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

Chi-Chao Chang

Computer Science Dept.

Cornell University

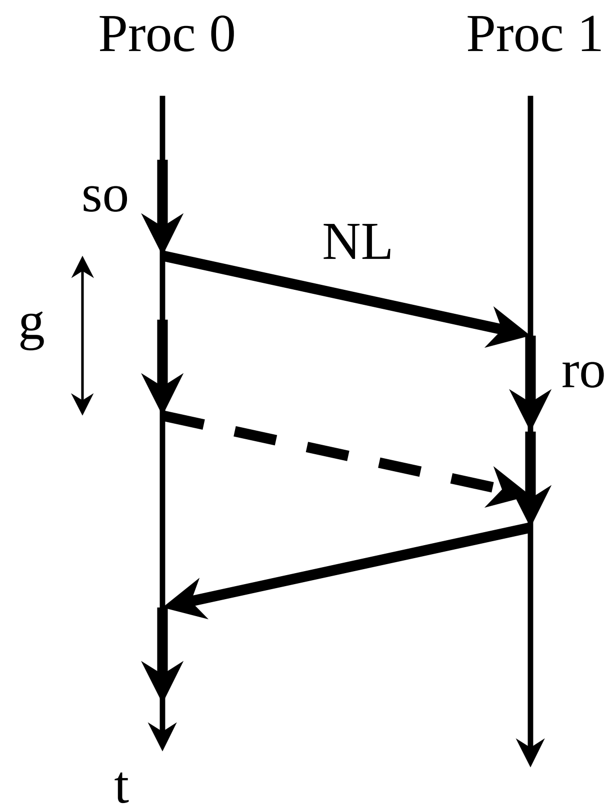
(joint work with Grzegorz Czajkowski, Chris Hawblitzel, and Thorsten von Eicken)

Paper’s Focus

- Efficient Communication on the IBM SP
  - SP’s network latency lags other MPPs
  - Focus on low message overhead
  - Implementation of Active Messages on top of SP’s adapter
- Active Messages as powerful communication primitives
  - Attractive alternative to IBM’s MPL
  - Parallel languages: Split-C
  - Parallel tools: MPI

Some definitions...

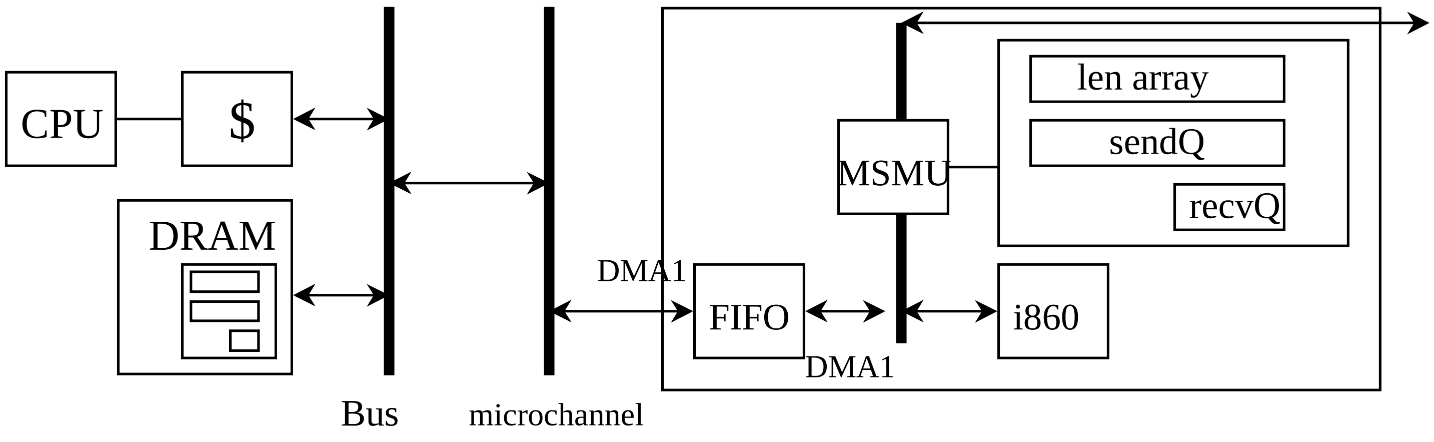
- **Overheads:**
  - **send (so)**
  - **recv/poll (so)**
- **Latencies**
  - **network latency (NL)**
  - **L = so + NL + ro**
  - **round-trip = 2 L**
- **Bandwidth (g)**



SP Network Adapter

Basic Memory Mapped Structures:

- two FIFOs between CPU and adapter
  - send FIFO: 128 entries of 256 bytes each
  - recv FIFO: N * 64 entries (N = # of active nodes)
- send packet array
  - each entry (1 byte) correspond to one entry in send FIFO
  - adapter transmits packet when slot in array becomes non-zero
- pointer to last filled recv FIFO entry



Active Messages

- **Efficient paradigm**
  - overlaps computation with communication
  - handler associated with message
- **GAM Spec 1.1**
- **Small messages**
  - requests and replies
  - up to 4 words of arguments
- **Bulk Transfers**
  - stores and gets
  - specifies remote memory address
- **Explicit polling**
- **Unmodified SP adapter**
  - memory mapped send/recv FIFOs

Active Messages Interface

Short Messages

```
am_request_M
am_reply_M (int dest, hdler_t h, int a0, ..., int aM-1)
```

Bulk Transfer

```
am_store
am_get(int node, void *lva, void *rva, int size, hdler_t h,
   void *a)
am_store_async(int dest, void *lva, void *rva, int size,
   hdler_t h0, void *a0, hdler_t h1, void *h1)
```

Polling

```
am_poll()
```

Basic Send and Receive

Send:

- fill next FIFO entry with data and header
- flush data cache lines so that adapter can see them
- store transfer size into packet length array

Receive:

- check recv pointer
- copy data from recv FIFO to user buffer
- flush the entry from the data cache for wrap-around
- pop entry from recv FIFO

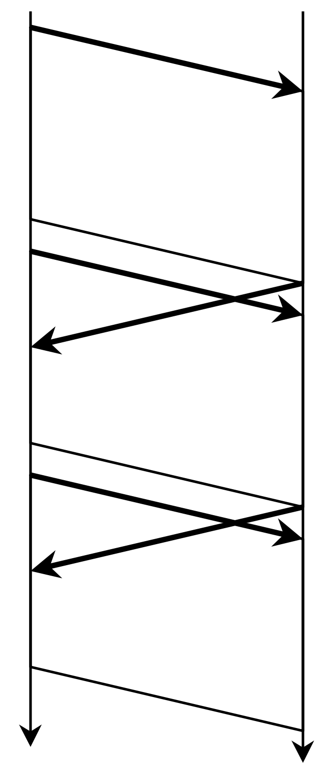
SP AM vs. Other Platforms

Machine	CPU Speed	Round-trip	Bandwidth
TMC CM-5	33 MHz Sparc-2	12 μs	10 MB/s
Meiko CS-2	40 MHz Sparc-20	25 μs	39 MB/s
U-Net/ ATM	50/60 MHz Sparc-20	66 μs	14 MB/s
IBM SP AM	66 MHz RS6k/590	51 μs	34.3 MB/s
IBM SP MPL	66 MHz RS6k/590	88 μs	34.6 MB/s

Flow Control

Reliable Transmission

- **Because...**
  - of receive buffer overflows
  - network is not 100% reliable
  - of switch faults
- **Design Criteria**
  - data divided into chunks (8 Kb)
  - one ack/chunk: avoid lengthy resends
  - double buffer overlap: tuning chunk size is important



Split-C

- Supercomputing ‘93
- Parallel extension to C
- Built on top of Active Messages
- Control Model: SPMD and barriers
- Global Address Space: global pointers
- Split-phase assignments
- One-way stores
- Bulk transfers: put and get

Split-C Benchmarks

Blocked Matrix Multiply

- loop: multiples two blocks while prefetches next two blocks
- different block sizes: 128x128 and 16x16 doubles

Sample Sort

- single key distribution phase
- comm phase: permutes values to the right processors
- local sort

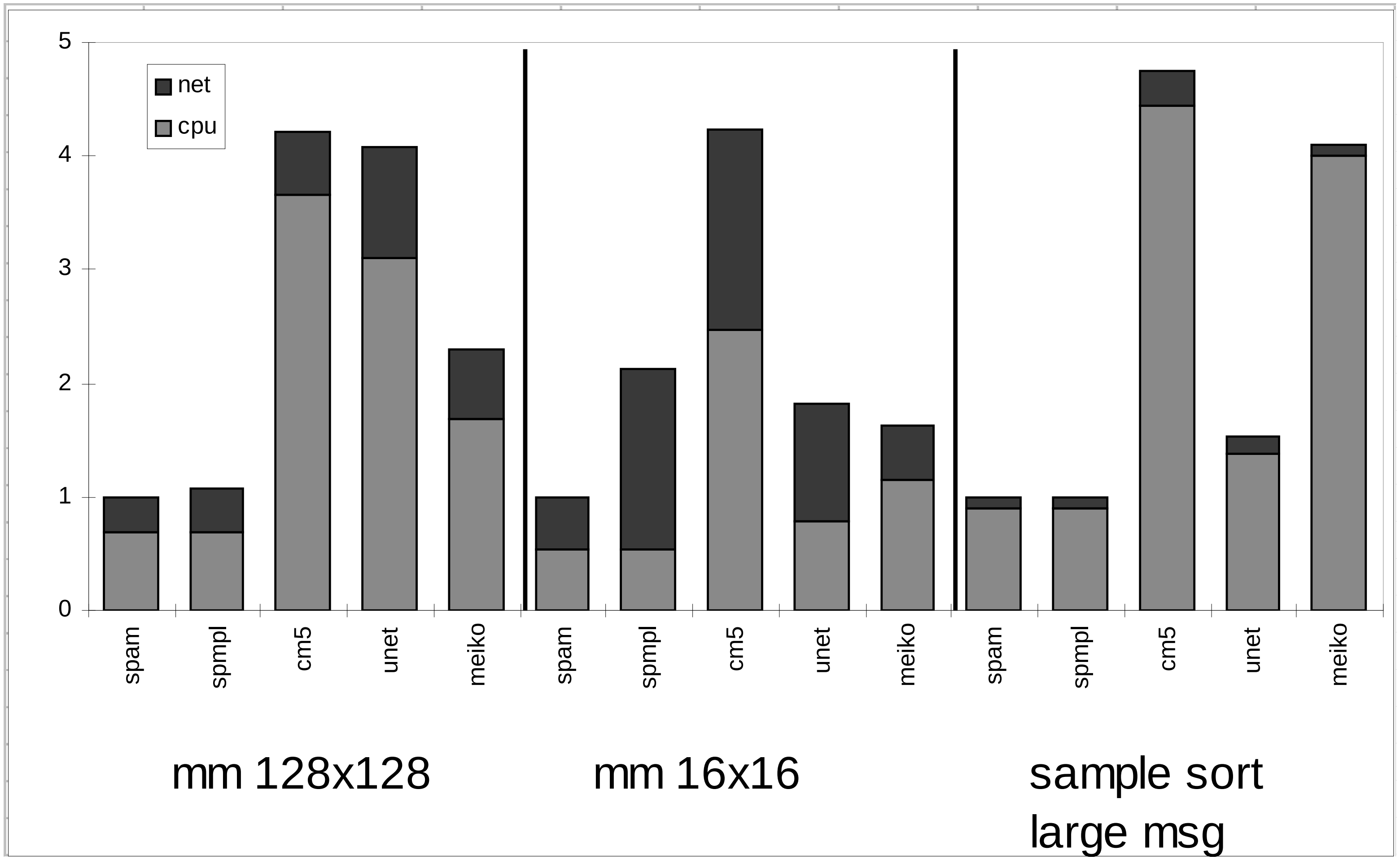
Radix Sort

- Alternating phases of local sort and irregular all-to-all comm:
  - local histogram computation
  - global histogram formed from local histograms
  - key distributed based on global histogram
  - local sort

Radix and Sample:

versions for small and large messages

Split-C Benchmarks



MPI over Active Messages

- Used freely available MPICH
- Simple port due to Abstract Device Interface (ADI)
- Focus on MPI point-to-point calls:

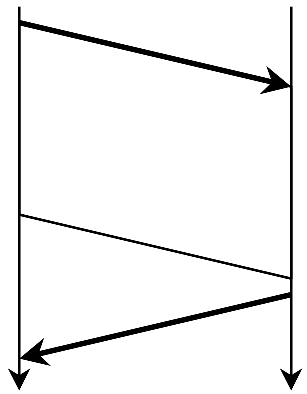
```
MPI_Send(void *buf, int size, MPI_Datatype dtype, int
   dest, int tag, MPI_Comm comm)
MPI_Recv(void *buf, int size, MPI_Datatype dtype, int
   dest, int tag, MPI_Comm comm, int *status)
```

Main Issue: Receive Buffer Naming

Implementations

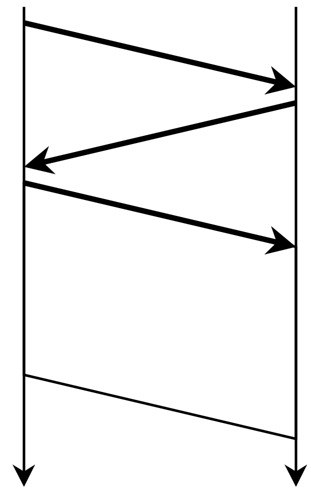
Buffered Protocol

- Sends to tmp buffer at receiver
- Receiver copies to user buffer
- Receiver sends ack back



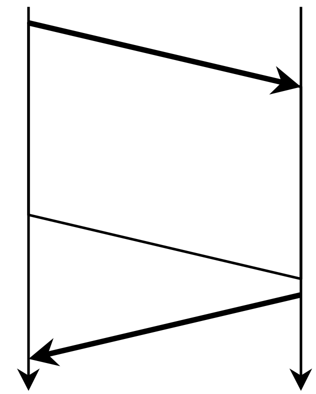
Rendez-vous Protocol

- Posts send to receiver
- Receiver replies with recv buffer
- Sends directly into recv buffer



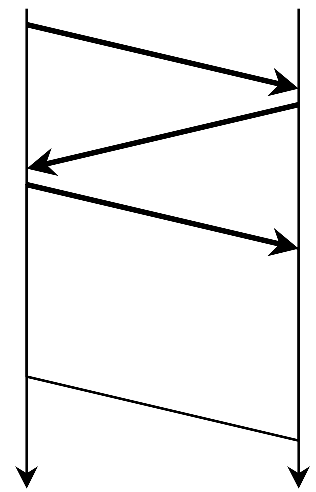
Buffered Protocol

- **Receiver holds 16K of buffer for every other processor**
- **Sender manages the receive buffers locally**
  - **First-fit allocation of receive buffers**
- `am_store` **or** `am_store_async` **transfers data**
  - **2 bytes for tag, 1 byte for sender's local rank, 1 byte for communicator**
- **Receiver issues** `am_reply_1` **or** `am_request_1` **to sender to free buffer space**
  - **1 word for temporary buffer address**



Rendevous Protocol

- `am_request_4` **from sender to receiver**
  - **one word for send request pointer, sender's local rank and communicator, tag, message length**
- `am_reply_4` **or** `am_request_4` **from receiver to sender**
  - **one word for send request pointer, receive request pointer, buffer address, buffer length**
- `am_store` **or** `am_store_async`
  - **one word for receive request pointer**

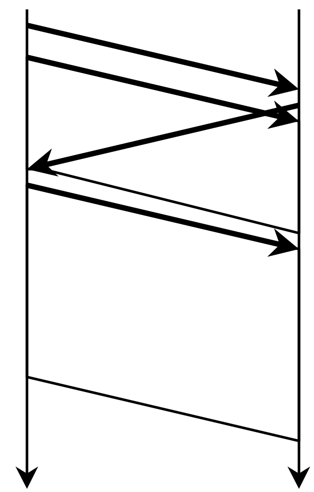


Comparison

Optimization

Hybrid Protocol

- Requests for receive buffer
- Sends a small prefix of data (4K) into tmp buffer
- Sends the rest of data into receive buffer



All three protocols are easily mapped onto Active Messages

MPI Latencies

MPI Bandwidth

NAS Benchmarks

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

Summary

- Efficient implementation of Active Messages on the SP
- Active Messages as powerful communication primitives
  - Low level optimizations done once and for all
  - Complex layers of software on AM without compromising performance
- Split-C: low overhead and high message throughput compensates for high network latency
- MPICH: competitive to highly tuned MPI-F

www.cs.cornell.edu/Info/Projects/CAM