Table of Contents

1 Misc Information
2 Distributed Memory Programming with MPI
   - Message Passing Interface
   - MPI Programming
   - MPI Communication
   - Trapezoid Rule for Numerical Integration
3 Next Time
Today:
- Resource Management)
- Using the PBS queue
- Running jobs on the student cluster

Next HW2: Basic MPI Routines; Due ..

Quiz 1: 09/25/14 (Thurs)
Distributed-memory system: collection of cores, connected with a network, each with its own memory.

Shared-memory system: collection of cores interconnected to a global memory.
Serial Hello World

/* hello.c  --  serial version */

#include <stdio.h>
#include <unistd.h>

int main(void)
{
    char cptr[100];

    gethostname(cptr,100);
    printf("hello, world from %s\n", cptr);

    return 0;
}
Parallel Hello World

/* File:
   * mpi_hello.c
   * Purpose:
   *   A "hello,world" program that uses MPI
   * Compile:
   *   mpicc -g -Wall -std=C99 -o mpi_hello mpi_hello.c
   * Usage:
   *   mpiexec -n<number of processes> ./mpi_hello
   * Input:
   *   None
   * Output:
   *   A greeting from each process
   * Algorithm:
   *   Each process sends a message to process 0, which prints
   *   the messages it has received, as well as its own message.
   * IPP: Section 3.1 (pp. 84 and ff.)
*/
#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]; /* String storing message */
    int comm_sz; /* Number of processes */
    int my_rank; /* My process rank */
    int q;

    /* Start up MPI */
    MPI_Init(NULL, NULL);

    /* Get the number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    /* Get my rank among all the processes */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        /* Create message */
        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);

        /* Send message to process 0 */
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        /* Print my message */
        printf("Greetings from process %d of %d!
", my_rank, comm_sz);

        for (q = 1; q < comm_sz; q++) {
            /* Receive message from process q */
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            /* Print message from process q */
            printf("%s\n", greeting);
        }
    }

    /* Shut down MPI */
    MPI_Finalize();

    return 0;
} /* main */
Distributed Memory Programming with MPI

Message Passing Interface

batch here

COMPILE CODE

ợmpicc -g -pg -Wall -o mpi_hello mpi_hello.c

RUN CODE FROM COMMAND LINE

ợmpirun -np 16 ./mpi_hello
Greetings from process 0 of 16!
Greetings from process 1 of 16!
Greetings from process 2 of 16!
Greetings from process 3 of 16!
Greetings from process 4 of 16!
Greetings from process 5 of 16!
Greetings from process 6 of 16!
Greetings from process 7 of 16!
Greetings from process 8 of 16!
Greetings from process 9 of 16!
Greetings from process 10 of 16!
Greetings from process 11 of 16!
Greetings from process 12 of 16!
Greetings from process 13 of 16!
Greetings from process 14 of 16!
Greetings from process 15 of 16!

ợmpirun -np 16 --nooversubscribe ./mpi_hello

There are not enough slots available in the system to satisfy the 16 slots that were requested by the application: ./mpi_hello

Either request fewer slots for your application, or make more slots available for use.
How does MPI work?

1. On Initialization, MPI assigns \( np \) processors/cores to a "communicator" group called \( MPI\_COMM\_WORLD \).

2. MPI launches an identical copy of \( mpi\_hello \) on each of the \( p \) requested processors.

3. Processes use \( MPI\_COMM\_WORLD \) to get information about the group:
   - Number of processes: \( MPI\_Comm\_size(MPI\_COMM\_WORLD, ...) \)
   - Process ID/Rank: \( MPI\_Comm\_rank(MPI\_COMM\_WORLD, ...) \)
**Message Passing Interface**

- Written in C (or Fortran, Python, etc.)
- 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
Basic MPI Routines

Message Passing Interface

- For running codes on distributed memory systems.
- Data resides on other processes – accessed through MPI calls.
- The minimal set of routines that most parallel codes use:
  - MPI_INIT
  - MPI_COMM_SIZE
  - MPI_COMM_RANK
  - MPI_SEND
  - MPI_RECV
  - MPI_FINALIZE
**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);
    /* No MPI calls after this */
    MPI_Finalize();
    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)*
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 */,
    int datatype, /* in */,
    int source, /* in */,
    int tag, /* in */,
    MPI_Comm communicator, /* in */,
    MPI_Status* status_p /* out */);
```
Message matching

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

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

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

```fortran
program template
   !-- Template for any mpi program
       implicit none ! highly recommended. It will make
      ! debugging infinitely easier.
   !--Include the mpi header file
      include mpif.h ! --> Required statement
   !--Declare all variables and arrays.
      integer ierr,myid,numprocs,itag
      integer irc
   !--Initialize MPI
      call MPI_INIT( ierr ) ! --> Required statement
   !--Who am I? get my rank=myid
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
   !--How many processes in the global group?
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
   !--Finalize MPI
      call MPI_FINALIZE(irc) ! --> Required statement
??stop end
```
The Trapezoid Rule for Numerical Integration

**Solve the Integral:** \( \int_{a}^{b} F(x) \, dx \)

**The Trapezoidal Rule**

Where \( F(x) \) can be any function of \( x \): \( f(x^2), f(x^3) \) See Pacheco (2011), Ch3.
Trapezoid Equations

Integral: \[ \int_{a}^{b} f(x) \, dx \]

Area of 1 trapezoid: \[ = \frac{h}{2} \left[ f(x_i) + f(x_{i+1}) \right] \]

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

Endpoints: \[ x_0 = a, \quad x_1 = a + h, \quad x_2 = a + 2h, \ldots, \quad x_{n-1} = a + (n - 1) h, \quad x_c = b \]

Sum of Areas: \[ \text{Area} = h \left[ \frac{f(x_0)}{2} + f(x_{i+1}) + f(x_{i+1}) + \ldots + f(x_{n-1}) \frac{f(x_n)}{2} \right] \]
Pseudocode for a Serial Algorithm

```c
/* 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 Pseudocode

Get $a, b, n$;
$h = b * a / n$;
local$_n = n / \text{comm\_sz}$;
local$_a = a + \text{my\_rank} * \text{local\_n} * h$;
local$_b = \text{local\_a} + \text{local\_n} * h$;
local$_\text{integral} = \text{Trap}(\text{local\_a}, \text{local\_b}, \text{local\_n}, h)$;
if (my\_rank != 0)
Send local$_\text{integral}$ to process 0;
else /* my\_rank==0 */
total$_\text{integral} = \text{local\_integral}$;
for (proc = 1; proc < comm$_\text{sz}$; proc++) {
Receive local$_\text{integral}$ from proc;
total$_\text{integral} += \text{local\_integral}$;
}
if (my\_rank == 0)
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
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*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);
  }
```

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 */
```
First version (3)

```c
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 */
```
Parallel *trap* distributes \( [x_i] \) data points to different processors.

- Each processor runs the same code but solves for different \( [x_i] \) data
- This is a 1-D data decomposition
- This is a 1-D processor arrangement
Distributed Memory Programming with MPI

Trapezoid Rule for Numerical Integration
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.

```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;
...```

What happens if you are using the batch queuing system? There is no way to capture stdin, so you have to read in data as command line args or from a file.
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 */
```
Next Time

- Next class: 09/23/14
- Topic: MPI Point-to-Point and collective communications