# Design and Performance of Active Messages on the IBM SP-2

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

February 23, 1996

**Abstract**

This technical report describes the design, implementation, and evaluation of Active Messages on the IBM SP-2. The implementation benchmarked here uses the standard TB2 network adapter firmware but does not use any IBM software on the Power2 processor. We assume familiarity with the concepts underlying Active Messages. The main performance characteristics are a one-word message round-trip time of 51.0 $\mu$s and an asymptotic network bandwidth of 34.3 MB/s [1]. After presenting selected implementation details, the paper focuses on detailed performance analysis, including a comparison with IBM's Message Passing Layer (MPL) and Split-C benchmarks.

## 1 Introduction

The IBM SP-2 supercomputer is a massively parallel processor (MPP) consisting of Power2 nodes interconnected by a custom network fabric as well as by Ethernet. Each node has its own memory, CPU, operating system (AIX), microchannel bus, Ethernet adapter, and high performance switch adapter [4]. There are two types of nodes: a *thin* SP-2 processing node is roughly equivalent to the RS/6000 model 390 (SPECint92: 114.3, SPECfp92: 205.3), and a *wide* node is equivalent to the RS/6000 model 590 (SPECint92: 121.6, SPECfp92: 259.7) [4]. The high performance switch provides bandwidth close to 40 MB/s and a round trip latency over "bare-bones" hardware of about 46 $\mu$s.

Active Messages is a low-latency communication mechanism that minimizes overheads and allows communication and computation to be overlapped [9] in multiprocessors. It is a very attractive alternative to the existing IBM Message Passing Layer (MPL) because of the low overhead in setting up messages and reliable transmission. Originally developed for the Thinking Machines CM-5, implementations are also available for the Meiko CS-2, HP workstations on FDDI ring [6], Intel Paragon, and the U-Net ATM cluster of Sun Sparcs.[1]. All the implementations are based on the Generic Active Message Specification Version 1.1 [3].

This technical report describes the design, implementation, and evaluation of Active Messages on the IBM SP-2. The implementation benchmarked here uses the standard network adapter (a.k.a. TB2) firmware but does not use any IBM software on the Power2 processor. The main performance characteristics are a one-word message round-trip time of 51.0 $\mu$s and an asymptotic network bandwidth of 34.3 MB/s for bulk transfers.

After a brief description of the implementation, the various latencies, bandwidths, and overheads measured are presented. Then, we compare the performance of a split-phase shared-memory extension to C based on Active Messages, Split-C [2], on the SP-2 with the U-Net ATM cluster as well as the CM-5 and Meiko CS-2. We also compare our measurements with IBM's Message Passing Layer (MPL).

[1] In this report, we use the somewhat incorrect definition of 1 MB as $10^6$ bytes instead of $2^{20}$ to allow for easy comparisons with the numbers reported by IBM.

# 2 Implementation

This section describes the implementation details of SP-2 AM. It starts with a brief introduction to the SP-2 processing nodes followed by a description of the network adapter's hardware, the software interface to the adapter, and the basic mechanisms of sending and receiving a packet. It concludes with a discussion of the design and implementation of Active Messages, focusing on optimizations for bulk transfers and flow control strategies employed for reliable delivery.

## 2.1 SP-2 Network Interface Operation

Thin and wide SP-2 processing nodes are roughly equivalent to the RS/6000 model 390 and 590 respectively [4]. They have a clock speed of 66MHz and a peak performance of 266 Mflops. A thin node contains a 64 KB data cache with lines of 64 bytes each, a memory bus of 64 bits, a resident main memory of 64 to 512 MB, a SPECint92 of 114.3 and a SPECfp92 of 205.3. A wide node has a 256 KB data cache with lines of 256 bytes each, a memory bus of 256 bits, a resident main memory of 64 to 2048 MB, a SPECint92 of 121.6 and a SPECfp92 of 259.7.

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, highly scalable switch. The switch provides four different routes between each pair of nodes and has a hardware latency of about 500ns.

SP-2 nodes are connected to the high-speed interconnection switch via communication adapters[2]. The SP2 network adapter (TB2) [8] contains an Intel i860 microprocessor with 8 MB of DRAM. It is connected to the node via a 32-bit microchannel bus with a 80 MB/s peak transfer rate, and to the network switch via the Memory and Switch Management Unit (MSMU), whose interface is on the i860 bus. Data transfer between the MSMU and the microchannel are performed using two DMA engines and an intermediate 4KB FIFO. Direct programmed I/O from the Power2 to the 860's RAM is also possible.

The i860 on the TB2 uses microchannel DMA operations to provide a virtual copy of the TB2's buffers and control structures in the processor's main memory. Memory accesses to these virtual buffers produce a microchannel transfer from the processor to the TB2. Locations in the TB2's memory are cacheable by the main processor although no coherence is guaranteed.

The software interface uses a send and a receive FIFO to communicate between the CPU and the TB2 adapter [7]. The send FIFO has 128 entries while the receive FIFO has 64 entries per active processing node (determined at runtime). Each entry has 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 transmits the packet when the slot in the packet length array becomes non-zero. The send and receive FIFOs and the packet length arrays are memory-mapped TB2 structures.

SP-2 AM obtains the pointer to the adapter's next entry and uses this pointer to store the packet's data. A packet is sent by first putting the data into the next entry of the send FIFO along with the hardware header (destination node and route). The adapter will not see the data until the relevant cache lines are flushed out to main memory. The transfer size (1 byte) is then stored in the packet length array which goes through the microchannel. Bulk transfers can be optimized by writing the lenghts of 4 packets (4 bytes) 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 in a lazy fashion to avoid excessive traffic in the microchannel since microchannel accesses are expensive (around $1\mu s$).

## 2.2 Flow Control

The design of Active Messages is optimized for a lossless SP-2 switch behavior given that the switch is highly reliable. But packets can still be lost due to input buffer overflows and switch faults. Thus, flow control and fast retransmission have proved essential in attaining reliable delivery without harming the overall performance of the layer.

[2]The nodes are also connected to external network via TCP/IP built on top of the same TB2 adapter and IP device drivers.

Sequence numbers are used to keep track of packet losses and a sliding window is used for flow control; for retransmissions, unacknowledged messages are saved by the sender. 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 expected as well as subsequent packets. Whenever possible, acknowledgements are piggybacked with requests and replies; 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. Packets making up a chunk carry the same sequence number, and the window slides by the number of packets in a chunk. Each chunk is acknowledged explicitly, in order to avoid lengthy retransmissions of the entire data in case of packet losses. Initially, two chunks are transmitted and the next chunk is sent only when the previous to last chunk is acknowledged, avoiding overflow at the receiving end. The overhead for sending a chunk (175 $\mu$s) is higher than one round-trip, and thus the acknowledgement of chunk $N$ will most likely have arrived by the time chunk $N+1$ is sent. Note that with this chunk protocol, there is virtually no distinction between blocking and non-blocking stores for very large transfer sizes.

The receiving buffer can potentially be overflown if the sender issues asynchronous transfers of size smaller than one chunk in a pipelined fashion. The above flow control scheme also handles this case by ensuring that no more than one chunk of data remains unacknowledged at any time.

After sending a request the network interface is automatically polled whereas no polling occurs after sending a reply. There are two flow control windows per connection because requests and replies need separate buffering in order to avoid the following deadlock scenario: processor $P_1$ could issue a window-full of requests to processor $P_2$; $P_2$ could, without polling first, send a request to $P_1$ and then wait for a reply. However, $P_1$ would not be able to reply because it did not receive an acknowledgement from $P_2$ for all previous requests, i.e. its window remains full.

Decoupling buffering of replies and requests is not enough to prevent deadlock when windows for both types of messages are of the same length. As replies piggyback acknowledgements for requests and vice-versa, it would be necessary to handle acknowledgements piggybacked on requests *before* sending a reply, therefore increasing the round-trip latency. By making the reply window larger than the request, one can always send a reply for a request.

The window size for requests is chosen to be 75 packets (76 for replies) such that a chunk size of 8064 bytes (corresponding to 36 packets) yields the maximal asymptotic bandwidth. Given the flow control scheme, the window must be at least twice as large as the chunk size (72 packets), plus at least one additional packet. Although this exceeds the size of the preallocated 64-packet input FIFO per active node, the sender is unlikely to overflow the receiver's input buffer in practice.

A *keep-alive* protocol is triggered when messages remain unacknowledged for a long period of time[3]. This protocol forces negative acknowledgements to be issued to the protocol initiator, causing the re-transmission of lost messages (if any).

## 3 Round-trip Latency Measurements

This section describes the tests used to obtain performance data about round-trip latencies observed using SP-2 AM and contains the actual numbers.

The one-word round-trip latency was measured using the simplified code shown in Figure 1. It also illustrates the way Active Messages are used: the receiving side must poll, either explicitly, as shown below, or implicitly, when calling *am_request_*()* or *am_bulk()*.

The measured round-trip time is 51.0 $\mu$s and this value increases very slightly when two, three or four words with appropriate request and reply operations are transferred. Table 1 shows that the difference between the round-trip latencies for one and four words is about 1 $\mu$s. For transfer sizes larger than 16 bytes, bulk store and get operations are used to obtain round-trip latencies.

The number we report is higher than the latency of 46.6-47.0 $\mu$s which we were able to achieve over almost bare-bones hardware [7]. In the latter measurement, the message consists of only a 2-word header with no user data. The additional overhead of 4$\mu$s is attributed to the polling overhead which includes the costs of copy one word of data at each end as well as the flow control bookeeping.

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

```
int flag;
void sender_handler(int rep_vnn, int set_flag) { flag = set_flag; }
void receiver_handler(int req_vnn, int ignore) { am_reply_1(req_vnn, sender_handler, 1); }

/* sender */
GetClock(&start)
for (i = 0; i < repeat; i++) {
   flag = 0;
   am_request_1(RECEIVER, &receiver_handler, SENDER)
   while (flag == 0) am_poll();
}
GetClock(&end);
printf("round-trip time:  %f ", (float)(usecs(&start, &end)/repeat));

/* receiver */
for (;;) am_poll();
```

Figure 1: Measuring round-trip latency with requests and replies.

| *Bytes* | *Round-trip Latency (μs)* |
|---|---|
| 4 | 51.0 |
| 8 | 51.3 |
| 12 | 51.7 |
| 16 | 52.0 |

Table 1: Round-trip latency for small messages.

In order to transfer larger amounts of data SP-2 AM provides bulk transfer operations, namely *am_store()* or *am_get()*. The latency of data transfer as a function of data size is shown in Figure 2 (note that the bottom plot is a magnification of the initial part of the top plot). For both stores and gets, we measured the interval between issuing the call to the appropriate AM routine and getting notified that the operation completed successfully. In both cases, it amounted to the time necessary to transfer data from one processor to the other plus time needed to inform the initiating processor of the completion of data transfer. Thus, the two tests for latency with large data should give roughly the same results. As can be seen in Figure 2, the curves for gets and stores almost overlap.

# 4 Bandwidth Measurements

Several tests are used to measure the performance of bulk data transfers: the peak processor bandwidth, the asymptotic network bandwidth ($r_\infty$), the data size at which the transfer rate is half the asymptotic rate ($n_{\frac{1}{2}}$), the peak bandwidth of bulk transfers, and the bandwidth on exchange. synchronous and asynchronous transfers (*am_store* and *am_get*), and the bandwidth on exchange.

The processor bandwidth characterizes how fast the CPU copies data from user buffers to the virtual adapter buffers resident in main memory. The asymptotic network bandwidth indicates how fast the network adapter moves data from the virtual buffers to the network. The peak bandwidth of bulk transfers is obtained from the bandwidth of blocking and pipelined non-blocking transfers.

## 4.1 Benchmarks

Four benchmarks are used to measure the performance of bulk transfers. A simplified version of the code for bandwidth tests is shown in Figure 3. All benchmarks involves two processing nodes and are performed for data sizes varying from 16 bytes to 4 MB:

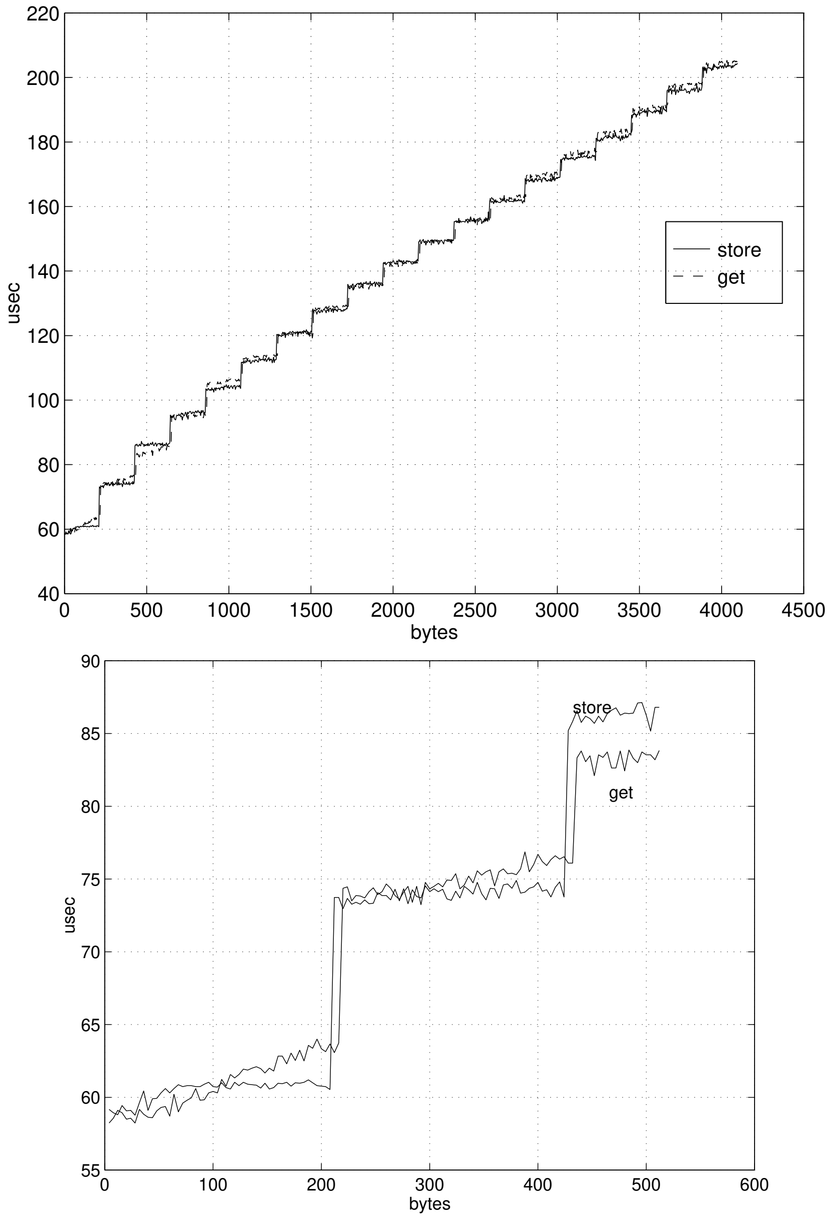


Figure 2: Latency when using am_store() and am_get() for message size from 16 to 4K bytes. The bottom plot is a magnification of the small-message part of the top plot.

```
int done = 0;
void store_handler1(int node, void *rva, int nbytes, void *arg)  done++;
void store_handler2(int node, void *rva, int nbytes, void *arg)  done++;
void done_handler(int node, int arg)  done = 1 ;

/* sender */
GetClock(&start);
for (i = 0; i < repeat; i++) {
#ifdef NON_BLOCKING
   am_store_async(receiver, table &sink, size,
      store_handler1, (void *) 0, store_handler2, (void *) 0);
#else
   am_store(receiver, table, &sink, size, store_handler1, (void *) 0);
#endif
}
#ifndef NON_BLOCKING
while (!done) am_poll();
#endif
GetClock(&end);
printf("store bandwidth = %f ", (float)repeat * size / usecs(&start, &end));

/* receiver */
while (done < repeat) am_poll;
am_request_1(sender, done_handler, 0);
```

Figure 3: Measuring bulk store bandwidth. If NON-BLOCKING is defined, the test uses asynchronous stores; otherwise it uses synchronous (blocking) stores.

- *Processor and Network Bandwidth*
  The peak processor bandwidth is measured by issuing one asynchronous transfer request (*am_store_async*) that is smaller than the virtual FIFO and measuring the copy time. For very large transfer sizes (larger than 1 MB), the processor bandwidth converges asymptotically to the network bandwidth as the CPU overhead accounts for cycles stolen by the microchannel DMA and back pressure from the network.

- *Blocking Transfer Bandwidth*
  The bandwidth of synchronous transfer requests is measured by issuing blocking requests (*am_store* and *am_get*) and waiting for their completion.

- *Pipelined Asynchronous Transfers*
  Another way of doing large bulk transfers is to pipeline a number of small transfer requests. Pipelining asynchronous transfers while waiting until completion allows us to better observe how fast the rate reaches its peak because the CPU and the DMA inevitably have to arbitrate for the microchannel (even for small amounts of data). This benchmark is performed using a total transfer size of 1 MB and by varying the request sizes from 64 bytes (15625 requests) to 1 MB (1 request).

- *Bandwidth on Exchange*
  This test measures the bandwidth achieved when swapping an array of integers between two processing nodes which initiate the data transfer roughly at the same time. Two versions of the same test were implemented. The first version uses a third processing node to start up the other two nodes, thereby synchronizing the data exchange which consisted of equally sized bulk store requests between the two nodes. The test is complete when both participants are sure that their data transfer is complete. The second version has one processing node issue a get request followed by a store request while the other node simply polls. Figure 4 shows the schematic view of both tests.

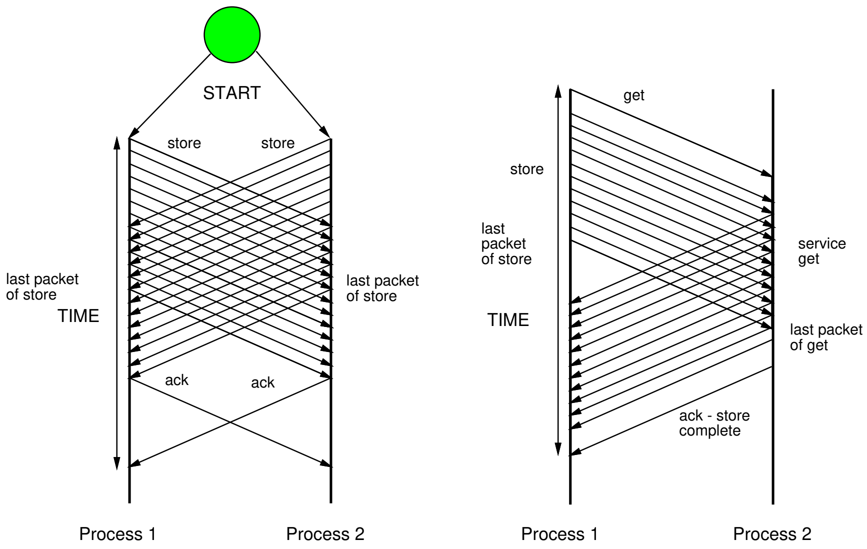


Figure 4: Schematic view of both tests of bandwidth on exchange.

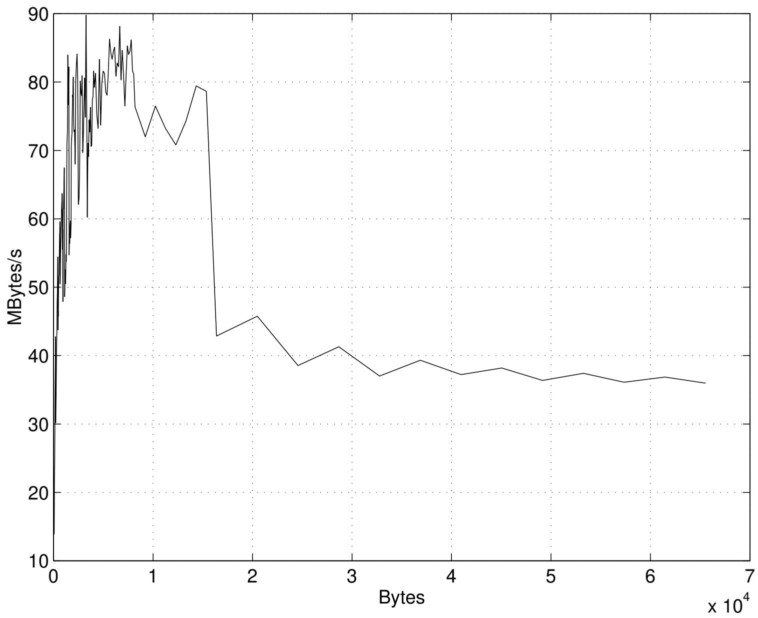


Figure 5: Processor Bandwidth

## 4.2 Results and Discussion

The curve in Figure 5 shows the processor bandwidth using asynchronous stores. The CPU can copy packets into the adapter's virtual buffer at up to 80 MB/s which is much higher than the rate at which the packets are transferred across the microchannel or the network. The asymptotic network bandwidth ($r_\infty$) is 34.3 MB/s (see also Table 2). The bandwidth reaches half of 80 MB/s for messages as small as 70 bytes. The sawtooth behavior observed for small messages is due to the packet size of 256 bytes. As an extra packet is needed, the total transfer overhead increases. The sharp decrease in the processor bandwidth at about 16

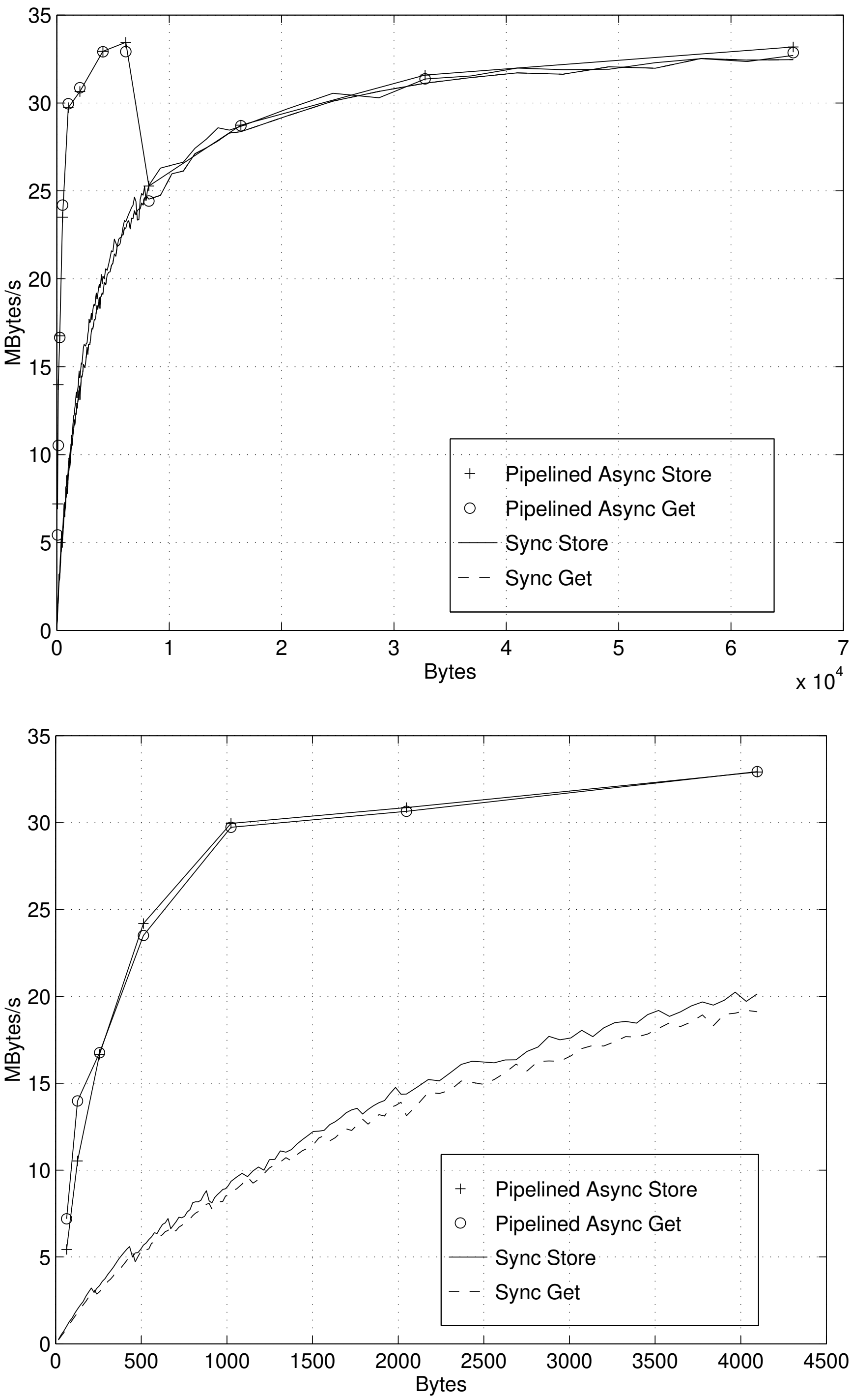


Figure 6: Bandwidth measurements of blocking and non-blocking bulk transfers. The bottom plot is a magnification of the small-message part of the top plot.

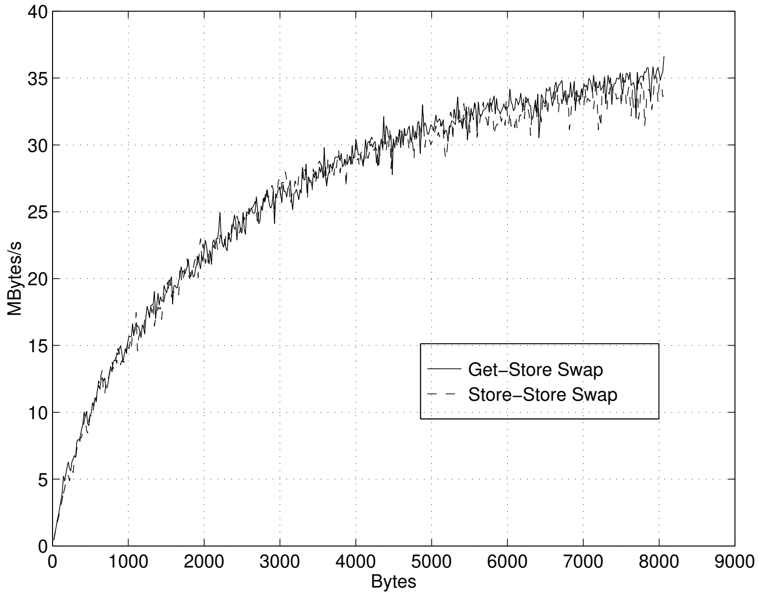


Figure 7: Aggregate Bandwidth on Exchange.

KB illustrates the activation of the flow control protocol.

| *Size (MBytes)* | *Bandwidth (MB/s)* |
|---|---|
| 1 | 34.50 |
| 2 | 34.36 |
| 3 | 34.35 |
| 4 | 34.27 |

Table 2: Asymptotic network bandwidth

Figure 6 shows the bandwidth achieved when transferring 1 MB of data by pipelining asynchronous stores and gets of a given size. The bandwidth tends to $r_\infty$, and rises at a much slower rate than the processor bandwidth because CPU cycles are being stolen by the microchannel DMA and by the back pressure from the network. Half the peak bandwidth ($n_{\frac{1}{2}}$) is reached at about 260 bytes. The bandwidth of synchronous stores and gets also converges to $r_\infty$ but at a slower rate. Also, for smaller transfer sizes, the curve for gets is slightly lower than for stores because of the overhead for issuing a get request. Consequently, the bandwidth for gets shows an $n_{\frac{1}{2}}$ of 3000 bytes compared to the 2800 bytes for synchronous stores. The effect of this overhead on the bandwidth vanishes as the transfer size increases, explaining the overlapping of both curves for sizes larger than 4 KB. Figure 6 clearly illustrates that asynchronous transfers are no better than their blocking counterparts for message sizes larger than one chunk, which is when the flow control kicks in.

In Figure 7, we observe that both exchange bandwidth tests yield the same curves with an asymptotic bandwidth of about 40 MB/s.

# 5 Overheads Measurements

This section contains the costs of calling *am_request_*()*, *am_reply_*()* and *am_poll()*. The cost associated with calling *am_store()* depends on the size of the data transferred and can be computed from processor bandwidth curve in Figure 5.

Table 3 summarizes the time needed to complete a successful request or reply call. The only difference between those two kinds of operations is that *am_request_*()* calls *am_poll()* after the message is sent, as opposed to *am_reply_*()*, which does not poll. The time needed to complete *am_poll()* may vary, as it depends on the number and kind of messages received from the network interface. The *am_request* column in Table 3 contains results for *am_request_*()* when no messages are received by the poll. Thus, for example, the first column in Table 3 says that a call to *am_request_1()* takes 7.7 $\mu$s while a call to *am_reply_1()* takes 4.0 $\mu$s.

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

Table 3: Cost of calling am_request_*() and am_reply_*() functions, in $\mu$s.

An unsuccessful polling overhead of 1.3 $\mu$s was measured by having a processing node constantly call *am_poll()* and making sure no messages are received. It should be noted that this is an average overhead of *am_poll()* when polling is done in a very tight loop; when called sporadically, its cost may be much higher.

As shown in Figure 8, the additional overhead per received message is about 1.8 $\mu$s. In this experiment, the parameter was a number of messages waiting in the network interface for *am_poll()*. Every message was an *am_request_1* with a handler returning immediately. Slight jumps are observable every 19 messages, i.e. quarter of the window size in the current implementation.

# 6 Comparison with MPL

This section compares the main performance characteristics of SP-2 AM with IBM's Message Passing Layer (MPL) [5]. The comparison is summarized as follows:

| | *Round-trip Latency* | | *Pt-Pt Bandwidth* | |
|---|---|---|---|---|
| *Node Type* | *MPL* | *AM* | *MPL* | *AM* |
| 66 MHz "Thin" | 80.0 $\mu$s | 51.0 $\mu$s | 35.4 MB/s | 34.3 MB/s |
| 66 MHz "Wide" | 78.4 $\mu$s | 50.3 $\mu$s | 35.6 MB/s | 34.6 MB/s |

Table 4: Performance Comparison between IBM MPL and SP-2 AM

MPL's user space round-trip latency is taken as twice the latency of sending a zero-byte message between two processing nodes using *mp_bsend* and *mp_brecv* from the MPL library. The SP-2 AM latency reported in Table 4 refers to a one-word ping-pong message. The asymptotic bandwidth of SP-2 AM is slightly smaller than the MPI's asymptotic bandwidth for unknown reasons.

# 7 Split-C Application Benchmarks

Split-C 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-2 AM on applications written in a parallel language. A Split-C program is comprised of a thread of control per processor from a single code image and the threads interact through reads and writes on shared data. The type system distinguishes between local and global pointers such that the compiler can issue the appropriate calls to Active Messages whenever a global pointer is dereferenced. Thus, dereferencing a global pointer to a scalar variable turns into a request and reply Active Message sequence exchange with the processor holding the data value. Split-C also provides bulk transfers which map into Active Messages bulk gets and stores to amortize the overhead over a large data transfer.

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 the IBM SP-2. A small set of application benchmarks is used here to compare the SP-2 version of Split-C to the CM-5, Meiko CS-2, and U-Net cluster versions. This comparison is particularly interesting as the CM-5, CS-2, and U-Net cluster machines are easily characterized with respect one another as shown in Table 5: the CM-5's processors are slower than the Meiko's and the U-Net

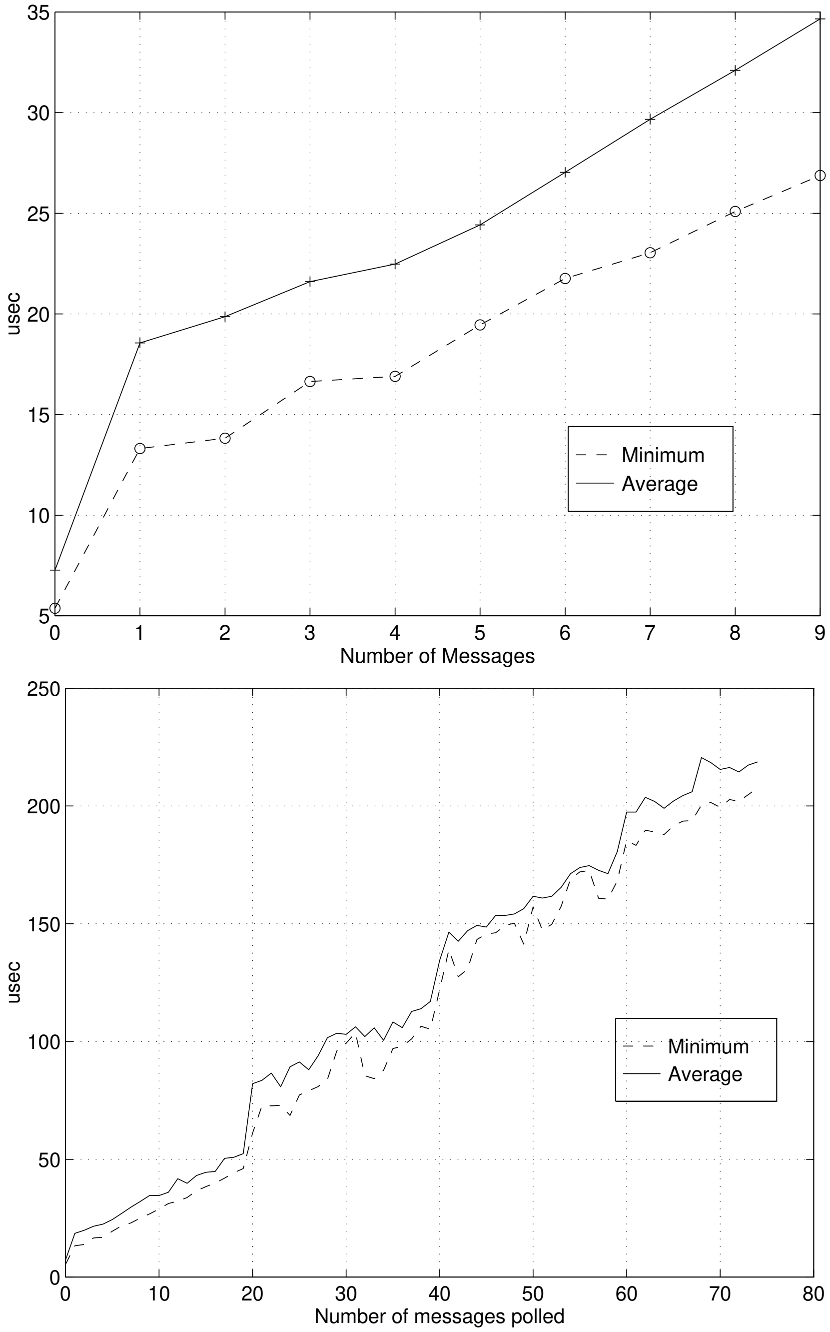


Figure 8: Overhead of am_poll() as a function of the number of messages.

cluster's, but its network has lower overheads and latencies. The CS-2 and the U-Net cluster have very similar characteristics with a slight CPU edge for the cluster. The SP-2 has the fastest CPU among its counterparts, a network bandwidth comparable to the CS-2, and a message overhead that is competitive to both the CM-5 and the U-Net cluster.

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

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

The Split-C benchmark set used here consists of five 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 6. Execution times normalized to the SP-2 are shown in Figure 9.

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

Table 6: Absolute Execution Times (seconds)

The matrix multiply uses matrices of 8 by 8 blocks with 128 by 128 double floats each block, and of 16 by 16 blocks with 16 by 16 double floats each. The main loop multiplies two blocks while it prefetches the two blocks needed in the next iteration.

The results show that the SP-2 outperforms the CM-5 for large size of blocks because of its high bandwidth. It also shows that the floating-point performance of Power2 CPUs give the SP-2 an additional edge over the CM-5, CS-2, and the U-Net/ATM cluster. Furthermore, notice that the SP-2 matrix multiply benchmarks exhibit a smaller network time for smaller blocks (e.g 16 by 16) as opposed to other machines. With smaller blocks, although more bulk transfers are performed, the size of the transfers are smaller. As long as the transfer sizes remain smaller than 8064 bytes, flow control is not activated which explains the better network time compared to larger blocks. Because of its high network bandwidth, the SP-2 does not suffer from a message traffic increase.

The sample and radix sort benchmarks sort an array of 4 million 32-bit integers with arbitrary distribution. As in most parallel sort algorithms, the radix sort employs alternating phases of variations on local sort and key distribution involving irregular, balanced all-to-all communication. The algorithm performs a number of passes over the keys where each pass consists of three steps: on each processor, a local histogram is computed based on its set of local keys; a global histogram is formed from the local histograms which allows the rank of each key to be determined; and keys are distributed to its sorted position based their ranks. The version optimized for small messages sends two keys at a time to other processors whereas the one optimized for large messages sends all the keys at once.

Instead of alternating computation and communication phases, the sample sort algorithm has a single key distribution phase. It first picks 64 samples on each processor, sorts all the samples on one processor, and then selects splitters to determine which range of values should end up on each processor. The splitters are then broadcast to the processors, the main communication phase permutes all the values to the right processor, and finally each processor sorts its values locally (which contributes most to the computation time). The version optimized for small messages packs two values per message while the one optimized for bulk transfers presorts the local values such that each processor sends exactly one message to every other

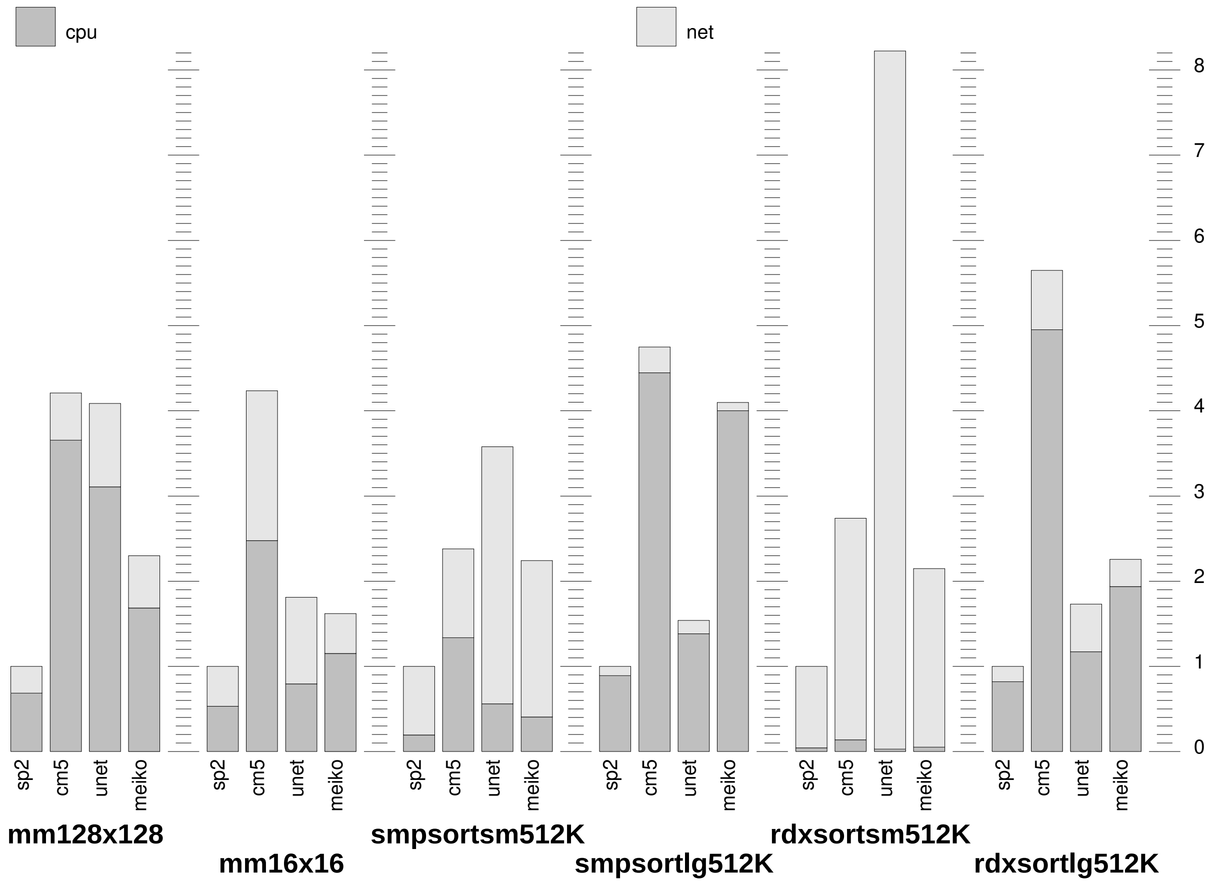


Figure 9: Split-C benchmark results normalized to SP-2.

processor during the permutation phase.

Figure 9 shows that the SP-2 spends less time in local computation phases because of the faster CPU. For small messages, SP-2 spends about the same amount of time, if not less, in the communication phases as the CM-5 and CS-2. Although SP-2's round-trip latency is relatively higher, it combines low message overhead with high network bandwidth which yields a higher message throughput. For large messages (albeit not large enough to activate flow control), SP-2 again outperforms its counterparts in both computation and communication phases.

## 8 Conclusions

This paper presents a very efficient implementation of Active Messages on the SP-2, making it an attractive alternative to IBM's MPL and enabling an application/compiler writer to fully utilize the communication capabilities of the SP-2 without sacrificing reliability. The reported round-trip latency of 51 $\mu$s for *am_request_1()* is only 2$\mu$s above the raw hardware. The latency for short data transfers over Active Messages is much better than performance of MPL, while the bandwidth of MPL is slightly better. The two DMA engines and an i860 processor of the SP-2's network adapter were positive factors whereas the lack of cache coherence and the high microchannel transfer costs were negative ones. The Split-C benchmarks confirm the performance results from our raw latency and bandwidth measurements and the superiority of the SP-2 among other MPPs such as CM-5 and Meiko.

## 9 Acknowledgements

We are grateful to Jamshed Mirza and M. T. Raghunath (IBM Kingston) for their invaluable help, and the Cornell Theory Center (CTC) for providing us computing time on the IBM SP-2 during the initial stages of the projec . We also acknowledge the assistance provided by Mark Smith (CTC/IBM) and Steve Szalewicz (CTC/IBM). 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 supported by a doctoral fellowship (200812/94-7) from the Brazilian Research Council (CNPq/Brazil).

## References

[1] Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th Symposium on Operating Systems Principles*, Cooper Mountain, December 1995.

[2] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumeta, and T. von Eicken. Introduction to Split-C. In *Proceedings of Supercomputing*, 1993.

[3] David Culler et al. Generic Active Message Interface Specification v. 1.1, November 1994.

[4] IBM. SP-2 Command and Technical Reference, December 1994.

[5] IBM. SP2PERF 1.7, 1994.

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

[7] M. T. Raghunath. Personal Communication, 1995.

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

[9] 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 International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.