CSC630/CSC730: Parallel Computing

Parallel Programming

Using MPI (1)

Dr. Joe Zhang

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

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

Dr. Joe Zhang

PDC-7: MPI (1)
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.
  - MP11 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
  - 128 routines callable from C/C++ or Fortran
  - MPI-1 doesn’t support F90, but MPI-2 does support F90 and C++
- Compiled with FORTRAN or C compilers
- Bindings are available for many other languages, including Perl, Python, R, Ruby, Java.

Introduction to MPI

- 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;
}
```
Identifying MPI processes

- Common practice to identify processes by nonnegative integer ranks.

- \( p \) processes are numbered 0, 1, 2, \( .. p-1 \)

Our first MPI program

```c
#include <stdio.h>
#include <string.h> /* For strlen */
#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 \#\# of \#\#!", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Greetings from process \#\# of \#\#!", my_rank, comm_sz);
        for (int q = 1; q <= comm_size; q++) {
            MPI_Send(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }
    MPI_Finalize();
    return 0;
    /* main */
```
### Compilation

- **mpicc** 
  - `-g` turns on all warnings
  - `-Wall` produces debugging information
  - `-o mpi_hello` creates this executable file name (as opposed to default `a.out`)

```
mpicc -g -Wall -o mpi_hello mpi_hello.c
```

### Execution

- **mpiexec** 
  - `-n <number of processes> <executable>`

```
mpiexec -n 1 ./mpi_hello
```

```
mpiexec -n 4 ./mpi_hello
```

- `run with 1 process`
- `run with 4 processes`
Execution

mpiexec -n 1 ./mpi_hello

Greetings from process 0 of 1!

mpiexec -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

• 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);
  ```

Basic Outline

```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;
}
```
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

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);

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
The Trapezoidal Rule

(a) 
(b)
The Trapezoidal Rule

Area of one trapezoid \( \frac{h}{2} [f(x_i) + f(x_{i+1})] \)

\[ h = \frac{b - a}{n} \]

\( x_0 = a, \ x_1 = a + h, \ x_2 = a + 2h, \ldots, \ x_{n-1} = a + (n - 1)h, \ x_n = b \)

Sum of trapezoid areas = \( h \left[ f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2 \right] \)
Pseudo-code for a serial program

```c
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

Parallelizing the Trapezoidal Rule

1. Partition problem solution into tasks.
2. Identify communication channels between tasks.
3. Aggregate tasks into composite tasks.
4. Map composite tasks to cores.
Parallel pseudo-code

1. Get a, b, n;
2. h = (b—a)/n;
3. local_n = n/comm_sz;
4. local_a = a + my_rank*local_n*h;
5. local_b = local_a + local_n*h;
6. local_integral = Trap(local_a, local_b, local_n, h);
7. if (my_rank != 0)
   Send local_integral to process 0;
8. else /* my_rank == 0 */
9.    total_integral = local_integral;
10. for (proc = 1; proc < comm_sz; proc++) {
11.    Receive local_integral from proc;
12.    total_integral += local_integral;
13. }
14. }
15. if (my_rank == 0)
16.   print result;

Tasks and communications for Trapezoidal Rule
First version (1)

```c
int main(void) {
    int my_rank, comm_sz, n = 1024, local_n;
    double a = 0.0, b = 3.0, h, local_a, local_b;
    double local_int, total_int;
    int source;

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

    h = (b-a)/n;  /* h is the same for all processes */
    local_n = n/comm_sz; /* So is the number of trapezoids */

    local_a = a + my_rank*local_n;  
    local_b = local_a + local_n*h;
    local_int = Trap(local_a, local_b, local_n, h);

    if (my_rank != 0) {
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
}
```

First version (2)

```c
} else {
    total_int = local_int;
    for (source = 1; source < comm_sz; source++) {
        MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total_int += local_int;
    }
}

if (my_rank == 0) {
    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.15e\n", a, b, total_int);
}
MPI_Finalize();
return 0;
} /* main */
```

Dr. Joe Zhang  
PDC-7: MPI (1)
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

Input

• Most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin.
• Process 0 must read the data (scanf) and send to the other processes.

MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
Get_data(my_rank, comm_sz, &a, &b, &n);
    h = (b-a)/n;
Function for reading user input

```c
void Get_input()
{
    int my_rank /* in */,
    int comm_sz /* in */,
    double* a_p /* out */,
    double* b_p /* out */,
    int* n_p /* out */
    int dest;

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %ld", a_p, b_p, n_p);
        for (dest = 1; dest < comm_sz; dest++) {
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else { /* my_rank != 0 */
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
                  MPI_STATUS_IGNORE);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
                  MPI_STATUS_IGNORE);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                  MPI_STATUS_IGNORE);
    }
} /* Get_input */
```

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
CSC630/CSC730: Parallel Computing

Questions?