters. The handler may reply using similar *am_reply_M* functions. *Am_store* copies *nbytes* of local data to the specified remote address and invokes a handler on the remote node after the transfer is complete. *Am_store* blocks until the source memory region can be reused, while *am_store_async* returns immediately and a separate completion function is called on the sending side at the end of the data transfer. *Am_get* initiates a data transfer from a remote memory block to a local one and invokes a local handler at the end of the transfer.

## 1.2 SP Overview

The IBM SP is an MPP consisting of Power2 RS/6000 nodes interconnected by a custom network fabric as well as by Ethernet. Each node has its own memory, CPU, operating system (AIX), MicroChannel I/O bus, Ethernet adapter, and high performance switch adapter [8]. The SP processing nodes operate at a clock speed of 66MHz and offer a peak performance of 266 Mflops. A model 390 "thin node" contains a 64 KB data cache with 64-byte lines, a 64-bit memory bus, and 64 to 512 Mbytes of main memory. The SPEC ratings are 114.3 SPECint92 and 205.3 SPECfp92. A model 590 "wide node" differs from thin nodes in that it has a 256 Kbyte data cache with 256-byte lines, a larger main memory of up to 2 Gbytes, and a SPEC rating of 121.6 SPECint92 and 259.7 SPECfp92.

The processing nodes are organized in racks of up to 16 thin nodes or 8 wide nodes each and are connected by a high-performance scalable switch. The switch provides four different routes between each pair of nodes, a hardware latency of about 500ns, and a bandwidth close to 40 MBytes/s.

SP nodes are connected to the high-speed interconnection switch via communication adapters (referred-to as "TB2") [11] which contain an Intel i860 microprocessor with 8 MBytes of DRAM. Shown in Figure 1, the adapter plugs into the 32-bit MicroChannel I/O bus with a 80 MB/s peak transfer rate and contains a custom Memory and Switch Management Unit (MSMU) to interface into the network. Data transfers between the MSMU and the MicroChannel are performed using two DMA engines and an intermediate 4KB FIFO. Direct programmed I/O from the host to the adapter RAM is also possible.

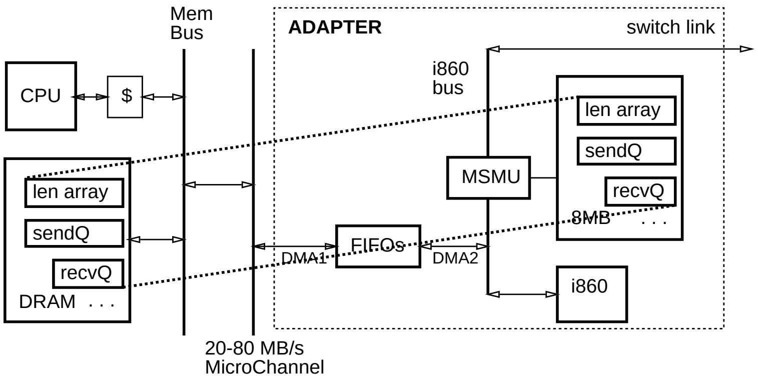


Figure 1: Schematic of the SP network interface

# 2 Active Messages Implementation

This section describes the software interface to the network adapter, the basic mechanisms of sending and receiving a packet, the flow control strategies employed for reliable delivery, and the main performance characteristics of SP AM.

## 2.1 Basic Send and Receive Mechanisms

SP AM relies on the standard TB2 network adapter firmware but does not use any IBM software on the Power2 processor. The adapter firmware allows one user process per node to access a set of memory-mapped send and receive FIFOs directly. These FIFOs allow user-level communication layers to access the network directly without any operating system intervention: the adapter monitors the send FIFO and delivers messages into the receive FIFO, all using DMA for the actual data transfer (Figure 1). The send FIFO has 128 entries while the receive FIFO has 64 entries per active processing node (determined at runtime). Each entry consists of 256 bytes and corresponds to a packet. A packet length array is associated with the send FIFO. Its slots correspond to entries in the send FIFO and indicate the number of bytes to be transferred for each packet. The adapter monitors this length array which is located in adapter memory and transmits a packet when the corresponding slot in the packet length array becomes non-zero.

A packet is sent by placing the data into the next entry of the send FIFO along with a header indicating destination node and route. Since the RS/6000 memory bus does not support cache coherency the relevant cache lines must be flushed out to main memory explicitly. Finally, the transfer size (1 byte) is stored in the packet length array located in adapter memory on the microchannel bus. In bulk transfers this store across the I/O bus can be optimized by writing the lengths of several packets at a time. To receive a packet, the data in the top entry of the receive FIFO is copied out to the user buffers. After being flushed out of the data cache in preparation for a FIFO wrap-around, the entry is popped from the adapter's receive FIFO. This is done lazily (after some fixed number of messages polled) to reduce the number of microchannel accesses (each access costs around 1$\mu$s).

## 2.2 Flow Control

SP AM provides reliable, ordered delivery of messages. The design is optimized for a lossless SP switch behavior given that the switch is highly reliable. Because packets can still be lost due to input buffer overflows, flow control and fast retransmission have proved essential in attaining reliable delivery without harming the performance of the layer.

Sequence numbers are used to keep track of packet losses and a sliding window is used for flow control; unacknowledged messages are saved by the sender for retransmissions. When a message with the wrong sequence number is received, it is dropped and a negative acknowledgement is returned to the sender forcing a retransmission of the missing as well as subsequent packets. Acknowledgements are piggybacked onto requests and replies whenever possible; otherwise explicit acknowledgements are issued when one-quarter of the window remains unacknowledged.

During a bulk transfer, data is divided into *chunks* of 8064 bytes[1]. Packets making up a chunk carry the same sequence number, and the window slides by the number of packets in a chunk; address offsets are used to order packets within a chunk and each chunk requires only one acknowledgment. Figure 2 illustrates the flow-control protocol. The transmission of chunks is pipelined such that chunk N is sent after an ack for chunk N-2 has been received (but typically before an ack for chunk N-1 arrives). The overhead for sending a chunk (175 $\mu$s) is higher than one round-trip which ensures that the pipeline remains filled. Note that with this chunk protocol, there is virtually no distinction between blocking and non-blocking stores for very large transfer sizes.

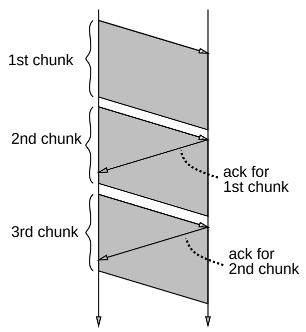


Figure 2: Flow-Control Protocol: initially, two chunks are transmitted and the next chunk is sent only when the previous to last chunk is acknowledged

Although a chunk of 36 packets slightly exceeds half the size of the receive FIFO (64 packets per active node), the sender is unlikely to overflow the receive FIFO in practice.

[1] A packet has 224 bytes of data and 32 bytes of header. A chunk corresponds to 36 packets.

Given the flow control scheme, the window size must be at least twice as large as a chunk (72 packets). To accommodate start-up request messages, the window size is chosen to be 75 packets for requests and 76 for replies.

A *keep-alive* protocol is triggered when messages remain unacknowledged for a long period of time[2]. This protocol forces negative acknowledgements to be issued to the protocol initiator, causing the retransmission of any lost messages.

## 2.3 Round-trip Latency

A simple ping-pong benchmark using *am_request_1* and *am_reply_1* shows a one-word round-trip latency of 51.0$\mu$s on thin nodes. This value increases by about 0.25$\mu$s per word when two, three, or four 32-bit words are transferred. This round trip latency compares well with a raw message (no data or sequence number) ping-pong latency of 46.6-47.0$\mu$s. The additional overhead of 4$\mu$s is due to the cost of the cache flushes and the flow control bookkeeping. The same ping-pong test using MPL's *mpc_send* and *mpc_recv* yields a round-trip latency of 88$\mu$s.

## 2.4 Bandwidth

Several tests are used to measure the asymptotic network bandwidth ($r_\infty$) and the data size at which the transfer rate is half the asymptotic rate ($n_{\frac{1}{2}}$). $r_\infty$ indicates how fast the network adapter moves data from the virtual buffers to the network while $n_{\frac{1}{2}}$ characterizes the performance of bulk transfers for small messages.

The bandwidth benchmarks involve two processing nodes and measure the one-way bandwidth for data sizes varying from 16 bytes to 1 Mbyte[3]. They were run using SP AM bulk transfer primitives as well as IBM MPL send and receive primitives for comparison. The *blocking transfer bandwidth* test measures synchronous transfer requests by issuing blocking requests (*am_store* and *am_get*) and waiting for their completion. For MPL an *mpc_bsend* is followed by 0-byte *mpc_brecv*. The *pipelined asynchronous transfer bandwidth* uses a number of small requests to transfer a large block. This benchmark sends $N$ bytes of data using $\lceil \frac{N}{n} \rceil$ transfers of $n$ bytes, where $N$ is 1 MByte and $n$ varies from 64 bytes to 1 MByte, using *am_store_async* and *mpc_send* respectively.

Figure 3 shows the results. The $r_\infty$ achieved by pipelining *am_store_async* and *am_get* is 34.3 MBytes/s compared to MPL's 34.6 MBytes/s using *mpc_send*. The $n_{\frac{1}{2}}$ value of about 260 bytes for *am_store_async* (slightly higher for *am_get*) compared to about 450 bytes for *mpc_send* indicates that SP AM achieves better performance with small messages.

The bandwidth of SP AM's synchronous stores and gets also converges to 34.3 MBytes/s but at a slower rate due to the round-trip latency as the sender blocks after every transfer waiting for an acknowledgement. Also, for smaller transfer sizes, the performance for gets is slightly lower than for stores because of the overhead of the get request. Consequently, the bandwidth curve for synchronous gets shows an $n_{\frac{1}{2}}$ of 3000 bytes compared to the 2800 bytes for stores. The effect of this overhead on the bandwidth vanishes as the transfer size

[2]Timeouts are emulated by counting the number of unsuccessful polls.

[3]Measurements of the bandwidth on exchange can be found in [2].

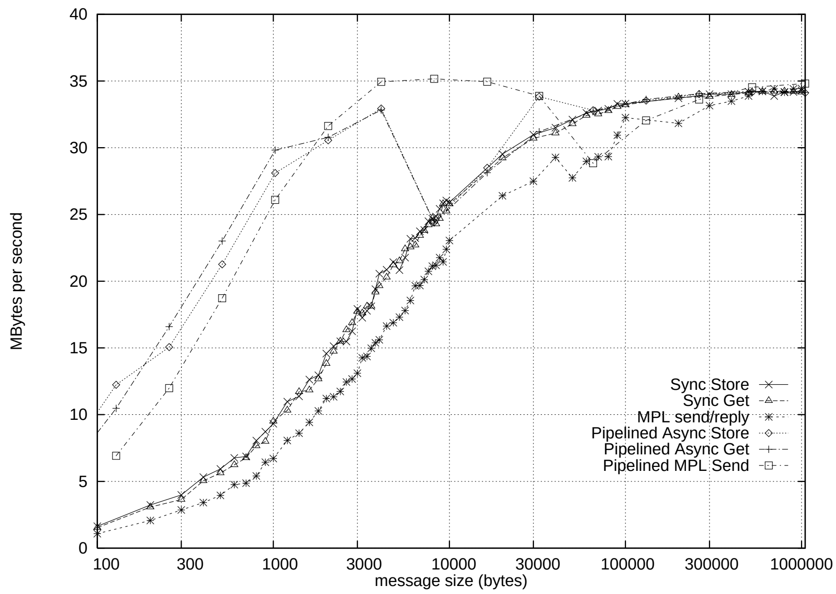


Figure 3: Bandwidth of blocking and non-blocking bulk transfers.

increases, explaining the overlapping of both curves for sizes larger than 4 KBytes. Despite a higher $r_\infty$ of 34.6 MBytes/s, synchronous transfers using MPL's sends and receives have an $n_{\frac{1}{2}}$ greater than 3500 bytes.

Figure 3 clearly shows that SP AM's asynchronous transfers are no better than their blocking counterparts for message sizes larger than one chunk (8064 bytes), which is when the flow control kicks in.

## 2.5 Overheads

Table 2 summarizes the time needed to complete a successful request or reply call. The difference between these two operations is that *am_request_N()* calls *am_poll()* after the message is sent while *am_reply_N()* does not. The time needed to complete *am_poll()* varies with the number and kind of messages received. The overhead of polling an empty network is 1.3$\mu$s. The additional overhead per one received message is about 1.8$\mu$s.

| N | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| am_request_N | 7.7$\mu$s | 7.9$\mu$s | 8.0$\mu$s | 8.2$\mu$s |
| am_reply_N | 4.0$\mu$s | 4.1$\mu$s | 4.3$\mu$s | 4.4$\mu$s |

Table 2: Cost of calling am_request_N() and am_reply_N() functions. (The column for am_request_N() assumes that no messages are received as part of the am_poll which is performed inside am_request.)

| *Performance Metric* | *AM* | *MPL* |
|---|---|---|
| One-Word Round-trip latency | 51.0$\mu$s | 88.0$\mu$s |
| Asymptotic Bandwidth, $r_\infty$ | 34.3MBytes/s | 34.6MBytes/s |
| Half-Power Point (non-blocking), $n_{\frac{1}{2}}$ | 260 bytes | 450 bytes |
| Half-Power Point (blocking), $n_{\frac{1}{2}}$ | 2800 bytes | $\geq$ 3500 bytes |

Table 3: Performance Summary of SP AM and IBM MPL

### 2.6 Summary and Comparison with MPL

MPL's user space round-trip latency is measured with a ping-pong benchmark which sends a one-word message back and forth between two processing nodes using *mpc_bsend* and *mpc_recv*. The asymptotic bandwidth of SP AM is comparable to MPL's asymptotic bandwidth. A half-power point comparison indicates that SP AM delivers better performance than MPL for short messages.

## 3 Split-C Application Benchmarks

Split-C [4] is a simple parallel extension to C for programming distributed memory machines using a global address space abstraction. It is implemented on top of Generic Active Messages and is used here to demonstrate the impact of SP AM on applications written in a parallel language. Split-C has been implemented on the CM-5, Intel Paragon, Meiko CS-2, Cray T3D, a network of Sun Sparcs over U-Net/ATM, as well as on the IBM SP using both SP AM and MPL. A small set of application benchmarks is used here to compare the two SP versions of Split-C with each other and to the CM-5, Meiko CS-2, and U-Net cluster versions. Table 4 compares the machines with respect to one another: the CM-5's processors are slower than the Meiko's and the U-Net cluster's, but its network has lower overheads and latencies. The CS-2 and the U-Net cluster have very similar characteristics. The SP has the fastest CPU, a network bandwidth comparable to the CS-2, but a relatively high network latency.

| *Machine* | *CPU speed* | *Msg Overhead* | *Round-trip Latency* | *Bandwidth* |
|---|---|---|---|---|
| TMC CM-5 | 33 MHz Sparc-2 | 3$\mu$s | 12$\mu$s | 10MBytes/s |
| Meiko CS-2 | 40 MHz Sparc-20 | 11$\mu$s | 25$\mu$s | 39MBytes/s |
| U-Net ATM | 50/60 MHz Sparc-20 | 3$\mu$s | 66$\mu$s | 14MBytes/s |
| IBM SP | 66MHz RS6000-590 | 4$\mu$s | 51$\mu$s | 34MBytes/s |

Table 4: Comparison of the TMC CM-5, Meiko CS-2, U-Net ATM cluster, and IBM SP performance characteristics

The Split-C benchmark set used here consists of three programs: a blocked matrix multiply, a sample sort optimized for small messages, the same sort optimized to use bulk transfers, and two radix sorts optimized for small and large transfers. All the benchmarks have been instrumented to account for the time spent in local computation phases and in communication phases separately such that the time spent in each can be related to the processor and network performance of the machines. The absolute execution times for runs on eight processor are shown in Table 5. Execution times normalized to the SP AM are shown in Figure 4. Detailed explanation of the benchmarks can be found in [2].

| *Benchmark* | *IBM SP AM* | *IBM SP MPL* | *TMC CM-5* | *Meiko CS-2* | *SS20/U-Net/ATM* |
|---|---|---|---|---|---|
| mm 128x128 | 1.094s | 1.180s | 4.606s | 2.516s | 4.470s |
| mm 16x16 | 0.229s | 0.489s | 0.970s | 0.371s | 0.415s |
| smpsort sm 512K | 4.393s | 18.570s | 10.448s | 9.845s | 15.730s |
| smpsort lg 512K | 1.814s | 1.811s | 8.612s | 7.432s | 2.792s |
| rdxsort sm 512K | 9.894s | 54.652s | 27.106s | 21.255s | 81.344s |
| rdxsort lg 512K | 3.543s | 3.587s | 20.011s | 7.995s | 6.126s |

Table 5: Absolute Execution Times (seconds)

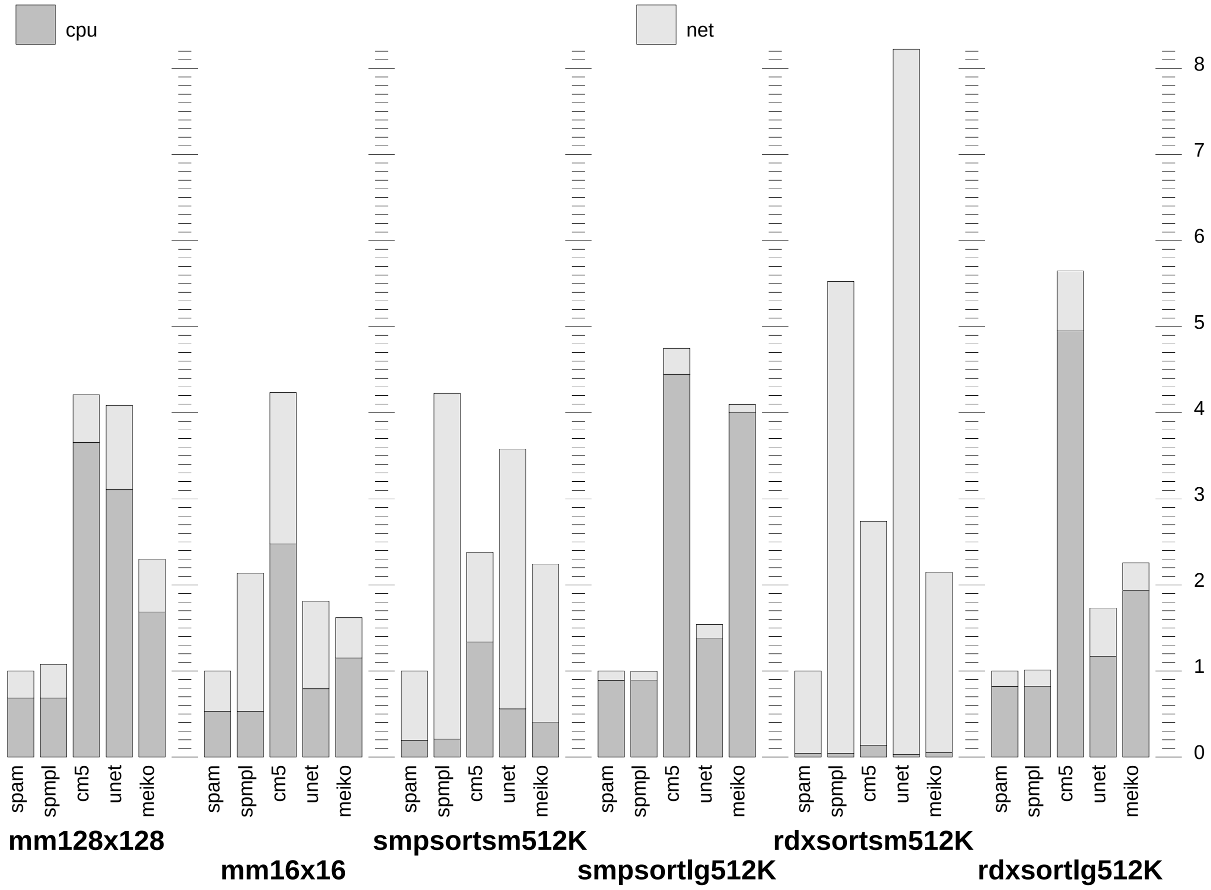


Figure 4: Split-C benchmark results normalized to SP.

The two matrix multiply runs use matrices of 4 by 4 blocks with 128 by 128 double floats per block, respectively 16 by 16 blocks with 16 by 16 double floats each. For large blocks, the performance of Split-C over SP AM and MPL is the same which can be explained by the comparable bandwidth in large block transfers. The floating-point performance of Power2 give the SP an additional edge over the CM-5, CS-2, and the U-Net/ATM cluster. For smaller blocks, however, the performance over MPL degrades significantly with respect to

SP AM because of higher message overheads. Notice that the results over SP AM exhibit a smaller network time compared to all other machines. As long as the transfer sizes remain below 8064 bytes, flow control is not activated and thus overhead matters more than latency.

For radix and sample sorts, Figure 4 shows that the SP spends less time in local computation phases because of the faster CPU. SP AM spends about the same amount of time, if not less, in the communication phases as the CM-5 and CS-2. Although SP's round-trip latency is relatively higher, SP AM combines low message overhead with high network bandwidth to achieve a higher message throughput. Again, the performance over MPL for small messages suffers from the high message overhead. For large messages (albeit not large enough to activate flow control), both the SP AM and MPL outperform the other machines in both computation and communication phases.

The Split-C benchmark results show that SP AM's low message overhead and high throughput compensates for SP's high network latency. The software overhead in MPL degrades the communication performance of fine-grain applications, allowing machines with slower processors (CM-5) or even higher network latencies (U-Net/ATM cluster) to outperform the SP.

# 4 MPI Implementation over Active Messages

Implementations of MPI currently fall into two broad categories: those implemented "from scratch" and tuned to the platform, and those built using the portable public-domain MPICH package. MPICH contains a vast set of machine independent functions as well as a machine specific "abstract device interface" (ADI). The main design goal of MPICH was portability while performance considerations took a second rank. For this reason MPI implementations built "from scratch" are expected to outperform MPICH based ones.

A close look at MPICH's ADI, however, reveals that the basic communication primitives can be implemented rather efficiently using AM and that, with a few optimizations to the ADI and the higher layers of MPICH, the package should yield excellent performance. This section describes such an implementation and presents a number of micro-benchmarks which demonstrate that MPICH layered over AM (MPI-AM) indeed achieves point-to-point performance competitive with or better than IBM's "from scratch" MPI-F implementation.

## 4.1 Basic Implementation

The major difficulty in layering MPI's basic send (*MPI_Send* or *MPI_Isend*) over AM lies in resolving the naming of the receive buffer: *am_store* requires that the sender specify the address of the receive buffer while message passing in general lets the receiver supply that address. This discrepancy can be resolved either by using a buffered protocol, where the message is stored into some temporary buffer at the receiver and then copied, or by using a rendez-vous protocol, where the receiver sends the receive buffer address to the sender which then stores directly from the send buffer into the receive buffer (Figure 6).

For small messages, the buffered protocol is most appropriate because the extra copy cost is insignificant. Each receiver holds one buffer (currently 16 Kbytes) for every other process in the system. To send a message, the sender allocates space within its buffer at the receiver (this allocation is done entirely at the sender side and involves no communication) and performs an *am_store* into that buffer. After the receiver has copied the message into the user's receive buffer, it sends a reply to free up the temporary buffer space.

The buffered protocol's requirements are well matched to *am_store*: the store transfers the data and invokes a handler at the receiving end which can update the MPICH data structures and send a small reply message back using *am_reply*. If the store handler finds that the receive has been posted it can copy the message and use the reply message to free the buffer space. If a matching receive has not been posted, the message's arrival is simply recorded in an "unexpected messages" list and an empty reply is sent back (it is actually used for flow-control by the underlying AM implementation). The buffer space is only freed when a matching receive is eventually posted.

For large messages the copy overhead and the size of the preallocated buffer become prohibitive and a rendez-vous protocol is more efficient. The sender first issues a "request for address" message to the receiver. When the application posts a matching receive, a reply containing the receive buffer address is sent back. The sender can then use a store to transfer the message. This protocol may lead to deadlock when using *MPI_Send* and *MPI_Recv* because the sender blocks while waiting for the receive buffer address. This is inherent in the message passing primitives and MPI offers nonblocking alternatives (*MPI_Isend* and *MPI_Irecv*).

In the implementation of the rendez-vous protocol *MPI_Send* or *MPI_Isend* causes a request to be sent to the receiving node. If a matching receive (*MPI_Recv* or *MPI_Irecv*) has been posted, the handler replies with the receive buffer address; otherwise the request is placed in the "unexpected messages" list and the receive buffer address is sent when the receive is eventually posted (see Figure 5). At the sender side, the handler for the "receive buffer address" message is not allowed to do the actual data transfer due to the restrictions placed on handlers by AM. Instead, it places the information in a list, and the store is performed by the blocked *MPI_Send* or, for nonblocking *MPI_Isend*s by any MPI communication function that explicitly polls the network.

MPI specifies that messages be delivered in order, and the current implementation assumes that messages from one processor to another are delivered in order. Although this is not guaranteed by the Generic Active Messages standard, SP AM does provide ordered delivery. On AM platforms without ordered delivery, a sequence number would have to be added to each store and request message to ensure ordering.

The current MPI-AM uses the polling version of SP AM. To ensure timely dispatch of handlers for incoming messages *am_poll* is called explicitly in all MPI communication functions which would not otherwise service the network. For applications which have long gaps between calls to MPI functions, a timer may be used to periodically poll for messages, although this has not been tested yet.

## 4.2 Optimizations

Profiling of the basic buffered and rendez-vous protocols uncovered inefficiencies that lead to a number of simple optimizations. The first-fit allocation of receive buffers in the buffered protocol turned out to be a major cost in sending small messages. The optimized implementation uses a binned allocator for small messages (currently 8 1K bins) and reverts to the

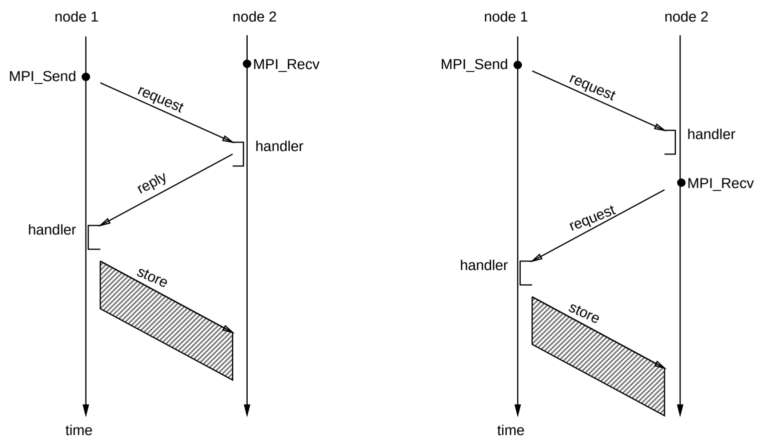


Figure 5: Rendez-vous protocol over AM, when MPI_Recv is posted before MPI_Send (left) and after MPI_Send (right)

first-fit algorithm only for "intermediate" messages. Using a message for freeing the small buffers was another source of overhead and combining several "free buffer" replies into a single message speeds up the execution of the receiver's store handler. These two optimizations, along with some slight code reorganization to cut down on function calls improved the small message latency to within a microsecond of MPI-F.

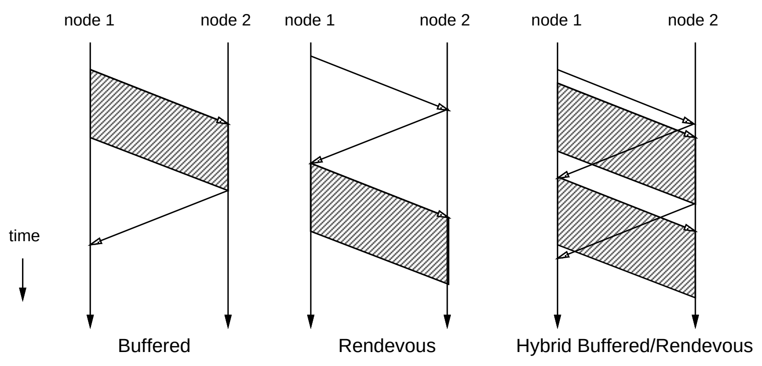


Figure 6: Diagram of Buffered and Rendez-Vous Protocols

Using two distinct strategies for small and large messages means that the implementation has to switch from one to the other at some intermediate message length. This often causes discontinuities in the performance as is the case in MPI-F where the bandwidth achieved using messages of 5 Kbytes is actually lower than with 4 Kbyte messages because of the rendez-vous latency introduced for the larger messages. The optimized MPI-AM augments the rendez-vous protocol by sending out a small prefix (4 Kbytes) of the message into a temporary buffer at the receiver while waiting for the rendez-vous reply. This hybrid buffered/rendez-vous protocol keeps the pipeline full while avoiding excessive buffer space requirements. (If no buffer space can be allocated the hybrid protocol simply reverts to a regular rendez-vous protocol.) By using the hybrid protocol for all messages longer than 8 Kbytes a performance discontinuity is avoided, and the hybrid protocol can reach a higher bandwidth than either the buffered or rendevous protocols could alone (Figure 7).

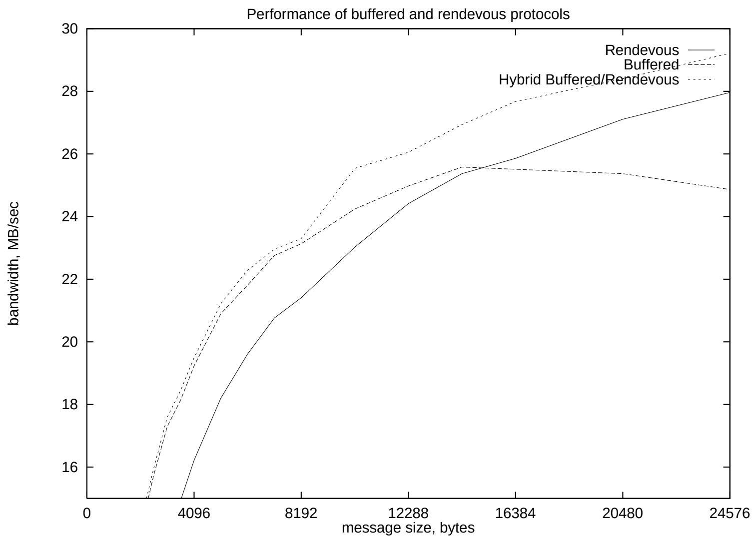


Figure 7: Performance of Buffered and Rendez-Vous Protocols

## 4.3 MPI Point-to-Point Performance

Point-to-point measurements were made by sending messages around a ring of 4 nodes using *MPI_Send* and *MPI_Recv*. All latencies shown are the time per hop (the time around the ring divided by 4).

The per hop latency is shown for thin nodes in Figure 8 and for wide nodes in Figure 10. On the thin nodes MPI over AM achieves a lower small-message latency than MPI-F while on wide nodes MPI-F is faster for messages of less than 100 bytes but slower for larger messages. The communication bandwidths using thin and wide nodes are shown in Figures 9 and 11, respectively. Evidently MPI-F was optimized for the wide nodes while MPI-AM was developed on thin ones.[4] The unoptimized version of MPI-AM shows no performance hit when switching from a buffered protocol to a rendez-vous protocol because the switch occurs at 16K byte messages where the copy overhead of the buffered protocol is already significant. The optimized version switches over at 8K, but shows no performance hit because of the hybrid buffered/rendez-vous protocol.

[4]The MPI-F results on wide nodes presented here differ somewhat from those in [6] in that the switch from a buffered to a rendez-vous protocol occurs at a message size of 4K bytes rather than the 8K bytes. The MPI-F library we used may have been configured differently than IBM's version.

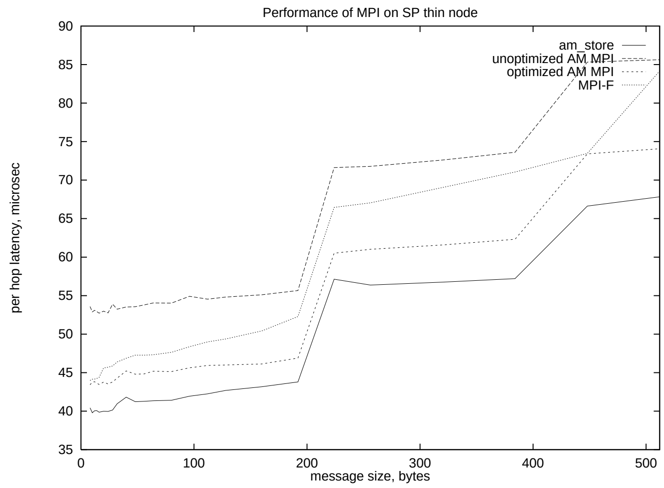


Figure 8: MPI Point to Point Latencies on Thin SP Nodes

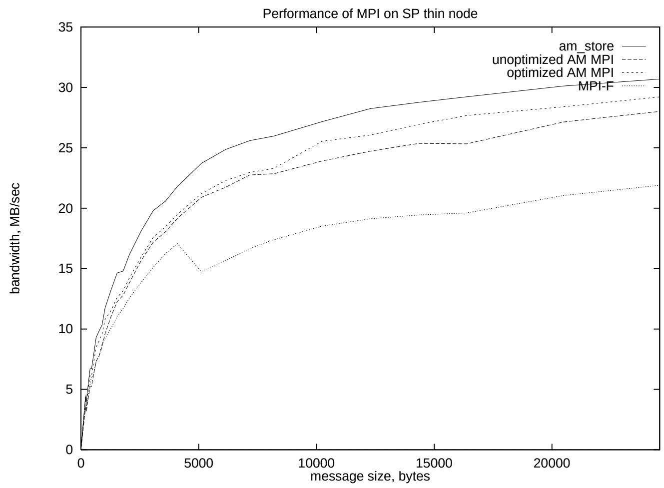


Figure 9: MPI Point to Point Bandwidths on Thin SP Nodes

## 4.4 NAS benchmarks

The NAS Parallel Benchmarks (version 2.0)[1] consist of five numerical benchmarks written in MPI and are used here to compare MPI over AM and MPI-F in a realistic setting. The running time of each benchmark on 16 thin SP nodes is shown in Table 6.

The running times of MPI-AM are close to those achieved by the native MPI-F implementation. The differences shown are due in part to the use of MPICH's generic collective communication routines which are not tuned for the SP. In particular, the all-to-all communication function used by the FT benchmark (*MPI_Alltoall*) caused unnecessary bottlenecks because all processors try to send to the same processor at the same time, rather than spreading out the communication pattern. Streamlining nonblocking communication routines and

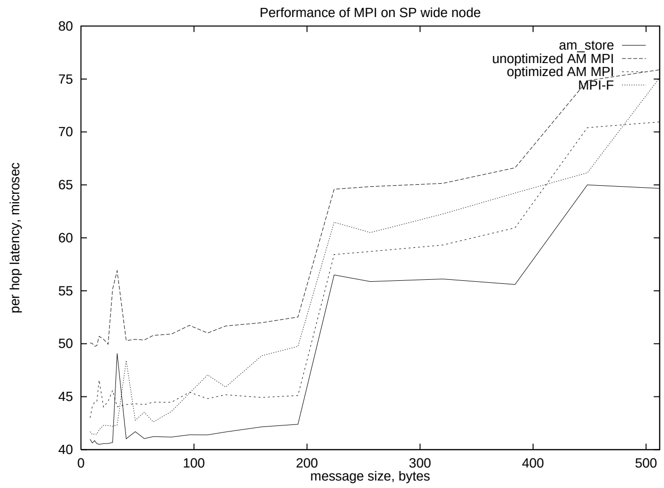


Figure 10: MPI Point to Point Latencies on Wide SP Nodes

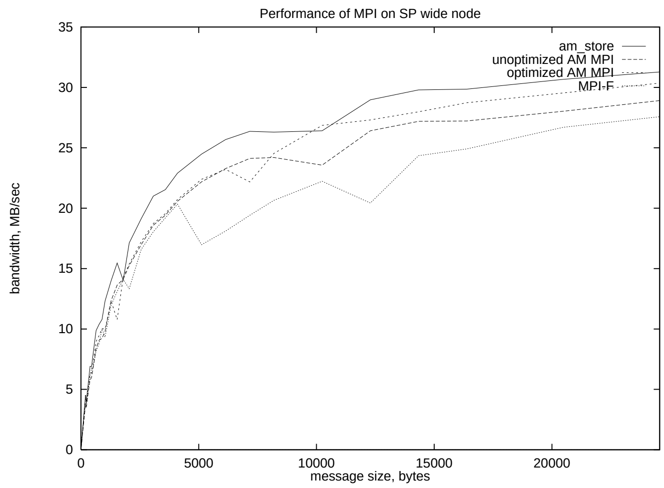


Figure 11: MPI Point to Point Bandwidths on Wide SP Nodes

implementing collective communication functions directly over AM (rather than using the default MPICH functions built over MPI sends) would improve performance.

| *Benchmark* | *MPI-F* | *MPI-AM* |
|---|---|---|
| BT | 347.84s | 359.16s |
| FT | 31.87s | 35.49s |
| LU | 216.56s | 220.92s |
| MG | 7.95s | 8.19s |
| SP | 240.37s | 249.08s |

Table 6: Run-times for NAS Class A Benchmarks on 16 Thin SP Nodes

## 5 Conclusions

Communication on the IBM SP has suffered from the high network latency exacerbated by the software overhead incurred by message passing layers such as MPL. SP AM is a very efficient implementation that reduces communication overhead, delivers high small message throughput, and achieves high bandwidth for bulk data transfers. Split-C benchmarks show that low message overhead and high throughput alleviate the high network latency caused by limitations of the SP network interface architecture. The Split-C and MPI over SP AM results demonstrate that AM is a good substrate for building parallel programming models such as Split-C and MPI without compromising performance. The AM version of Split-C outperforms the MPL version and the MPI NAS benchmarks show that the performance of MPI-AM is comparable to that of MPI-F.

## 6 Acknowledgements

We are grateful to Jamshed Mirza (IBM Poughkeepsie) and M. T. Raghunath (formerly with IBM Kingston) for their invaluable help, and the Cornell Theory Center (CTC) for providing us computing time on the IBM SP during the initial stages of the project. We appreciate the help offered by Beng-Hong Lim and Hubertus Franke (IBM Watson), and Gautam Shah (IBM Poughkeepsie) for dealing with interrupt-driven message handling and updating the SP AM release for AIX 4.1. We thank David Bader (U. of Maryland) for kindly making his Split-C port over MPL available to us. This project has been sponsored by IBM under the joint project agreement 11-2691-A and by NFS under contracts #CDA-9024600-SUB-E66-8381 and #ASC-8902827-SUB-U10-8301. Chi-Chao Chang is partially supported by a doctoral fellowship (200812/94-7) from the Brazilian Research Council (CNPq/Brazil). Chris Hawblitzel is supported by a NSF Graduate Fellowship.

## References

[1] D. Bailey, T. Harris, W. Sahpir, and R. van der Wijngaart, **The NAS Parallel Benchmarks 2.0, Report NAS-95-020**, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, December 1995.

[2] Chi-Chao Chang, Grzegorz Czajkowski, and Thorsten von Eicken, **Design and Performance of Active Messages on the IBM SP2**, Cornell CS Technical Report 96-1572, February 1996.

[3] D. E. Culler, K. Keeton, L. T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, and K. Wright, **Generic Active Messages Interface Specification**, UC Berkeley, November 1994.

[4] D. E. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumeta, T. von Eicken, and K. Yelick, **Parallel Programming in Split-C**, In Proceedings of Supercomputing '93.

[5] Message Passing Interface Forum, **The MPI Message Passing Interface Standard**, Technical Report, U. of Tennessee, Knoxville TN, April 1994.

[6] H. Franke, C. E. Wu, M. Riviere, P. Pattnaik, and M. Snir, **MPI Programming Environment for IBM SP1/SP2**, In Proceedings of the 22nd Int'l Symposium on Computer Architecture, 1995.

[7] W. Gropp and E. Lusk, **MPICH ADI Implementation Reference Manual (Draft)**, August 1995.

[8] IBM, **SP2 Command and Technical Reference**, December 1994.

[9] R. P. Martin, **HPAM: An Active Message Layer for a Network of Workstations**, In Proceedings of Hot Interconnects II, Palo Alto CA, August 1994.

[10] K. E. Schauser and C. J. Scheiman, **Experience with Active Messages on the Meiko CS-2**, In Proceedings of the 9th Int'l Parallel Processing Symposium, Santa Barbara, CA, April 1995.

[11] C. B. Stunkel et al., **The SP2 Communication Subsystem**, August 1994.

[12] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, **Active Messages: A Mechanism for Integrated Communication and Computation**, In Proceedings of the 19th Int'l Symposium on Computer Architecture, Gold Coast, Australia, May 1992.

[13] T. von Eicken, A. Basu, V. Buch, and W. Vogels, **U-Net: A User-Level Network Interface for Parallel and Distributed Computing**, In Proceedings of the 15th Symposium on Operating System Principles, Cooper Mountain, CO, December 1995.