MPI Basics

Ned Nedialkov

McMaster University

24 January 2023

Outline

Processes

Blocking communication

Program structure

Send/Receive

Example: parallel integration

Processes

- A process is an instance of a program.
- MPI-1 assumes statically allocated processes.
  - Their number is set at the beginning of program execution.
  - No additional processes are created.
- MPI-2 can create new processes on the fly.
- Each process is assigned a unique number or rank, which is from 0 to $p-1$, where p is the number of processes.

Blocking communication

- Assume that process 0 sends data to process 1.
- The sending routine returns only after the buffer it uses is ready to be reused.
- The receiving routine returns after the data is completely stored in its buffer.

- Blocking send and receive: `keywordstyleMPI_Send` and `keywordstyleMPI_Recv`.
- `keywordstyleMPI_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.
- `keywordstyleMPI_Recv`: receives data; it returns only after the receive buffer contains the newly received message.

MPI program structure

- Include `mpi.h`
- Initialize MPI environment: `keywordstyleMPI_Init`
- Do computations in parallel
- Terminate MPI environment: `keywordstyleMPI_Finalize`

```
#include "mpi.h"
int main(int argc, char* argv[])
{
  /* This must be the first MPI call */
   keywordstyleMPI_Init(&argc, &argv);
  /* Do computation */
   keywordstyleMPI_Finalize();
  /* No MPI calls after this line */
  return 0;
}
```

MPI_Send

```
int  keywordstyleMPI_Send(void *buf, int count,
        keywordstyleMPI_Datatype datatype, int dest, int tag,
        keywordstyleMPI_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. keywordstyleMPI_INT, keywordstyleMPI_DOUBLE, keywordstyleMPI_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 keywordstyleMPI_COMM_WORLD all the processes running when execution begins

Returns error code

MPI_Recv

```
int  keywordstyleMPI_Recv(void *buf, int count,
        keywordstyleMPI_Datatype datatype, int source, int tag,
        keywordstyleMPI_Comm comm,  keywordstyleMPI_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. `keywordstyleMPI_INT`, `keywordstyleMPI_DOUBLE`, `keywordstyleMPI_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

Example: numerical integration

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

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

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	a_i	b_i
0	$a_0 = a$	$b_0 = a_0 + \frac{n}{p}h$
1	$a_1 = a + \frac{n}{p}h$	$b_1 = a_1 + \frac{n}{p}h$
2	$a_2 = a + 2\frac{n}{p}h$	$b_2 = a_2 + \frac{n}{p}h$
$\vdots$		
$p-1$	$a_{p-1} = a + (p-1)\frac{n}{p}h$	$b_{p-1} = a_{p-1} + \frac{n}{p}h$

Example: parallel trapezoidal

Code adapted from P. Pacheco, Parallel Programming with MPI

```
/* func.c */
#include <math.h>
double f(double x) {
   return exp(-x * x);
}
```

The trapezoidal rule is implemented in

```
/* traprule.c */
extern double f(double x);
double Trap(double a, double b, int n, double h) {
   double integral, 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

```
#include "mpi.h"
#include <stdio.h>
extern double Trap(double a, double b, int n, double h);
int main(int argc, char **argv) {
   int myrank;   // My process rank
   int p;        // The number of processes
   double a = 0.0; // Left endpoint
   double b = 1.0; // Right endpoint
   int n = 16 * 1024; // Number of trapezoids
   double h;     // Trapezoid base length
   double local_a; // Left endpoint my process
   double local_b; // Right endpoint my process
   int local_n;  // Number of trapezoids for
                 // my calculation
   double integral; // Integral over my interval
   double total = -1; // Total integral
   int source;   // Process sending integral
   int dest = 0; // All messages go to 0
   int tag = 0;
    keywordstyleMPI_Status status;
```

```
 keywordstyleMPI_Init(&argc, &argv);
 keywordstyleMPI_Comm_rank( keywordstyleMPI_COMM_WORLD, &myrank);
 keywordstyleMPI_Comm_size( keywordstyleMPI_COMM_WORLD, &p);
// h is the same on all processes
h = (b - a) / n;
// number of subintervals
local_n = n / p;
// subinterval on myrank
local_a = a + myrank * local_n * h;
local_b = local_a + local_n * h;
printf("rank %d, interval [%f, %f]\n", myrank, local_a, local_b);
integral = Trap(local_a, local_b, local_n, h);
if (myrank == 0) {
   total = integral;
   for (source = 1; source < p; source++) {
      // receive values from each process
       keywordstyleMPI_Recv(&integral, 1,
              keywordstyleMPI_DOUBLE, source, tag,
              keywordstyleMPI_COMM_WORLD, &status);
      printf("process 0 <- %d : value %f\n", source, integral);
      // sum up
      total = total + integral;
   }
}
```

```
   else { // send to process 0
      printf("process %d -> 0 : value %f\n", myrank, integral);
       keywordstyleMPI_Send(&integral, 1,
              keywordstyleMPI_DOUBLE, dest, tag,
              keywordstyleMPI_COMM_WORLD);
   }
   // print the results
   if (myrank == 0)
      printf("Value of the integral from %f to %f = %f\n", a, b, total);
    keywordstyleMPI_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

```
#include "mpi.h"
#include <stdio.h>
void Get_data(double *a, double *b, int *n, int myrank, int p) {
   int source = 0, dest, tag;
    keywordstyleMPI_Status status;
   if (myrank == 0) {
      printf("Rank %d: Enter a, b, and n\n", myrank);
      scanf("%lf %lf %d", a, b, n);
      for (dest = 1; dest < p; dest++) {
         tag = 0;
          keywordstyleMPI_Send(a, 1,  keywordstyleMPI_DOUBLE, dest, tag,
                 keywordstyleMPI_COMM_WORLD);
         tag = 1;
          keywordstyleMPI_Send(b, 1,  keywordstyleMPI_DOUBLE, dest, tag,
                 keywordstyleMPI_COMM_WORLD);
         tag = 2;
          keywordstyleMPI_Send(n, 1,  keywordstyleMPI_INT, dest, tag,
                 keywordstyleMPI_COMM_WORLD);
      }
   }
```

```
   else {
      tag = 0;
       keywordstyleMPI_Recv(a, 1,  keywordstyleMPI_DOUBLE, source, tag,
              keywordstyleMPI_COMM_WORLD, &status);
      tag = 1;
       keywordstyleMPI_Recv(b, 1,  keywordstyleMPI_DOUBLE, source, tag,
              keywordstyleMPI_COMM_WORLD, &status);
      tag = 2;
       keywordstyleMPI_Recv(n, 1,  keywordstyleMPI_INT, source, tag,
              keywordstyleMPI_COMM_WORLD, &status);
   }
}
```

Now the parallel program with input is

```
#include "mpi.h"
#include <stdio.h>
extern void Get_data(double *a_ptr, double *b_ptr, int *n_ptr, int myrank,
         int p);
extern double Trap(double a, double b, int n, double h);
int main(int argc, char **argv) {
   int myrank, p;
   double a, b, h;
   int n;
   double local_a, local_b;
   int local_n;
   double integral;
   double total = -1;
   int source, dest = 0, tag = 0;
    keywordstyleMPI_Status status;
    keywordstyleMPI_Init(&argc, &argv);
    keywordstyleMPI_Comm_rank( keywordstyleMPI_COMM_WORLD, &myrank);
    keywordstyleMPI_Comm_size( keywordstyleMPI_COMM_WORLD, &p);
```

```
   Get_data(&a, &b, &n, myrank, p);
   h = (b - a) / n;
   local_n = n / p;
   local_a = a + myrank * local_n * h;
   local_b = local_a + local_n * h;
   integral = Trap(local_a, local_b, local_n, h);
   if (myrank == 0) {
      total = integral;
      for (source = 1; source < p; source++) {
          keywordstyleMPI_Recv(&integral, 1,
                 keywordstyleMPI_DOUBLE, source, tag,
                 keywordstyleMPI_COMM_WORLD, &status);
         total = total + integral;
      }
   }
   else
       keywordstyleMPI_Send(&integral, 1,
              keywordstyleMPI_DOUBLE, dest, tag,
              keywordstyleMPI_COMM_WORLD);
   if (myrank == 0) {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %f\n", a, b, total);
   }
    keywordstyleMPI_Finalize();
   return 0;
}
```

Makefile is

```
CC = mpicc
CFLAGS = -Wall -O2
OBJECTS1 = func.o traprule.o trap.o
OBJECTS2 = func.o traprule.o iotrap.o getdata.o
all: partrap iopartrap
partrap: $(OBJECTS1)
  mpicc -o $@ $?
iopartrap: $(OBJECTS2)
  mpicc -o $@ $?
clean:
  rm *.o partrap iopartrap
```

Summary

One can write many parallel programs using only

```
keywordstyleMPI_Init
keywordstyleMPI_Comm_rank
keywordstyleMPI_Comm_size
keywordstyleMPI_Send
keywordstyleMPI_Recv
keywordstyleMPI_Finalize
```