Basic Communication in MPI

CME342 / AA220
Lecture 4
April 9, 2014
Announcements

- Class email list: go to lists.stanford.edu → Manage Subscriptions section → search for the 'cme342-class' subscriber page → enter your info and hit 'subscribe'

- Accounts for icme-mpi1.stanford.edu have been set up. Please let Tom know if you have any trouble logging in. Demo today.

- Homework #1 on class web page (http://adl.stanford.edu/cme342/Assignments.html). Due April 21, 5pm, Durand 252 (or in class).

- Tom’s Office Hours: 11:00-12:15 on Friday in Durand 393 (this week)
Outline

- Review of point-to-point communicators.
- Review of collective communicators.
- *data ring* example.
- Latency and bandwidth.
- *bounce* example.
Send & Receive

- Cooperative data transfer
- To (from) whom is data sent (received)?
- What is sent?
- How does the receiver identify it?
Single Program Multiple Data (SPMD)

• Proc 0 & Proc 1 are actually performing different operations.
• However, not necessary to write separate programs for each processor.
• Typically, use conditional statement and proc id to define the job of each processor:

```c
int a[10];

if (my_id == 0)
    MPI_Send (a,10,MPI_INT,1,0,MPI_COMM_WORLD);
else if (my_id == 1)
    MPI_Recv (a,10,MPI_INT,0,0,MPI_COMM_WORLD);
endif
```
Deadlock

- Example: exchange data between 2 procs:

  - MPI_Send is a synchronous operation. If no system buffering, it keeps waiting until a matching receive is posted.
  - Both processors are waiting for each other → deadlock.
• However, OK if system buffering exists $\rightarrow$ unsafe program.

• Note: MPI\_Recv is blocking and nonbuffered.

• A real deadlock:

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Recv</td>
<td>MPI_Recv</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>MPI_Send</td>
</tr>
</tbody>
</table>

• Fix by reordering comm.:

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send</td>
<td>MPI_Recv</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>MPI_Send</td>
</tr>
</tbody>
</table>
Buffered / Nonbuffered Comm.

• No-buffering (*phone calls*)
  ▶ Proc 0 initiates the send request and rings Proc 1. It *waits* until Proc 1 is ready to receive. The transmission starts.
  ▶ Synchronous comm. — completed only when the message was received by the receiving proc.

• Buffering (*text message*)
  ▶ The message to be sent (by Proc 0) is copied to a system-controlled block of memory (buffer).
  ▶ Proc 0 can continue executing the rest of its program.
  ▶ When Proc 1 is ready to receive the message, the system copies the buffered message to Proc 1.
  ▶ Asynchronous comm. — may be completed even though the receiving proc has not received the message.
Buffering requires system resources, e.g. memory, and can be slower if the receiving proc is ready at the time of requesting the send.

Application buffer: address space that holds the data.

System buffer: system space for storing messages. In buffered comm., data in application buffer is copied to/from system buffer.

MPI allows comm. in buffered mode: MPI_Bsend(), MPI_Ibsend().

User allocates the buffer by: MPI_Buffer_attach(buffer, buffer_size)

Free the buffer by MPI_Buffer_detach.
Blocking / Nonblocking Comm.

- Blocking Comm. (*Jack in the Box Drive-Thru*)
  - The receiving proc has to wait if the message is not ready.
  - Different from synchronous comm.
  - Proc 0 may have already buffered the message to system and Proc 1 is ready, but the interconnection network is busy.

- Nonblocking Comm. (*Jack in the Box Dining Room*)
  - Proc 1 checks with the system if the message has arrived yet. If not, it continues doing other stuff. Otherwise, get the message from the system.

- Useful when computation and comm. can be performed at the same time.
• MPI allows both nonblocking send & receive: 
  MPI_Isend(), MPI_Irecv().

• In nonblocking send, program identifies an area in memory 
to serve as a send buffer. Processing continues immediately 
without waiting for message to be copied out from application 
buffer.

• The program **should not** modify the application buffer until 
  the nonblocking send has completed.

• Nonblocking comm. can combined with nonbuffering: 
  MPI_Issend(), or buffering: MPI_Ibsend().

• Use MPI_Wait() or MPI_Test() to determine if the 
  nonblocking send or receive has completed.
Summary: Communication Modes

- 4 comm. modes in MPI: standard, buffered, synchronous, ready. They can be either blocking or nonblocking.

- In standard modes (MPI_Send, MPI_Recv, ...), it is up to the system to decide whether messages should be buffered.

- In synch. mode, a send won’t complete until a matching receive has been posted which has begun reception of the data.
  - MPI_Ssend, MPI_Issend.
  - No system buffering.

- In buffered mode, the completion of a send does not depend on the existence of a matching receive.
▷ MPI_Bsend, MPI_Ibsend.
▷ System buffering by MPI_Buffer_attach & MPI_Buffer_detach.

- Ready mode not discussed. Use only if the programmer can be sure a receive has been posted first.

- Blocking means that we are posting the send/recv and waiting for our application buffer (address in our local memory) to be available for modification again.

- Non-blocking means that we only post the send/recv and carry on immediately. We can not modify the address that we specified until verifying that the communication has been completed later.

- Lastly, there is also a combined blocking MPI_Sendrecv().
Summary: Point-to-Point Communicators

- Our common communicators...

<table>
<thead>
<tr>
<th></th>
<th>Blocking</th>
<th>Non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Send</td>
<td>MPI_Send()</td>
<td>MPI_Isend()</td>
</tr>
<tr>
<td>Synchronous Send</td>
<td>MPI_Ssend()</td>
<td>MPI_Isend()</td>
</tr>
<tr>
<td>Buffered Send</td>
<td>MPI_Bsend()</td>
<td>MPI_Ibsend()</td>
</tr>
<tr>
<td>Ready Send</td>
<td>MPI_Rsend()</td>
<td>MPI_Irsend()</td>
</tr>
<tr>
<td>Receive</td>
<td>MPI_Recv()</td>
<td>MPI_Irecv()</td>
</tr>
</tbody>
</table>

- Can you explain the differences in each of the above?
- Don’t forget about other MPI web resources linked from http://adl.stanford.edu/cme342/Lecture_Notes.html.
- Homework #1 will give you a chance to exercise these.
Collective Communication

- Comm. involving all the procs in a single comm. group.
- MPI_Barrier: synchronize all procs.

- **Broadcast (MPI_Bcast)**
  - A single proc sends the same data to every proc.

- **Reduction (MPI_Reduce)**
  - All the procs contribute data that is combined using a binary operation, e.g., max, min, sum, etc.
  - One proc obtains the final answer.

- **Allreduce (MPI_Allreduce)**
  - Same as MPI_Reduce but every proc contains final answer.
  - Effectively MPI_Reduce + MPI_Bcast, but more efficient.
Other Collective Comm.

- **Scatter** (MPI_Scatter)
  - Split the data on proc *root* into *p* segments.
  - The 1st segment is sent to proc 0, the 2nd to proc 1, etc.
  - Similar to but more general than MPI_Bcast.

- **Gather** (MPI_Gather)
  - Collect the data from each proc and store the data on proc *root*.
  - Similar to but more general than MPI_Reduce.

- Can collect and store the data on *all* procs using MPI_Allgather.
Comparisons of Collective Comms.

- **Broadcast**:
  - $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$
  - Initial: $A \rightarrow A \rightarrow A \rightarrow A$

- **Scatter**:
  - $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$
  - Initial: $A \rightarrow B \rightarrow C \rightarrow D$

- **Gather**:
  - $P_1 \leftarrow P_2 \leftarrow P_3 \leftarrow P_4$
  - Initial: $A \leftarrow B \leftarrow C \leftarrow D$

- **All gather**:
  - $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$
  - Initial: $A \rightarrow B \rightarrow C \rightarrow D$

- **All to all**:
  - $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$
  - Initial: $A \rightarrow B \rightarrow C \rightarrow D$
Compiling MPI Programs on
icme-mpi1.stanford.edu

• All of the SUnetIDs now have access to the ICME cluster resources.

• Head node of the cluster at icme-mpi1.stanford.edu.

• 23 compute nodes:
  ▶ 8 cores each (Dual quad-core Intel Xeon X5667 @ 3.07 GHz)
  ▶ 8 GB memory

• http://icme.stanford.edu/computer-resources/gpu-cluster

• Some of the following slides are adapted from the CME 212 notes (thanks Patrick LeGresley!)
Head Node

• Used for login, editing files, compiling, testing, debugging, etc.

• Also runs the cluster wide file system (compute nodes see the shared space).

• Do not run compute/memory intensive jobs on the head node.

• Run your scaling tests on the compute nodes (not the head node), to avoid performance degradation.
Compute Nodes

- Run jobs on the compute nodes through a resource manager.
- In our case on icme-mpi1.stanford.edu, it is managed by Torque/PBS (qsub).
- Manager will prioritize and schedule jobs.
- Once the resources are available, it allocates a dedicated number of nodes to your job.
Basic Cluster Usage

Login to the head node with your SUNetID and password:

```
ssh sunetid@icme-mpi1.stanford.edu.
```

Compilers are available as modules. Use the following commands:

```
[economon@icmemaster1 ~]$ module avail
[economon@icmemaster1 ~]$ module add openmpi/gcc/64/1.6.5
[economon@icmemaster1 ~]$ module list
Currently Loaded Modulefiles:
  1) gcc/4.8.1
  2) torque/4.2.2
  3) openmpi/gcc/64/1.6.5
```

I have tested gcc and OpenMPI successfully. There are multiple compiler/MPI options available as modules for C/C++ or Fortran 90/95. You can add the module add commands to your ‘.bashrc’ files to have it done on login. Let us know if you have any issues.
Basic Cluster Usage

Assuming that you have the hello_world.c program in your home directory, you would compile it with:

```
mpicc -o hello_world hello_world.c
```

In order to run 8 copies of this program, simply type:

```
mpirun -np 8 ./hello_world
```

8 copies of your hello world executable will be launched onto the head node of the cluster (which itself has 24 cores) and will be executed within the same MPI session.
Basic Cluster Usage

Compute nodes can be requested in interactive mode:

[economon@icmemaster1 ~]$ qsub -I -V -l nodes=1:ppn=8
qsub: waiting for job 1841.icmemaster1.cm.cluster to start
qsub: job 1841.icmemaster1.cm.cluster ready
[economon@node010 ~]$ 

- Note that we’re now logged into a specific node (#10).
- We have dedicated access to the 8 cores on node # 10.
- -l → requests interactive mode.
- -V → passes environment variables on to session.
- -l nodes=1:ppn=8 → requests 8 cores on 1 node.
- End your interactive session by typing ‘logout’.
Basic Cluster Usage

Non-interactive jobs can be submitted using a job script. An example job script will be provided for the Hello, World! code:

```bash
#!/bin/bash

# Name for your job
#PBS -N hello_world

# Names for PBS error and log files
#PBS -e hello_world.err
#PBS -o hello_world.out

# Request resources in terms of cores
#PBS -l nodes=1:ppn=8
```
# Set max wall time (hh:mm:ss)
#PBS -l walltime=00:05:00

# Pass along your environment variables
#PBS -V

# Go to the directory where qsub command was given
cd ${PBS_O_WORKDIR}

# Call the executable. The output of our job
# will be in the log file specified above
mpirun -np 8 ./hello_world

• Note that, while text after # with a space is a comment, #PBS lines specify job settings
Basic Cluster Usage

To submit your job, use `qsub`:

```
[economon@icmemaster1 hello_world]$ qsub submit.sh
1844.icmemaster1.cm.cluster
[economon@icmemaster1 hello_world]$
```

- Jobs will be prioritized based on cores and wall time requested
- Some useful commands for monitoring jobs:
  - `showq`: show the resources and jobs across the machine
  - `qstat`: show status of your own jobs
  - `qdel job_id#`: delete a queued or running job
- Based on the ‘submit.sh’ script above for hello_world:
  - Check ‘hello_world.err’ for failure messages
  - ‘hello_world.out’ will contain the console output
Example: Data Exchange in a Ring Topology

- Blocking version:
  ```c
  for (i = 0; i < p; i++) {
    send_offset = ((my_id-i+p) % p)*blksize;
    recv_offset = ((my_id-i-1+p) % p)*blksize;
    MPI_Send(y+send_offset, blksize, MPI_INT, (my_id+1)%p, 0, MPI_COMM_WORLD);
    MPI_Recv(y+recv_offset, blksize, MPI_INT, (my_id-1)%p, 0, MPI_COMM_WORLD, &status);
  }
  ```

- Nonblocking version:
send_offset = my_id*blksize;
recv_offset = (my_id-1+p)*blksize;
for (i = 0; i < p; i++) {
  MPI_Isend(y+send_offset, blksize, MPI_INT,
            (my_id+1)%p, 0, MPI_COMM_WORLD, &send_request);
  MPI_Irecv(y+recv_offset, blksize, MPI_INT,
            (my_id-1)%p, 0, MPI_COMM_WORLD, &recv_request);
  send_offset = ((my_id-i+p) % p)*blksize;
  recv_offset = ((my_id-i-1+p) % p)*blksize;
  MPI_Wait(&send_request, &status);
  MPI_Wait(&recv_request, &status);
}

• The comm. & computations of next offsets are overlapped.

• What happens if we use MPI_Ssend() in the blocking version?
Definitions of Latency and Bandwidth

- **LATENCY** is the amount of wall clock time that it takes for a message of zero length to be sent from one processor to another.

- **BANDWIDTH** is the number of bytes/sec that can be sent from one processor to another.

- Latency penalizes programs that send/receive a large number of messages.

- Low bandwidth penalizes programs that send/receive a large amount of information.

- Ideal network has low latency and high bandwidth and is VERY expensive.
1. Start an even number of instances of the executable.

2. Pair them up (next and previous in a linear array).

3. Send/receive a series of messages of increasing size and record the time that it takes to send/receive each message.

4. Repeat a number of times and average results.

5. Report *bandwidth* and *latency*.
Timing

- MPI_Wtime() returns the wall-clock time in seconds.
- This is useful for scalability tests (HW #1).

```c
double start, finish, time;

MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
  :
  :
MPI_Barrier(MPI_COMM_WORLD);
finish = MPI_Wtime();
time = finish - start;
```
program bounce
parameter (maxcount=1000000)
parameter (nrepeats=50)
parameter (nsizes=7)

implicit real*8 (a-h,o-z)
include "mpif.h"

dimension sbuf(maxcount), rbuf(maxcount)
dimension length(nsizes)
integer status(mpi_status_size)
! define an array of message lengths
!---------------------------------------
!
  length(1) = 0
  length(2) = 1
  length(2) = 10
  length(3) = 100
  length(4) = 1000
  length(5) = 10000
  length(6) = 100000
  length(7) = 1000000
!
!---------------------------------------
! initialize the send buffer to zero
!---------------------------------------
!
  do n=1,maxcount
    sbuf(n) = 0.0d0
    rbuf(n) = 0.0d0
  end do
!
!---------------------------------------
! set up the parallel environment
!---------------------------------------
!
  call mpi_init(ierr)
  call mpi_comm_size(mpi_comm_world,nnodes,ierr)
  call mpi_comm_rank(mpi_comm_world,nodeid,ierr)
if (nodeid.eq.0) write(*,*)'number of processors =',nnodes
!
signal error if an odd number of nodes is specified
!
if (mod(nnodes,2) .ne. 0) then
  if (nodeid .eq. 0) then
    write(6,*)'you must specify an even number of nodes.'
  end if
  call mpi_finalize(ierr)
end if
!
!---------------------------------------------------------
! send or receive messages, and time it.
! even nodes send, odd nodes receive, then the reverse
!---------------------------------------------------------
!
do ns=1, nsizes
  time1 = mpi_wtime()
  do nr=1, nrepeats
  !
  !---------------------------------------------------------
  ! send in one direction i->i+1
  !---------------------------------------------------------
  !
  if (mod(nodeid,2) .eq. 0) then
    call mpi_send(sbuf, length(ns), mpi_real8, nodeid+1, 1, &
                  mpi_comm_world, ierr)
  else
call mpi_recv(rbuf, length(ns), mpi_real8, nodeid-1, 1, &
    mpi_comm_world, status, ierr)
end if

!---------------------------------------------------
! send in the reverse direction i+1->i
!---------------------------------------------------
!
if (mod(nodeid,2) .eq. 1) then
  call mpi_send(sbuf, length(ns), mpi_real8, nodeid-1, 1, &
    mpi_comm_world, ierr)
else
  call mpi_recv(rbuf, length(ns), mpi_real8, nodeid+1, 1, &
    mpi_comm_world, status, ierr)
end if
end do
time2 = mpi_wtime()

!---------------------------------------------------
! timings and report results
!---------------------------------------------------
!
if (nodeid .eq. 0) then
  write(6,fmt='(a,i9,a,10x,a,f10.4,a)') 'msglen =',8*length(ns),&
    ' bytes,' , 'elapsed time =',0.5d3*(time2-time1)/nrepeats,' msec'
call flush(6)
end if
if (ns .eq. 1) then
  tlatency = 0.5d6*(time2-time1)/nrepeats
end if
if (ns .eq. nsizes) then
  bw = 8.*length(ns)/(0.5d6*(time2-time1)/nrepeats)
end if
end do
!
!---------------------------------------------------------
! report approximate numbers for bandwidth and latency
!---------------------------------------------------------
!
if (nodeid .eq. 0) then
  write(6,fmt='(a,f6.1,a)') 'latency =',tlatency,' microseconds'
  write(6,*) 'bandwidth =',bw,' mbytes/sec'
  write(6,fmt='(a)') '(approximate values for mp_bsend/mp_brecv)'
end if

! call mpi_finalize(ierr)
end
Output of bounce on junior.stanford.edu

With ANL MPICH implementation:

junior:~/bounce /afs/ir/class/cs238/mpich/bin/mpif90 -o bounce bounce.f
junior:~/bounce /afs/ir/class/cs238/mpich/bin/mpirun -np 8 ./bounce
Number of processors = 8
msglen = 0 bytes, elapsed time = 0.1490 msec
msglen = 80 bytes, elapsed time = 0.1496 msec
msglen = 800 bytes, elapsed time = 0.1802 msec
msglen = 8000 bytes, elapsed time = 0.3409 msec
msglen = 80000 bytes, elapsed time = 1.4261 msec
msglen = 800000 bytes, elapsed time = 13.9892 msec
msglen = 8000000 bytes, elapsed time = 137.0006 msec
latency = 149.0 microseconds
bandwidth = 58.39391336883627 MBytes/sec
(approximate values for mp_bsend/mp_brecv)
With SUNWhpc MPI implementation:

```
junior:~/bounce /opt/SUNWhpc/bin/mpf90 -o bounce bounce.f -lmpi
junior:~/bounce /opt/SUNWhpc/bin/mprun -np 8 ./bounce
```

Number of processors = 8

<table>
<thead>
<tr>
<th>msglen</th>
<th>elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 bytes</td>
<td>0.0066 msec</td>
</tr>
<tr>
<td>80 bytes</td>
<td>0.0102 msec</td>
</tr>
<tr>
<td>800 bytes</td>
<td>0.0606 msec</td>
</tr>
<tr>
<td>8000 bytes</td>
<td>0.0781 msec</td>
</tr>
<tr>
<td>80000 bytes</td>
<td>0.5065 msec</td>
</tr>
<tr>
<td>800000 bytes</td>
<td>4.7291 msec</td>
</tr>
<tr>
<td>8000000 bytes</td>
<td>46.0369 msec</td>
</tr>
</tbody>
</table>

latency = 6.6 microseconds

bandwidth = 173.77383012979993 MBytes/sec

(approximate values for mp_bsend/mp_brecv)
Output of bounce on calving.stanford.edu

[jjalonso@calving ~]$ /opt/mpich/intel/bin/mpif90 -o bounce bounce.f
[jjalonso@compute-0-7 ~]$ /opt/mpich/intel/bin/mpirun -nolocal ...
-machinemfile /opt/torque/aux/1883.calving.stanford.edu -np 8 ./bounce
Running executable bounce
Number of processors = 8
msglen = 0 bytes, elapsed time = 0.0000 msec
msglen = 80 bytes, elapsed time = 0.0000 msec
msglen = 800 bytes, elapsed time = 0.1953 msec
msglen = 8000 bytes, elapsed time = 0.0000 msec
msglen = 80000 bytes, elapsed time = 0.1953 msec
msglen = 800000 bytes, elapsed time = 2.9297 msec
msglen = 8000000 bytes, elapsed time = 30.2734 msec
latency = 0.0 microseconds
bandwidth = 264.258064516129 MBytes/sec
(approximate values for mp_bsend/mp_brecv)
With Myrinet on Beowulf cluster:

n8 3% mpiexec bounce_myr
   Number of processors = 8
msglen = 0 bytes, elapsed time = 0.0117 msec
msglen = 80 bytes, elapsed time = 0.0149 msec
msglen = 800 bytes, elapsed time = 0.0488 msec
msglen = 8000 bytes, elapsed time = 0.2376 msec
msglen = 80000 bytes, elapsed time = 1.2652 msec
msglen = 800000 bytes, elapsed time = 12.3286 msec
msglen = 8000000 bytes, elapsed time = 122.5315 msec
latency = 11.7 microseconds
bandwidth = 65.28934971611689 MBytes/sec
(approximate values for mp_bsend/mp_brecv)
With 100BT Switched Ethernet on Beowulf cluster:

```
n8 3% /usr/local/mpich-1.2.0/bin/mpirun -np 8 ./bounce_eth
Number of processors = 8
msglen = 0 bytes, elapsed time = 0.0942 msec
msglen = 80 bytes, elapsed time = 0.0991 msec
msglen = 800 bytes, elapsed time = 0.2459 msec
msglen = 8000 bytes, elapsed time = 0.9316 msec
msglen = 80000 bytes, elapsed time = 7.1869 msec
msglen = 800000 bytes, elapsed time = 71.6712 msec
msglen = 8000000 bytes, elapsed time = 712.2905 msec
latency = 94.2 microseconds
bandwidth = 11.23137338146257 MBytes/sec
(approximate values for mp_bsend/mp_brecv)
```
Using GNU Fortran (GCC) 4.8.1 & OpenMPI 1.6.5 across 3 nodes with 8 cores each (2014):

- number of processors = 24
- msglen = 0 bytes, elapsed time = 0.0026 msec
- msglen = 80 bytes, elapsed time = 0.0017 msec
- msglen = 800 bytes, elapsed time = 0.0029 msec
- msglen = 8000 bytes, elapsed time = 0.0104 msec
- msglen = 80000 bytes, elapsed time = 0.0557 msec
- msglen = 800000 bytes, elapsed time = 0.5147 msec
- msglen = 8000000 bytes, elapsed time = 5.8477 msec

Latency = 2.6 microseconds
- Bandwidth = 1368.0548584810849 mbytes/sec

(approximate values for mp_bsend/mp_brecv)