An Intro to Distributed Memory Computing and MPI

Jack Poulson, Rob Schreiber
Stanford ICME

Class 2

Why tags?

- A tag is a message type
- MPI_ANY_TAG matches any tag
- Specific tag matches only identical tag

When a receiver isn't sure of source, tag, count

```
MPI_Status status;
MPI_Recv( ..., &status );
... status.MPI_TAG;
... status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &count );
```

status.MPI_TAG and status.MPI_SOURCE used with MPI_ANY_TAG and/or MPI_ANY_SOURCE in the receive.

MPI_Get_count -- the actual number received.

What does MPI_Send really do?

```
MPI_Send(&mybuffer, numitems, …);
for (i = 0;   i < numitems;  i++) mybuffer[i] = 0;
```

This is safe.

mybuffer has been emptied out by MPI_send

Has the message arrived?   Maybe, maybe not.

Recv

```
MPI_recv(&b, numitems, ...)
for (i = 0; i < actlen; i++) x += *b++;
```

- This is also correct.
- Send and Recv are “blocking” until the local buffer is empty (send) or full (recv).

Talkin bout Send and Recv

- MPI guarantees the message is delivered.
- Messages are delivered “in the order sent”.

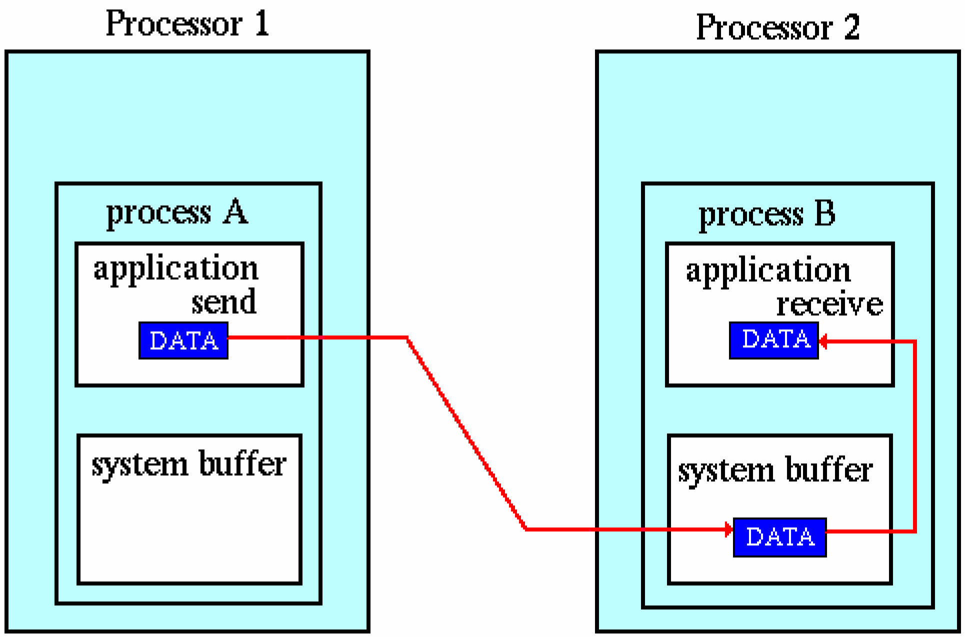
- How does MPI do this?
  - What if the MPI_recv is delayed?
  - Where would the data be placed if it starts to arrive before the Recv?

Buffering by MPI

- MPI tries to return control to the sender quickly.
  - Short messages -- sends them immediately, or copies into a buffer at the sender
  - Sender may block if the buffer is not available.
  - Receiver looks for a matching receive and tries to copy the data directly to the calling program's receive buffer.
  - Receiver may buffer a short message if no receive yet.

- Long messages: Too big to buffer
  - – MPI sends a short request to send
  - – Sender then blocks.

One possible implementation, buffer at destination



Path of a message buffered at the receiving process

A new kind of bug!

Sender may block until something happens at the receiver!

```
if (me%2 == 0) other = me+1; else other = me-1;
MPI_Send(&sendbuf, n, MPI_INT, other, ...
MPI_Recv(&recvbuf, n, MPI_INT, other, ...
```

Deadlock.

Another kind -- a performance bug

- Processes send to me+1, receive from me-1, in parallel

```
if (myid < nprocs-1)
    MPI_send(&x, items, MPI_INT, myid+1, …
if (myid > 0) recv(&y, 100000, myid-1);
     MPI_Recv(&y, items, MPI_INT, myid-1, …
```

Asynchronous Send and Recv

int **MPI_Isend (*buf, count, MPI_Datatype, dest, tag, comm, MPI_Request *request)**

int **MPI_Irecv (*buf, count, MPI_Datatype, source, tag, comm, MPI_Request *request)**

-- Returns without removing the data from buf.

-- Caller may not overwrite buf until:

int **MPI_Wait(*request, *status)**

which blocks till the buffer is full (Irecv) or empty (Isend)

Safety from deadlock

- Experienced hands use Isend, Irecv, Wait
- Waitall, Waitsome, Waitany, all available using a vector of requests.
- Deadlock preventing and performance improving
- Overlap communication with other work.

De-Sequentialization

```
if (myid > 0) mpi_irecv(rbuf, count, myid-1, &reqr);
/* maybe use a barrier here? */
if (myid < nprocs-1) mpi_isend(sbuf, count, myid+1,
    &reqs);
if (myid > 0) wait(reqr);
/* Now data have arrived in rbuf */
if (myid < nprocs-1) wait(reqs);
/* Now data have left sbuf */
```

Alternative sends

MPI provides mulitple modes for sending messages:

- MPI_Ssend: the send does not complete until a matching receive has begun.
- MPI Bsend: the caller supplies the buffer to system for its use.
- MPI Rsend: the programmer guarantees that a matching receive has been posted.

Performance in MPI

Parallel programs should be faster than sequential programs

time(nprocs)

speedup = -------------------------------

time(1 proc)

Performance in MPI

Some parallel programs do not get any speedup.

Some get “linear” speedup. Speedup proportional to nprocs

Some can get superlinear speedup, due to the additional cache.

How to get good speedup

1. Avoid having all processes wait while one of them does some work -- avoid sequential bottlenecks.

Amdahl

- Gene Amdahl (builder of big business machines): Parallel machines won't work
- "Amdahl's Law"
- But he was wrong: today, world's most powerful machine has over 500,000 processor cores. the second most has 1.5 million.
- One Job / Job One for computational scientists -- get the sequential out of the algorithm.

Distribute the work evenly

The most heavily loaded process will determine the compute time.

Redistribute data, and work, if necessary.

But the biggest issue is …

Communication. Messages cost:

Memory (buffering)

Processing (in the MPI library code)

Network hardware resources (wires, transceivers, buffers, swithes)

Simplest model I: The time between send and completion of receive is

$$T_{\mathrm{comm}}(n) = \alpha + \beta\, n \ (seconds)$$

for an n byte message. Normally $\alpha >> \beta$.

Moral of that story

Send fewer messages. Combine data into a longer message.

Send less data.

Requires careful thought about the assignment of data and work to processes.

More about communication cost

Communication must be load balanced too.

A processor has limited bandwidth into the network.
Probably β bytes/sec.

If many send to one, the messages queue up:

```
mpi_isend(x, n, MPI_CHAR, 0, …)
if (me == 0) for(i = 0; i < nprocs; i++)
   mpi_recv(y, n, MPI_CHAR, i, …)
```

This will take at least $\alpha + \beta\ n\ nprocs$, and maybe $(\ \alpha + \beta\ n\ )\ nprocs$.

Problem to ponder

- Suppose each process must send n bytes to every other process; assume it’s the same n bytes to each destination, but each sender has its own data to send.

- What is a bad way to do this, that creates traffic jams?

- Find a way to avoid bottlenecks.