# MPI Basics

Ned Nedialkov

Department of Computing and Software
McMaster University, Hamilton, Ontario
Canada

# Preliminaries

A *process* is an instance of a program

Processes can be created and destroyed

MPI assumes statically allocated processes

Their number is set at the beginning of program execution

No additional processes are created

Each process is assigned a unique number or *rank*, which is from 0 to $p - 1$, where $p$ is the number of processes

Number of processes is not necessarily number of processors; a processor may execute more than one process

# Blocking communication

Assume that process 0 sends data to process 1

In a blocking communication, the sending routine returns only after the buffer it uses is ready to be reused

Similarly, in process 1, the receiving routine returns after the data is completely stored in its buffer

Blocking send and receive: MPI_Send and MPI_Recv

MPI_Send: sends data; does not return until the data have been safely stored away so that the sender is free to access and overwrite the send buffer

The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

MPI_Recv: receives data; it returns only after the receive buffer contains the newly received message

# MPI program structure

- Include mpi.h
- Initialize MPI environment
- Do computations
- Terminate MPI environment

Initialize with MPI_Init

Terminate with MPI_Finalize

```c
#include "mpi.h"

int main(int argc, char* argv[])
{
  /* ... */

  /* This must be the first MPI call */
  MPI_Init(&argc, &argv);

  /* Do computation */

  MPI_Finalize();
  /* No MPI calls after this line */

  /* ... */
  return 0;
}
```

# MPI_Send

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

buf beginning of the buffer containing the data to be sent

count number of elements to be sent (not bytes)

datatype type of data, e.g. MPI_INT, MPI_DOUBLE, MPI_CHAR

dest rank of the process, which is the destination for the message

tag number, which can be used to distinguish among messages

comm communicator: a collection of processes that can send messages to each other, e.g. MPI_COMM_WORLD

MPI_COMM_WORLD: all the processes running when execution begins

Returns error code

# MPI_Recv

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

buf beginning of the buffer where data is received

count number of elements to be received (not bytes)

datatype type of data, e.g. MPI_INT, MPI_DOUBLE, MPI_CHAR

source rank of the process from which to receive

tag number, which can be used to distinguish among messages

comm communicator

status information about the data received, e.g, rank of source, tag, error code

Returns error code

# First program

Programs are adapted from P. Pacheco, Parallel Programming with MPI

---

```
/* greetings.c
 * Send a message from all processes with rank != 0
 * to process 0.
 * Process 0 prints the messages received.
 */
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char* argv[])
{
   int         my_rank;       /* rank of process      */
   int         p;             /* number of processes  */
   int         source;        /* rank of sender       */
   int         dest;          /* rank of receiver     */
   int         tag = 0;       /* tag for messages     */
   char        message[100];  /* storage for message  */
   MPI_Status  status;        /* status for receive   */
```

```
/* Start up MPI */
MPI_Init(&argc, &argv);

/* Find out process rank  */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Find out number of processes */
MPI_Comm_size(MPI_COMM_WORLD, &p);


if (my_rank != 0)
  {
    /* Create message */
    sprintf(message, "Greetings from process %d!",
            my_rank);
    dest = 0;
    /* Use strlen+1 so that '\0' gets transmitted */
    MPI_Send(message, strlen(message)+1, MPI_CHAR,
             dest, tag, MPI_COMM_WORLD);
  }
```

```
    else
      { /* my_rank == 0 */
        for (source = 1; source < p; source++)
          {
            MPI_Recv(message, 100, MPI_CHAR, source, tag,
                     MPI_COMM_WORLD, &status);
            printf("%s\n", message);
          }
      }

    /* Shut down MPI */
    MPI_Finalize();

    return 0;
}
```

# Compilation and execution

An executable can be created with

```
mpicc -o greetings greetings.c
```

Run with 4 processes

```
mpirun -np 4 greetings
```

# Example: numerical integration

The trapezoid rule for $\int_a^b f(x)dx$ with $h = (b-a)/n$ is

$$f(x) \approx \frac{h}{2}\big(f(x_0) + f(x_n)\big) + h\sum_{i=1}^{n-1} f(x_i),$$

where $x_i = a + ih$, $i = 0, 1, \ldots, n$

Given $p$ processes, each process can work on $n/p$ subintervals (assume $n/p$ is an integer)

| process | interval |
|---|---|
| 0 | $[a, a + \frac{n}{p}h]$ |
| 1 | $[a + \frac{n}{p}h, a + 2\frac{n}{p}h]$ |
| $\vdots$ | |
| $p-1$ | $[a + (p-1)\frac{n}{p}h, b]$ |

# Parallel trapezoid

Assume $f(x) = x^2$

We write our function $f(x)$ in

```
/* func.c */

float f(float x)
{
   return x*x;
}
```

The trapezoid rule is implemented in

```
/* traprule.c */

extern float f(float x); /* function we're integrating */

float Trap(float a, float b, int n, float h)
{
  float integral;   /* Store result in integral  */
  float x;
  int i;

  integral = (f(a) + f(b))/2.0;

  x = a;
  for ( i = 1; i <= n-1; i++ )
    {
      x = x + h;
      integral = integral + f(x);
    }

  return integral*h;
}
```

The parallel program is

```
/* trap.c -- Parallel Trapezoidal Rule
 *
 * Input: None.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Algorithm:
 *    1.  Each process calculates "its" interval of
 *        integration.
 *    2.  Each process estimates the integral of f(x)
 *        over its interval using the trapezoidal rule.
 *    3a. Each process != 0 sends its integral to 0.
 *    3b. Process 0 sums the calculations received from
 *        the individual processes and prints the result.
 *
 *        The number of processes (p) should evenly divide
 *        the number of trapezoids (n = 1024)
 */


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

extern float Trap(float a, float b, int n, float h);
```

```
int main(int argc, char** argv)
{
   int           my_rank;    /* My process rank                */
   int           p;          /* The number of processes        */
   float         a = 0.0;    /* Left endpoint                  */
   float         b = 1.0;    /* Right endpoint                 */
   int           n = 1024;   /* Number of trapezoids           */
   float         h;          /* Trapezoid base length          */
   float         local_a;    /* Left endpoint my process       */
   float         local_b;    /* Right endpoint my process      */
   int           local_n;    /* Number of trapezoids for       */
                             /* my calculation                 */
   float         integral;   /* Integral over my interval      */
   float         total=-1;   /* Total integral                 */
   int           source;     /* Process sending integral       */
   int           dest = 0;   /* All messages go to 0           */
   int           tag = 0;
   MPI_Status    status;

   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
```

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

/* Length of each process' interval of
   integration = local_n*h. */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);

/* Add up the integrals calculated by each process */
if (my_rank == 0)
  {
    total = integral;
    for (source = 1; source < p; source++)
      {
        MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                 MPI_COMM_WORLD, &status);
        printf("PE %d <- %d, ... %f\n", my_rank,source,
               integral);
        total = total + integral;
      }
  }
else
  {
    printf("PE %d -> %d, ... %f\n", my_rank, dest, integral);
```

```
      MPI_Send(&integral, 1, MPI_FLOAT, dest,
                  tag, MPI_COMM_WORLD);
   }



   /* Print the result */
   if (my_rank == 0)
   {
      printf("With n = %d trapezoids, our estimate\n",
               n);
      printf("of the integral from %f to %f = %f\n",
              a, b, total);
   }

   MPI_Finalize();

   return 0;
}
```

# I/O

We want to read $a$, $b$, and $n$ from the standard input

Function Get_data reads $a$, $b$, and $n$

Cannot be called in each process

Process 0 calls Get_data, which sends these data to processes $1, 2, \ldots, p-1$

The same scheme applies if we read from a file

This is the Get_data function

```
/* getdata.c

 * Reads in the user input a, b, and n.
 * Input parameters:
 *     1.  int my_rank:  rank of current process.
 *     2.  int p:  number of processes.
 * Output parameters:
 *     1.  float* a_ptr:  pointer to left endpoint a.
 *     2.  float* b_ptr:  pointer to right endpoint b.
 *     3.  int* n_ptr:  pointer to number of trapezoids.
 * Algorithm:
 *     1.  Process 0 prompts user for input and
 *         reads in the values.
 *     2.  Process 0 sends input values to other
 *         processes.
 */
```

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

void Get_data( float* a_ptr, float* b_ptr, int* n_ptr,
               int my_rank, int p )
{
  int source = 0, dest, tag;
  MPI_Status status;


  if (my_rank == 0)
    {
      printf("Rank %d: Enter a, b, and n\n", my_rank);
      scanf("%f %f %d", a_ptr, b_ptr, n_ptr);

      for (dest = 1; dest < p; dest++)
        {
          tag = 0;
          MPI_Send(a_ptr, 1, MPI_FLOAT, dest, tag,
                   MPI_COMM_WORLD);
          tag = 1;
          MPI_Send(b_ptr, 1, MPI_FLOAT, dest, tag,
                   MPI_COMM_WORLD);
          tag = 2;
          MPI_Send(n_ptr, 1, MPI_INT, dest, tag,
                   MPI_COMM_WORLD);
```

```
        }
    }
  else
    {
      tag = 0;
      MPI_Recv(a_ptr, 1, MPI_FLOAT, source, tag,
               MPI_COMM_WORLD, &status);
      tag = 1;
      MPI_Recv(b_ptr, 1, MPI_FLOAT, source, tag,
               MPI_COMM_WORLD, &status);
      tag = 2;
      MPI_Recv(n_ptr, 1, MPI_INT, source, tag,
               MPI_COMM_WORLD, &status);
    }
}
```

Now the parallel program with input is

```
/* get_data.c -- Parallel Trapezoidal Rule,
    uses basic Get_data function for input.
*/

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

extern void Get_data(float* a_ptr, float* b_ptr,
                     int* n_ptr, int my_rank, int p);
extern float Trap(float a, float b, int n, float h);

int main(int argc, char** argv)
{
   int           my_rank, p;
   float         a, b, h;
   int           n;
   float         local_a, local_b;
   int           local_n;

   float         integral;
   float         total=-1;
   int           source, dest = 0, tag = 0;
   MPI_Status    status;
```

```
MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

Get_data(&a, &b, &n, my_rank, p);

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

/* Length of each process' interval of
 * integration = local_n*h.  So my interval
 * starts at: */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);
```

```
  /* Add up the integrals calculated by each process */
  if (my_rank == 0)
    {
      total = integral;
      for (source = 1; source < p; source++)
        {
          MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                   MPI_COMM_WORLD, &status);
          total = total + integral;
        }
    }
  else
    MPI_Send(&integral, 1, MPI_FLOAT, dest,
             tag, MPI_COMM_WORLD);
  /* Print the result */
  if (my_rank == 0)
    {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %f\n",
             a, b, total);
    }

  MPI_Finalize();
  return 0;
}
```

Makefile is

```
CC = gcc
MPICC = mpicc
CFLAGS = -Wall -O2 -g

OBJECTS1 = trap.o func.o traprule.o
OBJECTS2 = func.o traprule.o iotrap.o getdata.o

all: partrap iopartrap

partrap: $(OBJECTS1)
        $(MPICC) -o partrap $(OBJECTS1)

iopartrap: $(OBJECTS2)
        $(MPICC) -o iopartrap $(OBJECTS2)

trap.o: trap.c
        $(MPICC) $(CFLAGS) -c trap.c

traprule.o: traprule.c
        $(MPICC) $(CFLAGS) -c traprule.c

func.o: func.c
        $(MPICC) $(CFLAGS) -c func.c
```

```
getdata.o: getdata.c
	$(MPICC) $(CFLAGS) -c getdata.c

iotrap.o: iotrap.c
	$(MPICC) $(CFLAGS) -c iotrap.c

clean:
	rm $(OBJECTS1) $(OBJECTS2)
```

# Summary

One can write many parallel programs using only

- MPI_Init
- MPI_Comm_rank
- MPI_Comm_size
- MPI_Send
- MPI_Recv
- MPI_Finalize