Introduction to parallel computing

Distributed Memory Programming with MPI (2)

Zhiao Shi (additions by Will French)
Advanced Computing Center for Education & Research
Vanderbilt University
Last Time

• **MPI API**
  • MPI_Init
  • MPI_Finalize
  • MPI_Send
  • MPI_Recv
  • MPI_Comm_size
  • MPI_Comm_rank
Message Matching

MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag, send_comm);

MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag, recv_comm, &status);

Process rank

MPI_COMM_WORLD

Process rank
A Note About Message Order

• What happens if a process sends multiple messages to another process in succession?
  • The MPI standard guarantees that one message will not overtake another.
  • The receiving process will get the messages in the order they were received.
Communication Overhead

• There are two main ways to “hide” overhead due to inter-process communication:
  • Buffering
    • MPI uses a “system buffer” for doing this automatically.
    • A user can also create and manage his/her own buffer space.
  • Interleaving communication with computation.
    • Non-blocking send/receive calls.
Buffering

Path of a message buffered at the receiving process
Buffering

• System Buffer
  • Block of memory where sent data can be stored so that the sending process can continue.
  • Enables message passing to be performed asynchronously.
  • Managed entirely by the MPI library.
  • Opaque to the programmer.

• Application Buffer
  • Variables where a programmer stores data that are being sent/received between processes.
Communication Modes

• **Standard** mode
  • Buffering is system dependent.

• **Buffered** mode
  • A buffer must be provided by the application.

• **Synchronous** mode
  • Completes only after a matching receive has been posted.

• **Ready** mode
  • May only be called when a matching receive has already been posted.
## Communication Modes

<table>
<thead>
<tr>
<th>Communication Mode</th>
<th>Blocking Routines</th>
<th>Non-Blocking Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td></td>
<td>MPI_Recv</td>
<td>MPI_Irecv</td>
</tr>
<tr>
<td>Buffered</td>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
</tr>
<tr>
<td>Synchronous</td>
<td>MPI_Ssend</td>
<td>MPI_Issend</td>
</tr>
<tr>
<td>Ready</td>
<td>MPI_Rsend</td>
<td>MPI_Irsend</td>
</tr>
</tbody>
</table>
Standard Mode

• Exact behavior is determined by the MPI implementation.
• MPI_Send may behave differently with regard to buffer size and blocking behavior.
  • Data generally buffered, at which point process may proceed to next instruction.
• MPI_Recv always blocks until a matching message is received.
Buffered Mode

- MPI_Bsend (buf, count, datatype, dest, tag, comm)
- MPI_Buffer_attach (buff, size)
- MPI_Buffer_detach (void * buff_addr, int * bufsize)

- Buffered sends do not rely on system buffers.
- The user supplies a buffer that **MUST be large enough** for the messages.
- Only one buffer is defined at any one time (for a given process).
- The user **MUST ensure there is no buffer overflow**.
Buffer Management

• The user must provide the buffer.
  • A statically allocated array, or dynamically allocated with `malloc` in C.
• One user-supplied message buffer active at a time.
  • It will store multiple messages.
  • The system keeps track of when messages ultimately leave the buffer, and will reuse buffer space.
• For a program to be safe, it should not depend on such reuse.
MPI_Bsend

MPI_BSEND (buffered send)

- copy data to buffer
- data transfer to user-supplied buffer complete

S

MPI_RECV

task wait

R
Synchronous Mode

• **MPI_Ssend** \((\text{buf}, \text{count}, \text{datatype}, \text{dest}, \text{tag}, \text{comm})\)

• “Handshake” process
  • Sender sends “ready to send” to receiver.
  • Receiver sends a "ready to receive" message when call MPI_Recv. The data are then transferred.

• Does not complete until a matching receive has been posted and the receive operation has been started.
  • Does NOT mean the matching receive has completed.

• Buffering can be avoided.
MPI_Ssend

**MPI_SSEND** (blocking synchronous send)

Task waits

Data transfer from source complete

Wait

**MPI_RECV** (receiving task waits until buffer is filled)
Synchronization Mode Overhead

• System overhead
  • Transferring the message data from the sender onto the network, and from transferring the message data from the network into the receiver.

• Synchronization overhead
  • The time spent waiting for an event to occur on another task.
  • (In previous slide) The sender must wait for the receive to be executed and for the handshake to arrive before the message can be transferred.
Ready Mode

- MPI_Rsend (buf, count, datatype, dest, tag, comm)
- May ONLY be started (called) if a matching receive has already been posted.
- If a matching receive has not been posted, the results are undefined (by default, it will exit).
- May be most efficient when appropriate.
  - Removal of handshake operation.
- Should only be used with **extreme caution**.
MPI_Rsend

**MPI_RSEND** (blocking ready send)

- Data transfer from source complete
- **MPI_RECV** (receiving task waits until buffer is filled)

S

R

wait
Ready Mode

• Aims to minimize system overhead and synchronization overhead incurred by the sending task.

• In the blocking case, the only wait on the sending node is until all data have been transferred out of the sending task's message buffer.

• The receive can still incur substantial synchronization overhead, depending on how much earlier it is executed than the corresponding send.
Deadlock Example

... supposing send and recv matches the other side

<table>
<thead>
<tr>
<th></th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MPI_Ssend()</td>
<td>MPI_Ssend()</td>
</tr>
<tr>
<td></td>
<td>MPI_Recv()</td>
<td>MPI_Recv()</td>
</tr>
<tr>
<td></td>
<td>! Deadlock</td>
<td>! Deadlock</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>MPI_Buffer_attach()</td>
<td>MPI_Buffer_attach()</td>
</tr>
<tr>
<td></td>
<td>MPI_Bsend()</td>
<td>MPI_Bsend()</td>
</tr>
<tr>
<td></td>
<td>MPI_Recv()</td>
<td>MPI_Recv()</td>
</tr>
<tr>
<td></td>
<td>! Deadlock</td>
<td>! Deadlock</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Non-Blocking Send/Receive

- Call returns immediately, which allows for work to be overlapped.
- User must ensure that:
  - Data to be sent is out of the send buffer.
  - Data to be received has finished arriving.

```c
MPI_Request req;
if ( rank == 0 ) {
    MPI_Isend (buf, count, datatype, dest, tag, comm,&req);
    // do work
    MPI_Wait(req,MPI_STATUS_IGNORE);
}
else {
    MPI_Irecv (buf, count, datatype, dest, tag, comm,&req);
    // do work
    MPI_Wait(req,MPI_STATUS_IGNORE);
}
```
Summary: Communication Modes

• Synchronous mode is the "safest", and therefore also the most portable.
  • It does not depend upon the order in which the send and receive are executed (unlike ready mode) or the amount of buffer space (unlike buffered mode and standard mode).
• Synchronous mode can incur substantial synchronization overhead.
Summary: Communication Modes

• Ready mode has the lowest total overhead
  • It does not require a handshake between sender and receiver (like synchronous mode) or an extra copy to a buffer (like buffered or standard mode).
  • The receive must precede the send.
  • This mode will not be appropriate for all messages.
Summary: Communication Modes

• Buffered mode decouples the sender from the receiver.
  • Eliminates synchronization overhead on the sending task and ensures that the order of execution of the send and receive does not matter (unlike ready mode).
  • The programmer can control the size of messages to be buffered, and the total amount of buffer space.
  • Additional system overhead incurred by the copy to the buffer.
Summary: Communication Modes

• Standard mode behavior is implementation-specific
  • The library developer chooses a system behavior that provides good performance and reasonable safety
  • e.g. for many MPI implementations, small messages are buffered (to avoid synchronization overhead) and large messages are sent synchronously (to minimize system overhead and required buffer space)
Which Mode Should I Use?

• Begin with standard mode and modify to incrementally test impact on performance.
• For example, you may find that you can post send and receives early with non-blocking calls, and then do a bunch of computations to hide inter-process communication.
Compiling MPI Programs

• Select the proper package:
  • e.g. `setpkgs -a intel_cluster_studio_compiler`

• Compile:
  • `mpicc -O2 -Wall -o foo foo.c`

• `mpicc`: wrapper around compiler, MPI libs, etc.

• Flags: same meaning as C compiler.
  • `-O2`: optimize
  • `-Wall`: show all warnings
  • `-o <file>`: name of executable
Running MPI Programs

• Add the following to your .bashrc (located in your home directory):

    setpkgs -a intel_cluster_studio_compiler

• Rules of thumb for running, testing, and benchmarking MPI jobs:
  • During development and testing, just run the job from a gateway (login node) with mpirun.
  • If you want to run across multiple nodes, you must submit to the scheduler (see next slide).
Sample SLURM Script

#!/bin/bash
#SBATCH --nodes=2
#SBATCH --tasks-per-node=8
#SBATCH --constrain=intel
#SBATCH --time=00:00:30:00
#SBATCH --mail-user=alex.trebek@jeopardy.com
#SBATCH --mail-type=ALL

setpkgs -a intel_cluster_studio_compiler
export I_MPI_PMI_LIBRARY=/usr/scheduler/slurm/lib/libpmi.so

srun ./mpi_job

This script launches a MPI program using SLURM’s srun command (similar to mpirun). srun will max out the allocation by default, so mpi_job will be run with 16 tasks (i.e. processes) on 2 nodes (8 CPU cores per node).
Next time

• Collective communication