Lecture 10: MPI continued

David Bindel

27 Feb 2014
Logistics

- 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
OpenMPI on crocus

- Two quad-core chips per nodes, five nodes
- Heterogeneous network:
  - Crossbar switch between cores (?)
  - Bus between chips
  - Gigabit ethernet between nodes
- Default process layout (16 process example)
  - Processes 0-3 on first chip, first node
  - Processes 4-7 on second chip, first node
  - Processes 8-11 on first chip, second node
  - Processes 12-15 on second chip, second node
- Test ping-pong from 0 to 1, 7, and 8.
Approximate $\alpha$-$\beta$ parameters (on node)

\[ \alpha_1 \approx 1.0 \times 10^{-6}, \beta_1 \approx 5.7 \times 10^{-10} \]

\[ \alpha_2 \approx 8.4 \times 10^{-7}, \beta_2 \approx 6.8 \times 10^{-10} \]
Approximate $\alpha - \beta$ parameters (cross-node)

$$\alpha_3 \approx 7.1 \times 10^{-5}, \beta_3 \approx 9.7 \times 10^{-9}$$
Moral

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.
Problem 1: Potential deadlock

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 $\implies$ evil!
  - Read buffer before data ready $\implies$ 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
Start send
Start recv
Compute, but don’t touch buffers!
End send
End recv
End send
End recv
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);
Multiple outstanding requests

Sometimes useful to have multiple outstanding messages:

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

Multiple versions of test as well.
Other send/recv variants

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_Isend`)

`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
Broadcast and reduce

\[
\text{MPI\_Bcast(} \text{buffer, count, datatype, root, comm)}; \\
\text{MPI\_Reduce(} \text{sendbuf, recvbuf, count, datatype, op, root, comm)};
\]

- \text{buffer} \text{ is copied from root to others}
- \text{recvbuf} \text{ receives result only at root}
- \text{op} \in \{ \text{MPI\_MAX, MPI\_SUM, ...} \}
Example: basic Monte Carlo

```c
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv) {
    int nproc, myid, ntrials;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    if (myid == 0) {
        printf("Trials per CPU:\n");
        scanf("%d", &ntrials);
    }
    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)
Barrier

MPI_Barrier(comm);

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

P0  |  A  |
P1  |     |
P2  |     |
P3  |     |

P0  |  A  |
P1  |     |
P2  |     |
P3  |     |

Broadcast
Scatter/gather

<table>
<thead>
<tr>
<th>P0</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scatter: P0

<table>
<thead>
<tr>
<th>P0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
</tr>
</tbody>
</table>

Gather: P2

<table>
<thead>
<tr>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
</tr>
</tbody>
</table>
Allgather

P0
A

P1
B

P2
C

P3
D

Allgather

P0
A

P1
B

P2
C

P3
D
Alltoall

P0

A0 | A1 | A2 | A3

P1

B0 | B1 | B2 | B3

P2

C0 | C1 | C2 | C3

P3

D0 | D1 | D2 | D3

Alltoall

P0

A0 | B0 | C0 | D0

P1

A1 | B1 | C1 | D1

P2

A2 | B2 | C2 | D2

P3

A3 | B3 | C3 | D3
Reduce

P0
A

P1
B

P2
C

P3
D

Reduce

P0
ABCD

P1

P2

P3
Scan

```
P0: A
P1: B
P2: C
P3: D

P0: A
P1: AB
P2: ABC
P3: ABCD
```
The kitchen sink

- 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