• HW 2 is up, due Mar 11 (overlap).
• HW 3 will go out start of next week.
Previously on Parallel Programming

Can write a lot of MPI code with 6 operations we’ve seen:

- MPI_Init
- MPI_Finalize
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Send
- MPI_Recv

... but there are sometimes better ways. Decide on communication style using simple performance models.
Communication performance

• Basic info: latency and bandwidth
• Simplest model: \( t_{\text{comm}} = \alpha + \beta M \)
• More realistic: distinguish CPU overhead from “gap” (\( \sim \) inverse bw)
• Different networks have different parameters
• Can tell a lot via a simple ping-pong experiment
Intel MPI on totient

• Two six-core chips per nodes, eight nodes
• Heterogeneous network:
  • Crossbar switch between cores (?)
  • Bus between chips
  • Gigabit ethernet between nodes
• Default process layout (16 process example)
  • Processes 0-5 on first chip, first node
  • Processes 6-11 on second chip, first node
  • Processes 12-17 on first chip, second node
  • Processes 18-23 on second chip, second node
• Test ping-pong from 0 to 1, 11, and 23.
Approximate $\alpha$-$\beta$ parameters (on chip)

\[ \alpha \approx 3.74 \times 10^{-7}, \beta \approx 1.77 \times 10^{-10} \]
Approximate $\alpha$-$\beta$ parameters (cross-chip)

\[\alpha \approx 7.63 \times 10^{-7}, \beta \approx 3.53 \times 10^{-10}\]
Approximate $\alpha$-$\beta$ parameters (cross-node)

$\alpha \approx 3.54 \times 10^{-5}$, $\beta \approx 3.66 \times 10^{-9}$
Not all links are created equal!

• Might handle with mixed paradigm
  • OpenMP on node, MPI across
  • Have to worry about thread-safety of MPI calls

• Can handle purely within MPI
• Can ignore the issue completely?

For today, we’ll take the last approach.
Reminder: basic send and recv

MPI_Send(buf, count, datatype, dest, tag, comm);

MPI_Recv(buf, count, datatype, source, tag, comm, status);

MPI_Send and MPI_Recv are blocking

• Send does not return until data is in system
• Recv does not return until data is ready
Blocking and buffering

Block until data “in system” — maybe in a buffer?
Blocking and buffering

Alternative: don’t copy, block until done.
Both processors wait to finish send before they can receive!
May not happen if lots of buffering on both sides.
Solution 1: Alternating order

Could alternate who sends and who receives.
Solution 2: Combined send/recv

Common operations deserve explicit support!
Combined sendrecv

MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,
recvbuf, recvcount, recvtype, source, recvtag,
comm, status);

Blocking operation, combines send and recv to avoid deadlock.
Problem 2: Communication overhead

Partial solution: nonblocking communication
Blocking vs non-blocking communication

• **MPI_Send** and **MPI_Recv** are *blocking*
  • Send does not return until data is in system
  • Recv does not return until data is ready
  • Cons: possible deadlock, time wasted waiting

• Why blocking?
  • Overwrite buffer during send $\rightarrow$ evil!
  • Read buffer before data ready $\rightarrow$ evil!

• Alternative: *nonblocking* communication
  • Split into distinct initiation/completion phases
  • Initiate send/recv and promise not to touch buffer
  • Check later for operation completion
Overlap communication and computation

Start send
Start recv

End send
End recv

Start send
Start recv

End send
End recv

Compute, but don’t touch buffers
Nonblocking operations

Initiate message:

MPI_Isend(start, count, datatype, dest tag, comm, request);
MPI_Irecv(start, count, datatype, dest tag, comm, request);

Wait for message completion:

MPI_Wait(request, status);

Test for message completion:

MPI_Test(request, status)
Sometimes useful to have multiple outstanding messages:

```c
MPI_Waitall(count, requests, statuses);
MPI_Waitany(count, requests, index, status);
MPI_Waitsome(count, requests, indices, statuses);
```

Multiple versions of test as well.
Other variants of `MPI_Send`

- **MPI_Ssend** (synchronous) – do not complete until receive has begun
- **MPI_Bsend** (buffered) – user provides buffer (via `MPI_Buffer_attach`)
- **MPI_Rsend** (ready) – user guarantees receive has already been posted
- Can combine modes (e.g. `MPI_Issend`)

`MPI_Recv` receives anything.
Another approach

- Send/recv is one-to-one communication
- An alternative is one-to-many (and vice-versa):
  - Broadcast to distribute data from one process
  - Reduce to combine data from all processors
  - Operations are called by all processes in communicator
MPI_Bcast(buffer, count, datatype, root, comm);
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm);

- **buffer** is copied from root to others
- **recvbuf** receives result only at root
- **op** \(\in\) \{ MPI_MAX, MPI_SUM, ...\}
Example: basic Monte Carlo

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char** argv) {
    int nproc, myid, ntrials = atoi(argv[1]);
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    MPI_Bcast(&ntrials, 1, MPI_INT, 0, MPI_COMM_WORLD);
    run_trials(myid, nproc, ntrials);
    MPI_Finalize();
    return 0;
}
```
Example: basic Monte Carlo

Let $\text{sum}[0] = \sum_i X_i$ and $\text{sum}[1] = \sum_i X_i^2$.

```c
void run_mc(int myid, int nproc, int ntrials) {
    double sums[2] = {0,0};
    double my_sums[2] = {0,0};
    /* ... run ntrials local experiments ... */
    MPI_Reduce(my_sums, sums, 2, MPI_DOUBLE,
                MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) {
        int N = nproc*ntrials;
        double EX = sums[0]/N;
        double EX2 = sums[1]/N;
        printf("Mean: %g; err: %g\n",
                EX, sqrt((EX*EX-EX2)/N));
    }
}
```
Collective operations

- Involve all processes in communicator
- Basic classes:
  - Synchronization (e.g. barrier)
  - Data movement (e.g. broadcast)
  - Computation (e.g. reduce)
MPI_Barrier(comm);

Not much more to say. Not needed that often.
Broadcast

P0
A

P1

P2

P3

Bcast

P0
A

P1

P2

P3
A
Scatter/gather
Allgather

P0  | A
---|---
P1  | B
---|---
P2  | C
---|---
P3  | D

P0  | A
---|---
P1  | B
---|---
P2  | C
---|---
P3  | D

A  | B  | C  | D
---|---|---|---
A  | A  | A  | A
---|---|---|---
B  | B  | B  | B
---|---|---|---
C  | C  | C  | C
---|---|---|---
D  | D  | D  | D
---|---|---|---
Alltoall

P0: A0  A1  A2  A3
P1: B0  B1  B2  B3
P2: C0  C1  C2  C3
P3: D0  D1  D2  D3

P0: A0  B0  C0  D0
P1: A1  B1  C1  D1
P2: A2  B2  C2  D2
P3: A3  B3  C3  D3
Reduce

\[ P0 \quad A \]
\[ P1 \quad B \]
\[ P2 \quad C \]
\[ P3 \quad D \]

\[ \rightarrow \]

\[ P0 \quad A + B + C + D \]
\[ P1 \quad \]
\[ P2 \quad \]
\[ P3 \quad \]
Scan

P0  A
P1  B
P2  C
P3  D

P0  A
P1  A+B
P2  A+B+C
P3  A+B+C+D
• In addition to above, have vector variants (v suffix), more All variants (Allreduce), Reduce_scatter, ...
• MPI3 adds one-sided communication (put/get)
• MPI is not a small library!
• But a small number of calls goes a long way
  • Init/Finalize
  • Get_comm_rank, Get_comm_size
  • Send/Recv variants and Wait
  • Allreduce, Allgather, Bcast