Distributed Memory Programming With MPI

2014 Spring

Jinkyu Jeong
(jinkyu@skku.edu)
Roadmap

- Hello World in MPI program
- Basic APIs of MPI
- Example program
  - The Trapezoidal Rule in MPI.
- Collective communication.
- MPI derived datatypes.
- Performance evaluation of MPI programs.
- Parallel sorting.
- Safety in MPI programs.
A shared memory system
A distributed memory system

- CPUs
- Memory
- Interconnect
Hello World!

```c
#include <stdio.h>

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

    return 0;
}
```

(a classic)
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 %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 */
```
Compilation

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

- **Wrapper script to compile**
- **Source file**
- **Produces debugging information**
- **Turns on all warnings**
- **Create this executable file name (as opposed to default `a.out`)**
 mpiexec  -n  <number of processes>  <executable>

 mpiexec  -n  1  ./mpi_hello
 run with 1 process

 mpiexec  -n  4  ./mpi_hello
 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

... include <mpi.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

- Communicator = context + group
- Communication in MPI takes place with respect to communicators
  - `MPI_COMM_WORLD` is a predefined one
  - All processes are initially in the group
- Processes may belong to many different communicators
Communicators

- **MPI_Comm_size**

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

  number of processes in the communicator

- **MPI_Comm_rank**

  ```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

- **MPI_Send**
  - Send a message

```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>unsigned long long int</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>float</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
**Communication**

- **MPI_Recv**
  - Receive a message

```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 */) ;
```
Point-to-Point Communications

- Sending and receiving messages

Message

- Data
  - Buffer: address
  - Count
  - Datatype: MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_DOUBLE, ...

- Envelope
  - Process ID (source/destination rank)
  - Message tag
  - Communicator

Process 0
A: Send
Process 1
B: Receive
Point-to-Point Communications (cont’d)

- **Message tag**
  - Tags allow programmers to deal with the arrival of messages in an orderly manner
  - Range of tag
    - 0 .. 32767 ($2^{15} - 1$) are guaranteed
    - The upper bound is provided by MPI_TAG_UB
    - MPI_ANY_TAG can be used as a wildcard value

- **Basic point-to-point communication**
  - MPI_Send (buf, count, datatype, destID, tag, comm);
  - MPI_Recv(buf, count, datatype, sourceID, tag, comm, status);
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);
```
Receiving messages

- **Two wildcards of MPI_recv**
  - MPI_ANY_SOURCE
  - MPI_ANY_TAG

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

- **A receiver can get a message without knowing:**
  - the amount of data in the message,
  - the sender of the message,
  - or the tag of the message.
**status_p argument**

```c
MPI_Status* status;
status.MPI_SOURCE
status.MPI_TAG
```
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

- Exact behavior is determined by the MPI implementation.
- MPI_Send may behave differently with regard to buffer size, cutoffs and blocking.
- MPI_Recv always blocks until a matching message is received.
- Know your implementation; don’t make assumptions!
TRAPEZOIDAL RULE IN MPI
The Trapezoidal Rule

(a)

(b)
One trapezoid

\[ f(x_i) \]

\[ f(x_{i+1}) \]

\[ y = f(x) \]

\[ x_i \quad x_{i+1} \]

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

/* Input:  a, b, n */
h = (b−a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 0; 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

```c
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)
8     Send local_integral to process 0;
9  else /* my_rank == 0 */
10     total_integral = local_integral;
11     for (proc = 1; proc < comm_sz; proc++) {
12         Receive local_integral from proc;
13         total_integral += local_integral;
14     }
15 }
16  if (my_rank == 0)
17     print result;
```
Tasks and communications for Trapezoidal Rule

- Compute area of trap 0
- Compute area of trap 1
- ... Compute area of trap \( n-1 \)
- Add areas
```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*h;
    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);
    }
}```
```c
} else {
    total_int = local_int;
    for (source = 1; source < commSz; 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 */
```
double Trap(
    double left_endpt /* in */,
    double right_endpt /* in */,
    int trap_count /* in */,
    double base_len /* in */) {

double estimate, x;
int i;

estimate = (f(left_endpt) + f(right_endpt))/2.0;

for (i = 1; i <= trap_count - 1; i++) {
    x = left_endpt + i*base_len;
    estimate += f(x);
}

estimate = estimate*base_len;

    return estimate;
} /* Trap */
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.

```c
. . .
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 %d", 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 */
```