An Intro to Distributed Memory Computing and MPI

Rob Schreiber

Stanford ICME

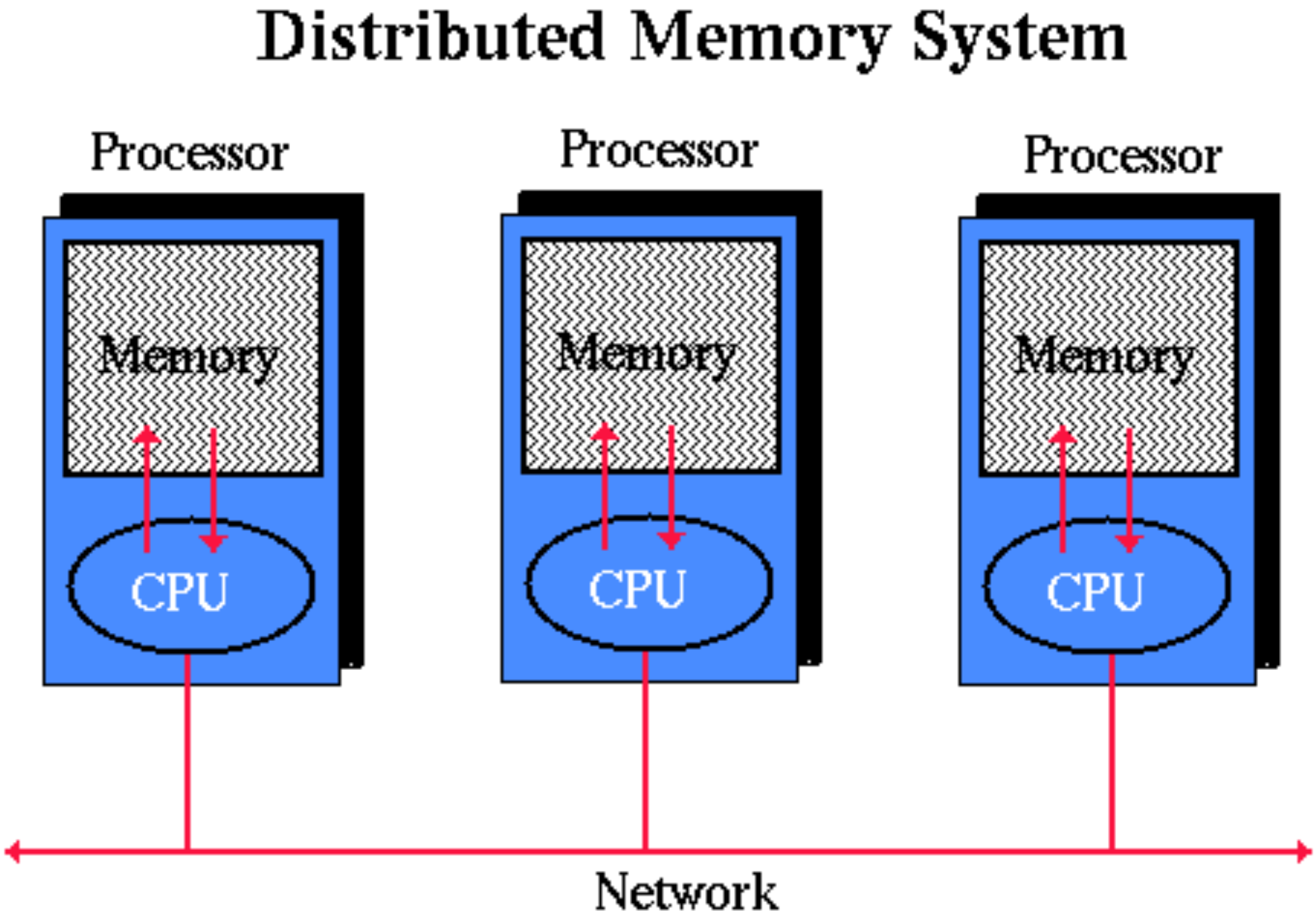
MIMD, SIMD, SPMD (models of) parallel machines

- MIMD: Each processor runs its own program, has its own program counter, works on its own data
- SIMD: All the processors share one program and program counter, but each processor has its own data
- SPMD: Like MIMD, but you write one program and a copy runs on each processor
- myprocid, numprocs – Built-in variables in the SPMD model, allow programs to know what their role is. SPMD is really MIMD.

Two kinds of parallel machines

- Shared-memory multiprocessors
  - SMPs share the global memory of a GPU
  - The cores in a server share the main memory of the server
  - (Every core has its own set of registers.)
- Distributed-memory multiprocessors
  - Lots of smaller “nodes”, each a multicore with its own internally shared memory. Normally, a node can load/store only its own memory.
  - A.K.A. Multicomputer
- Scale-up vs. scale-out.

A multicomputer / cluster



How do you write and run programs?

- gcc myprog.c –o node.out
- Login to bigcluster
- Bigcluster_login>> jobsubmit –nprocs 512 node.out

- Starts node.out on each of 512 nodes of the cluster.

Embarrassingly parallel problems

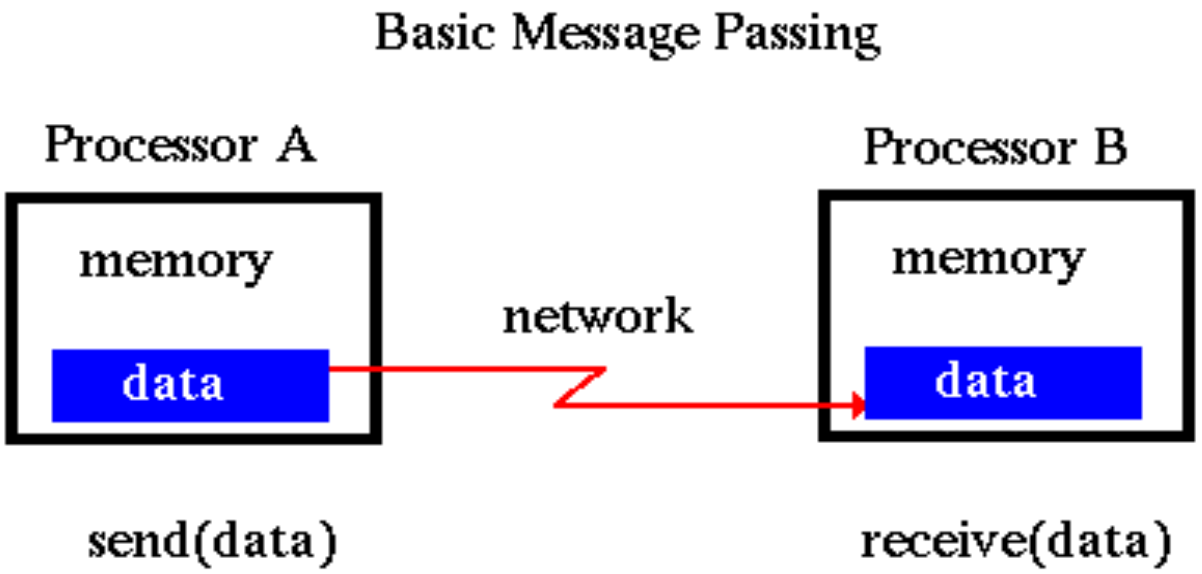
- Every node computes something, independent of the other nodes
- They share nothing
- They do not communicate
- Efficient if it is load-balanced

Can the nodes actually cooperate? Share data?

- There is no shared memory with which to share data
- How can they cooperate?
- Send messages
- Like the postal service: message goes in at one node, with a destination address, the communication system delivers it (reliably, later) to the intended recipient
- Broadcast is possible. (Junk mail?)

A message is sent and received

- Node A: send(A_buffer, len, node_B_id, tag)
- Node B: recv(B_buffer, len, node_A_id, tag, &actlen)
- Messaging library hides LOTS of details →portability



What are the limits of parallelism?

- Amdahl (builder of big business machines) concluded early that they would be useless
- Amdahl's law: useful processors proportional to reciprocal of the serial fraction
- But he was wrong: today, world's most powerful machine has over 200,000 processor cores
- Conclusion – we've been able to drive the serial fraction in scientific computing towards zero

Why are we learning to program a cluster?

- Invented by Chuck Seitz, Geoffrey Fox, Caltech, 1981++. 64 early PC (i8086) processors.
- The world is using them more and more
  - Google, et al
  - Large scientific machines – all of them are clusters
- Very large machines that share memory have disappeared
  - Silicon Graphics, Inc.
- Portable parallel code is now written this way
- The standard has stood the test of some time (approaching 20 years)
- Why has this happened?

Why are clusters everywhere?

- Cheap
  - Many copies of standard servers
- Fault tolerant
  - Still quite usable with many failed nodes
- Memory grows with processing
- Memory performance grows with processing
- Simple programming model: message passing

Why do processes communicate?

- One produces data that the other consumes:

(compute process) → (disk process)

(Both the data, and the synchronization)

- They share a resource:

(proc 1) → I/O channel ← (proc 2)

(Synchronization)

Synchronization is part of the message passing

Blocking send and receive

- Node A:

  send(a, node_B_id, tag);

  for (i=0; i<len; i++) *a++ = newdata(i);

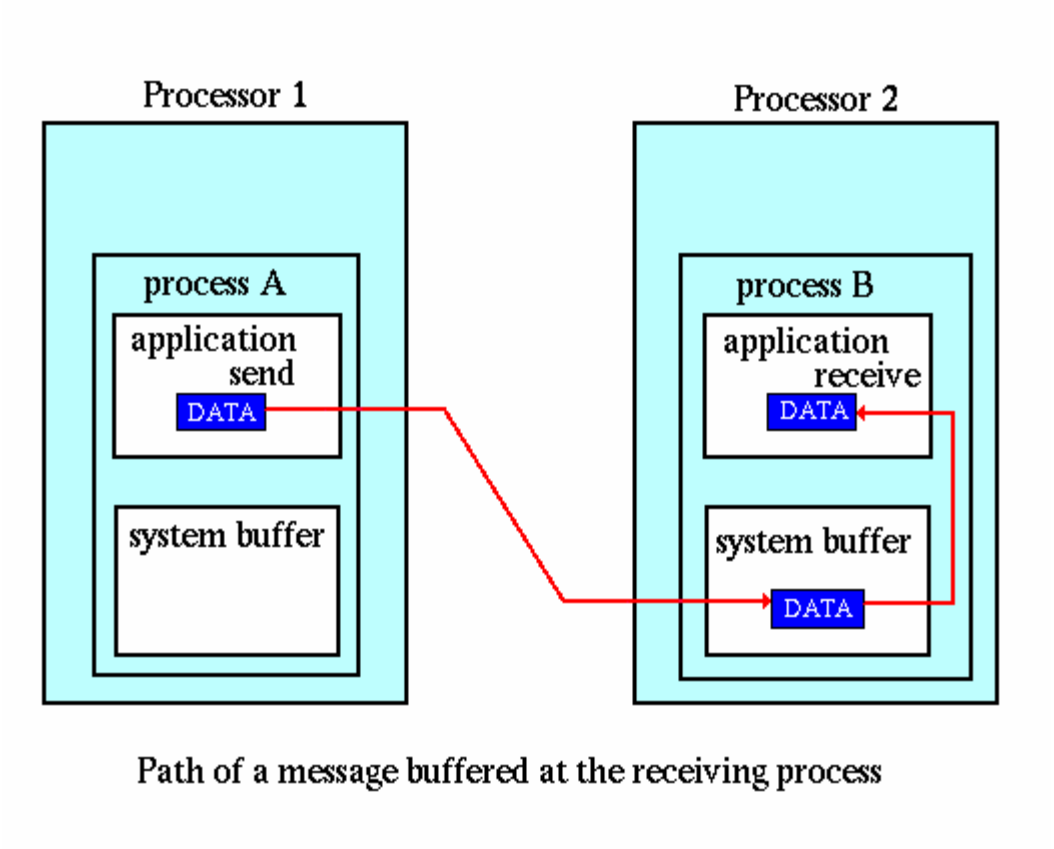
  /* the data may not yet have been received */

- Node B:

  recv(b, len, node_A_id, tag, &actlen)
  for (I = 0; I < actlen; i++) x += *b++;

- The communication action (at the calling node) is complete when control returns to the calling program

One possible implementation, buffer at destination



Path of a message buffered at the receiving process

Other implementations

- Send waits for receive to occur, moves data directly to the destination buffer
- Buffer on the sender

Collective operations

- Let's compute dotproduct(x,y,n) in parallel
- Sequential:

```
for (sum=0, i = 0; i<n; i++) sum += x[i]*y[i];
```

- Create x and y as distributed vectors, with identical distributions

```
/* node code: all data local to one processor, so the
   vector x is the concatenation of the vectors lx on
   each of the processors */
for (lsum=0, i = 0; i<my_n; i++) lsum += lx[i]*ly[i];
sum_reduce(lsum, &totalsum); /* communicate
   and add up the local sums, broadcast the result*/
```

sum_reduce(localx, &sumofx)

```
/* add up everyone's localx */
 send(localx, 0);  /* all send to zero */
/* node 0 does the adding */
if (myprocid == 0)  {
     s = 0;
     for (from = 0 ; from < numprocs; from++)  {
          recv(&temp, from);
          s += temp;
          }
     /* broadcast from zero to all others
    for (to = 0; to <numprocs; to++)
          send(s, to);
    } /* end of the work for node zero
recv(&ssumofx, 0); /* everyone else is waiting */
```

An asynchronous sum_reduce(localx, &sumofx)

```
/* add up everyone's localx */
 send(localx, 0);  /* all send to zero */
/* node 0 does the adding */
if (myprocid == 0)  {
     s = 0;
     for (k = 0 ; k < numprocs; k++)  {
          recv(&temp, any_source);
          s += temp;
          }
     /* broadcast from zero to all others
    for (to = 0; to <numprocs; to++)
          send(s, to);
    } /* end of the work for node zero
recv(&ssumofx, 0); /* everyone else is waiting */
```

What is good/bad about my sum_reduce?

- It is simple to write and understand
- Not much communication
- Load is not balanced (node 0 does all the work)
- Numprocs messages received and sent by node zero
- It might deadlock
- Why might it deadlock?
- Can you reduce numprocs to log(numprocs)?

MPI_REDUCE

- MPI provides reduce operations that hide the implementation details
- All nodes must call before it can return: collective communication
- Similarly: MPI_BROADCAST
- Other collectives

Deadlock and sequentialization

- With 2 processes
  send(x, 100000, 1 - myid);
  recv(y, 100000, 1 - myid);
- MPI_SEND blocks on both processes waiting for the receive on the other process, because the message is big

Simple fix to MPI deadlock

```
handle = irecv(y, 100000, 1-myid);
send(x, 100000, 1-myid);
wait(handle);
```

- irecv does not block. The data are NOT in y until after the wait.
- The MPI envelope protocol -- how does MPI do this?

De-Sequentialization

```
if (myid > 0) handle = irecv(y, 100000, myid-1);
if (myid < n-1) send(x, 100000, myid+1);
if (myid > 0) wait(handle);
```

Sequentialization

- A row of processes: 0, 1, …, n-1
- All have to send something to the right, except n-1

```
if (myid < n-1) send(x, 100000, myid+1);
if (myid > 0) recv(y, 100000, myid-1);
```

- No deadlock, since n-1 always recv's, then n-2 sends, then n-2 recv's, etc, etc.
- No deadlock,
- But this is not a good parallel program!

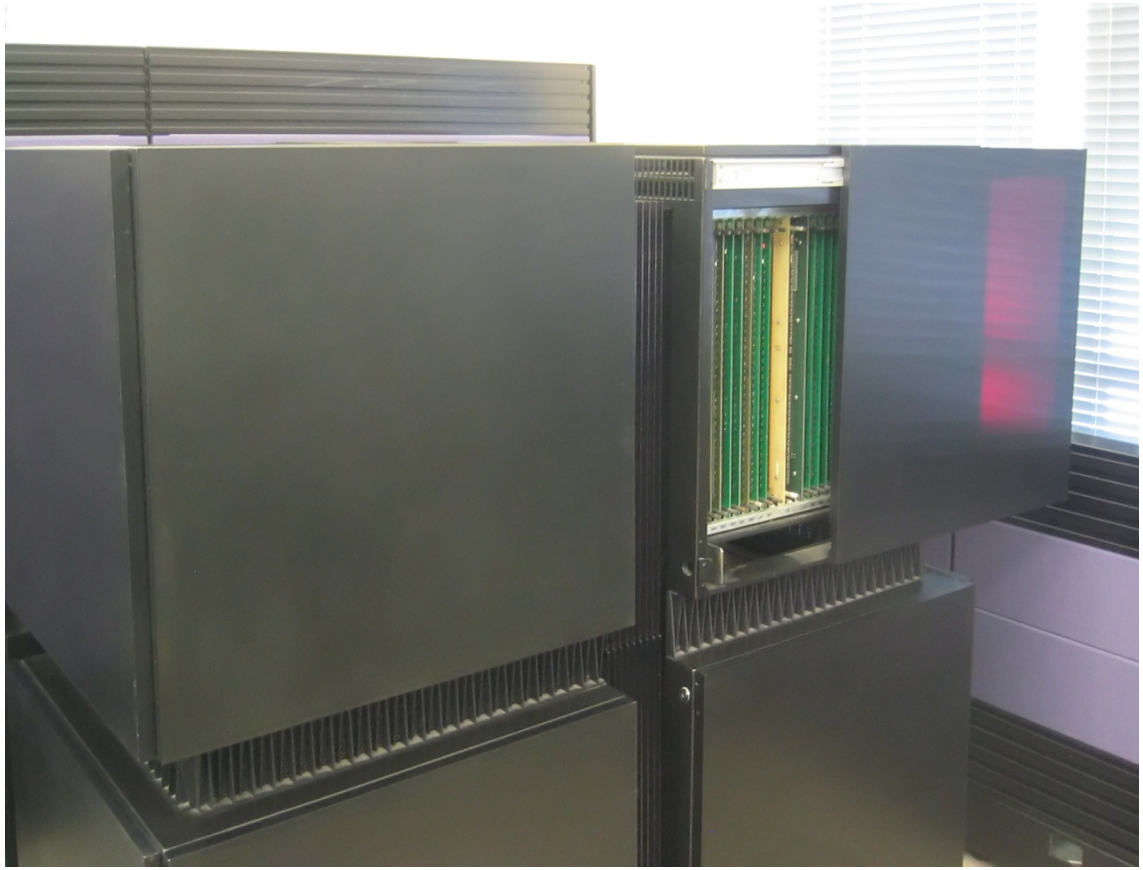
The early days of supercomputers

- From the mid 1970s to the mid 1980s, vector computers were the only supercomputers.
- Made by Cray Research. Competitors from Texas Instruments, CDC, Fujitsu, Hitachi, NEC, DEC, IBM.
- A vector machine had vector registers, vector load, store, and arithmetic operations; like SSE instructions in processors today
- A form of SIMD
- Arithmetic units were pipelined

A revolution in the 1980s

- Cray was highly successful – darling of Wall Street
- Imitators. Budget Crays
- The IBM and Apple PCs appeared
- One Cray-1 > 300 PCs in performance
- But lots of startups saw the coming of Moore's Law
- Ncube, Sequent, Cydrome, Multiflow, Saxpy, ASC, Allliant, Convex, KSR, TMC, Maspar, Celerity, Meiko
- Many different, proprietary, incompatible programming languages

The Connection Machines



Thinking Machines and HPF

- I want to build a machine who thinks
  -- Danny Hillis
- SIMD (CM-2) -- 1987
- MIMD (CM-5) - 1991
- CM Fortran -- Data parallel
  - A = B + C for whole arrays
  - One thread of control
  - Compiled to the MIMD CM-5
- An effort to standardize this way to program highly parallel machines:
  High Performance Fortran

Message Passing

- Inspired by the effort to standardize HPF, the message passing community convened the Message Passing Interface Standard effort (1991)
- Lots of trips to a boring hotel in Dallas
- First standard adopted in June 1994

The MPI Library

- A standard message passing library
- Created by a committee in early 1990s
- Used very widely for portable distributed memory programming
- High quality implementations, open source
- MPI-2 is current version
- MPI-3 under development
- Full featured; hundreds of routines
- A minimal subset, easy to use and understand

What other programming languages are there for clusters?

- OpenMP
  - Directives (comments) to create threads, say that loops are parallel
  - Mainly now in use for multicore processors
  - Compatible with MPI
- Unified Parallel C
  - C
  - SPMD
  - Shared arrays, with distributions as in HPF
  - A shared-memory model – locks, barriers, other synchronization is required

More alternatives

- Linda
  - Galernter and Carriero
  - SPMD, but the processes share a “tuple space”
    - Out(1, “mydata”, A) puts a tuple into the space
    - Read(1, ?, ?) gets a tuple from the space
    - In gets a tuple and removes it from the tuple space
    - Eval spawns a parallel computation

Beyond simple MPI

- MPI-2
  - Parallel file accesses
  - MPI_PUT, MPI_GET – allow one process to read or write into the memory of another
    - (How is this done without hardware support?)
  - Can change numprocs during the computation
- MPI-3 Under development