CSC510
Parallel Programming

Distributed-Memory Programming
(Part 1)

Content

• Introduction to MPI
• Point-Point Communication
  – Send and receive
• Writing your first MPI program
• Using the common MPI functions
• The Trapezoidal Rule in MPI

Textbook:
An Introduction to Parallel Programming
Author: Peter Pacheco
http://www.cs.usfca.edu/~peter/ipp/
What is MPI?

- An Interface Specification:
  - MPI = Message Passing Interface
- MPI is a specification of message passing libraries for the developers and users.
- The goal of MPI
  - to provide a widely used standard for writing message passing programs. The interface attempts to be
    - practical
    - portable
    - efficient
    - flexible

History

- MPI 1
  - The message passing interface effort began in the summer of 1991.
  - MPI1 was released in 1994. About 128 functions in MPI 1.3
- MPI-2
  - Picked up where the first MPI specification left off, and addresses topics which go beyond the first MPI specification.
- MPI-3
  - Includes new Fortran 2008 bindings,
- MPICH
  - a freely available, portable implementation of MPI.
  - The original implementation of MPICH is called MPICH1 and it implements the MPI-1.1 standard.
  - MPICH 2, implementing the MPI-2.1.
  - MPICH v3.0 implements the MPI-3.0 standard.
Reason for Using MPI

- Standardization
  - MPI is the standard message passing library
  - It is supported on virtually all HPC platforms
- Portability
  - There is no need to modify your source code when you run your application on different platforms that support (and is compliant with) the MPI standard.
- Functionality
  - Many routines are defined.
- Availability
  - A variety of implementations are available, both vendor and public domain.

Programming Models

- Hardware platforms:
  - Distributed Memory
    - Originally, MPI was targeted for distributed memory systems.
  - Shared Memory
  - Hybrid
    - MPI is now used on most common parallel architectures
- All parallelism is explicit
  - The programmer is responsible for
    - identifying parallelism and implementing parallel algorithms using MPI constructs.
    - The processes coordinate their activities by explicitly sending and receiving messages
- Static
  - The number of tasks dedicated to run a parallel program is static.
Call MPI Functions

• MPI subroutines and functions can be called from Fortran and C/C++, respectively
• Compiled with FORTRAN or C compilers
• Bindings are available for many other languages, including Perl, Python, R, Ruby, Java.

Identifying MPI Processes

• Message Passing Interface (MPI)
  • Statically allocate processes
    • The number of processes is set at the beginning of the program execution
    • no additional processes are created during execution
  • Each process is assigned a unique integer rank in the range 0, 1, … p-1
    • p is the total number of processes defined
  • Basically, one can write a single program and execute on different processes (SPMD)
Point-to-Point Communication

- Basic communication pattern is send/receive
  - One process sends a message to another process
  - Receiver asks to receive the message
  - Send could precede the receive or vice versa

Hello World!

```c
#include <stdio.h>

int main(void) {
    printf("hello, world\n");

    return 0;
}
```
Our first MPI program

```c
#include <stdio.h>
#include <string.h>  /* For strings */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
  char* greeting[MAX_STRING];
  int comm_sz;    /* Number of processes */
  int my_rank;    /* My process rank */
  MPI_Init(NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  if (my_rank == 0) {
    sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
    MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
             MPI_COMM_WORLD);
  } else {
    printf("Greetings from process %d of %d\n", my_rank, comm_sz);
    for (int q = 1; q < comm_sz; ++q) {
      MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
                0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
      printf("%s\n", greeting);
    }
  }
  MPI_Finalize();
  return 0;
} /* main */
```

Basic Outline of MPI a Program

```c
#include <mpi.h>

int main(int argc, char* argv[]) {
  /* No MPI calls before this */
  MPI_Init(&argc, &argv);
  /* Do work and make message passing calls */
  MPI_Finalize();
  /* No MPI calls after this */
  return 0;
}
```
MPI Programs

- Written in C.
  - Has main.
  - Uses stdio.h, string.h, etc.
- 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);
```
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.

```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)*
SPMD

- Single-Program Multiple-Data
- We compile one program.
- Process 0 does something different.
  - Receives messages and prints them while the other processes do the work.
- The if-else construct makes our program SPMD.

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>

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

\[
\text{MPI\_Send}(\text{send\_buf\_p}, \text{send\_buf\_sz}, \text{send\_type}, \text{dest}, \text{send\_tag}, \text{send\_comm});
\]

\[
\text{MPI\_Recv}(\text{recv\_buf\_p}, \text{recv\_buf\_sz}, \text{recv\_type}, \text{src}, \text{recv\_tag}, \text{recv\_comm}, &\text{status});
\]

status_p argument

\[
\text{MPI\_Recv}(\text{recv\_buf\_p}, \text{recv\_buf\_sz}, \text{recv\_type}, \text{src}, \text{recv\_tag}, \text{recv\_comm}, &\text{status});
\]

\[
\text{MPI\_Status}\ast \text{status};
\]

\[
\text{status.MPI\_SOURCE}
\]

\[
\text{status.MPI\_TAG}
\]

\[
\text{status.MPI\_ERROR}
\]
Our first MPI program

```c
#include <stdio.h>
#include <string.h> /* For strings */
#include <mpi.h> /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
    char greeting[MAX_STRING];
    int comm_sz; /* Number of processes */
    int my_rank; /* My process rank */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0) { /* master */
        printf("Greetings from process \%d of \%d\n", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting), MPI_CHAR, 0, 0,
                 MPI_COMM_WORLD);
    } else { /* worker */
        printf("Greetings from process \%d of \%d\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
                      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```

Compile and Execute MPI Programs

- Step 1: Set up the environment on sequoia:
  . /usr/local/apps/mpt_2.sh
- Step 2: Compile your mpi_hello.c program:
  gcc -lmpi -o mpi_hello mpi_hello.c
- Step 3: Specify the executable file in the pbs file
  mpiexec -np 8 ./mpi_hello
- Step 4: Submit the job using qsub:
  qsub test.pbs
PBS Example

// Example of PBS for sequoia at MCSR

#PBS -S /bin/bash
#PBS -q MCSR-4N   //Using this type of processors
#PBS -l nodes=2:ppn=4   //2 node, each with 4 processors
#PBS –l cput=00:10:00  //Estimated time in 10 minutes
#PBS -l mem=100MB    //Estimated memory usage
#PBS -N test   //Name this job “test”
cd ${PBS_O_WORKDIR}
./usr/local/apps/mpt_2.sh  //set up the environment
mpiexec -np 8 ./mpi_hello    //Execute file with 8 processors

View Results

• Use ls –l to list all the files, you will see the the output files (test.o234019)
• Use cat test.o234019 to see the results
Dealing with I/O

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

int main(void) {
    int my_rank, comm_sz;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    printf("Proc %d of %d > Does anyone have a toothpick?\n", my_rank, comm_sz);

    MPI_Finalize();
    return 0;
} /* main */
```

Each process just prints a message.

Running with 6 processes

Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?

unpredictable output
Summary

• MPI = Message Passing Interface
  – MPI is a specification of message passing libraries for the developers and users.

• Platforms
  – MPI was targeted for distributed memory systems.
  – MPI is now used on any common parallel architecture

• All parallelism is explicit
  – Programmers specify communications

• The number of tasks dedicated to run a parallel program is static.

• MPI Programming