Introduction to parallel computing

Distributed Memory Programming with MPI (1)

Zhiao Shi (modifications by Will French)
Advanced Computing Center for Education & Research
Vanderbilt University
No more shared memory! Each process runs on its own CPU and has its own independent memory space.
The Message-Passing Model

• A process is (traditionally) a program counter and address space.

• Processes may have multiple threads (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.

• Interprocess communication consists of:
  • Synchronization.
  • Movement of data from one process’s address space to another’s.
What is MPI?

• A *message-passing library specification*.  
  • extended message-passing model.
  • not a language or compiler specification.
  • not a specific implementation or product.

• For parallel computers, clusters, and heterogeneous networks.

• Designed to provide access to advanced parallel hardware for:
  • end users.
  • library writers.
  • tool developers.
Versions of MPI

• MPI itself is a standard that defines an API (namely a collection of functions and data types) for passing data between processes.
• There are a number of different MPI versions that implement these data types and functions in slightly different ways (but the API is the same!).
• Examples: Open MPI, MPICH, Intel MPI
• We’ll be using Intel’s MPI:
  • setpkgs –a intel_cluster_studio_compiler
A Minimal MPI Program (C)

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

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```
Better Hello (C)

#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
Compilation

mpicc -g -Wall -o mpi_hello mpi_hello.c

wrapper script to compile

source file

produce debugging information

turns on all warnings

create this executable file name (as opposed to default a.out)
mpicc

• Just a wrapper around the actual compiler.
• Intel’s mpicc is a Bash script.
• Links to appropriate libraries.
  • Don’t need to manually link with –L and –l.
• Points to mpi.h directory.
  • Don’t need to manually point to include directory with –l.
Execution: mpirun

```
mpirun -n <number of processes> <executable>
```

```
mpirun -n 1 ./mpi_hello
```

*run with 1 process*

```
mpirun -n 4 ./mpi_hello
```

*run with 4 processes*
Execution

```bash
mpirun -n 1 ./mpi_hello
Greetings from process 0 of 1 !

mpirun -n 4 ./mpi_hello
Greetings from process 0 of 4 !
Greetings from process 1 of 4 !
Greetings from process 2 of 4 !
Greetings from process 3 of 4 !
```
MPI Programs

- Can be written in C/C++, Fortran, Java, Python.
  - We’ll be using C.
  - Need to add mpi.h header file.
- Identifiers defined by MPI start with “MPI_”.
- First letter following underscore is uppercase.
  - For function names and MPI-defined types.
  - Helps to avoid confusion.
MPI Components

• **MPI_Init**
  • Tells MPI to do all the necessary setup.

```c
int MPI_Init(
    int* argc_p  /* in/out */,
    char*** argv_p /* in/out */);
```

• **MPI_Finalize**
  • Tells MPI we’re done, so clean up anything allocated for this program.

```c
int MPI_Finalize(void);
```
Basic Outline

#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    /* no MPI calls before this */
    MPI_Init( &argc, &argv );

    .......
    .......
    .......
    MPI_Finalize();

    /* no MPI calls after this */
    return 0;
}
Communicators

• A collection of processes that can send messages to each other.
• MPI_Init defines a communicator that consists of all the processes created when the program is started.

• Called MPI_COMM_WORLD.
Communicators

```c
int MPI_Comm_size(
    MPI_Comm comm /* in */,
    int* comm_sz_p /* out */);
```

*number of processes in the communicator*

```c
int MPI_Comm_rank(
    MPI_Comm comm /* in */,
    int* my_rank_p /* out */);
```

*my rank (the process making this call)*
Basic Communication

```c
int MPI_Send(
    void* msg_buf_p, /* in */,
    int msg_size,  /* in */,
    MPI_Datatype msg_type, /* in */,
    int dest,       /* in */,
    int tag,        /* in */,
    MPI_Comm communicator /* in */);
```
## Data types

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
Basic Communication

```c
int MPI_Recv(
    void* msg_buf_p /* out */,
    int buf_size /* in */,
    MPI_Datatype buf_type /* in */,
    int source /* in */,
    int tag /* in */,
    MPI_Comm communicator /* in */,
    MPI_Status* status_p /* out */);
```
Message matching

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

```c
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag, recv_comm, &status);
```
Receiving messages

• A receiver can get a message without knowing:
  • the amount of data in the message
  • the sender of the message
  • the tag of the message

MPI_ANY_SOURCE

MPI_ANY_TAG
status_p argument

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

MPI_Status* status;

status.MPI_SOURCE
status.MPI_TAG
status.MPI_ERROR
status.size
How much data am I receiving?

```c
int MPI_Get_count(
    MPI_Status* status_p /* in */,
    MPI_Datatype type    /* in */,
    int*        count_p  /* out */);
```
Issues with send and receive

• MPI_Recv always **blocks** until a matching message is received.
• MPI_Send will return when it is safe to use the send buffer.
  • This does not imply that the message has been completely received!
Case study: The Trapezoidal Rule

(a) 

(b)
One trapezoid
Tasks and communications for Trapezoidal Rule

- Compute area of trap 0
- Compute area of trap 1
- Compute area of trap $n-1$
- Add areas
Next time

• Communication modes