**b Program 1**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* FILE: mpi\_array.c

\* DESCRIPTION:

\* MPI Example - Array Assignment - C Version

\* This program demonstrates a simple data decomposition. The master task

\* first initializes an array and then distributes an equal portion that

\* array to the other tasks. After the other tasks receive their portion

\* of the array, they perform an addition operation to each array element.

\* They also maintain a sum for their portion of the array. The master task

\* does likewise with its portion of the array. As each of the non-master

\* tasks finish, they send their updated portion of the array to the master.

\* An MPI collective communication call is used to collect the sums

\* maintained by each task. Finally, the master task displays selected

\* parts of the final array and the global sum of all array elements.

\* NOTE: the number of MPI tasks must be evenly divided by 4.

\* AUTHOR: Blaise Barney

\* LAST REVISED: 04/13/05

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "mpi.h"

#include <stdio.h>

#include <stdlib.h>

#define ARRAYSIZE 16000000

#define MASTER 0

float data[ARRAYSIZE];

int main (int argc, char \*argv[])

{

int numtasks, taskid, rc, dest, offset, i, j, tag1,

tag2, source, chunksize;

float mysum, sum;

float update(int myoffset, int chunk, int myid);

MPI\_Status status;

/\*\*\*\*\* Initializations \*\*\*\*\*/

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numtasks);

if (numtasks % 4 != 0) {

printf("Quitting. Number of MPI tasks must be divisible by 4.\n");

MPI\_Abort(MPI\_COMM\_WORLD, rc);

exit(0);

}

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&taskid);

printf ("MPI task %d has started...\n", taskid);

chunksize = (ARRAYSIZE / numtasks);

tag2 = 1;

tag1 = 2;

/\*\*\*\*\* Master task only \*\*\*\*\*\*/

if (taskid == MASTER){

/\* Initialize the array \*/

sum = 0;

for(i=0; i<ARRAYSIZE; i++) {

data[i] = i \* 1.0;

sum = sum + data[i];

}

printf("Initialized array sum = %e\n",sum);

/\* Send each task its portion of the array - master keeps 1st part \*/

offset = chunksize;

for (dest=1; dest<numtasks; dest++) {

MPI\_Send(&offset, 1, MPI\_INT, dest, tag1, MPI\_COMM\_WORLD);

MPI\_Send(&data[offset], chunksize, MPI\_FLOAT, dest, tag2, MPI\_COMM\_WORLD);

printf("Sent %d elements to task %d offset= %d\n",chunksize,dest,offset);

offset = offset + chunksize;

}

/\* Master does its part of the work \*/

offset = 0;

mysum = update(offset, chunksize, taskid);

/\* Wait to receive results from each task \*/

for (i=1; i<numtasks; i++) {

source = i;

MPI\_Recv(&offset, 1, MPI\_INT, source, tag1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&data[offset], chunksize, MPI\_FLOAT, source, tag2,

MPI\_COMM\_WORLD, &status);

}

/\* Get final sum and print sample results \*/

MPI\_Reduce(&mysum, &sum, 1, MPI\_FLOAT, MPI\_SUM, MASTER, MPI\_COMM\_WORLD);

printf("Sample results: \n");

offset = 0;

for (i=0; i<numtasks; i++) {

for (j=0; j<5; j++)

printf(" %e",data[offset+j]);

printf("\n");

offset = offset + chunksize;

}

printf("\*\*\* Final sum= %e \*\*\*\n",sum);

} /\* end of master section \*/

/\*\*\*\*\* Non-master tasks only \*\*\*\*\*/

if (taskid > MASTER) {

/\* Receive my portion of array from the master task \*/

source = MASTER;

MPI\_Recv(&offset, 1, MPI\_INT, source, tag1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&data[offset], chunksize, MPI\_FLOAT, source, tag2,

MPI\_COMM\_WORLD, &status);

mysum = update(offset, chunksize, taskid);

/\* Send my results back to the master task \*/

dest = MASTER;

MPI\_Send(&offset, 1, MPI\_INT, dest, tag1, MPI\_COMM\_WORLD);

MPI\_Send(&data[offset], chunksize, MPI\_FLOAT, MASTER, tag2, MPI\_COMM\_WORLD);

MPI\_Reduce(&mysum, &sum, 1, MPI\_FLOAT, MPI\_SUM, MASTER, MPI\_COMM\_WORLD);

} /\* end of non-master \*/

MPI\_Finalize();

} /\* end of main \*/

float update(int myoffset, int chunk, int myid) {

int i;

float mysum;

/\* Perform addition to each of my array elements and keep my sum \*/

mysum = 0;

for(i=myoffset; i < myoffset + chunk; i++) {

data[i] = data[i] + i \* 1.0;

mysum = mysum + data[i];

}

printf("Task %d mysum = %e\n",myid,mysum);

return(mysum);

}

**Program 2**

**RING\_MPI** is a C program which estimates the time it takes to send a vector of N double precision values through each process in a ring.

Process 0 sends N double precision values to process 1, which passes them to process 2, and so on, until process P-1 sends them back to process 0. The time for this transmission is recorded. The experiment is repeated several times, and for several different array sizes N.

# include <stdlib.h>

# include <stdio.h>

# include "mpi.h"

int main ( int argc, char \*argv[] );

void ring\_io ( int p, int id );

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main ( int argc, char \*argv[] )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

MAIN is the main program for RING\_MPI.

Discussion:

RING\_MPI sends messages of various size from process 0 to 1 to 2 to

...the last process and then back to 0.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

21 October 2012

Author:

John Burkardt

Reference:

Peter Pacheco,

Parallel Programming with MPI,

Morgan Kaufman, 1996,

ISBN: 1558603395,

LC: QA76.642.P3.

\*/

{

int error;

int id;

int p;

/\*

Initialize MPI.

\*/

MPI\_Init ( &argc, &argv );

/\*

Get the number of processes.

\*/

MPI\_Comm\_size ( MPI\_COMM\_WORLD, &p );

/\*

Get the individual process ID.

\*/

MPI\_Comm\_rank ( MPI\_COMM\_WORLD, &id );

/\*

Print a message.

\*/

if ( id == 0 )

{

printf ( "\n" );

printf ( "RING\_MPI:\n" );

printf ( " C/MPI version\n" );

printf ( " Measure time required to transmit data around\n" );

printf ( " a ring of processes\n" );

printf ( "\n" );

printf ( " The number of processes is %d\n", p );

}

ring\_io ( p, id );

/\*

Shut down MPI.

\*/

MPI\_Finalize ( );

/\*

Terminate.

\*/

if ( id == 0 )

{

printf ( "\n" );

printf ( "RING\_MPI:\n" );

printf ( " Normal end of execution.\n" );

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void ring\_io ( int p, int id )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

RING\_IO carries out the tasks of process ID, of a total of P processes.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

21 October 2012

Author:

John Burkardt

Reference:

Peter Pacheco,

Parallel Programming with MPI,

Morgan Kaufman, 1996,

ISBN: 1558603395,

LC: QA76.642.P3.

\*/

{

int dest;

int i;

int j;

int n;

int n\_test[5] = { 100, 1000, 10000, 100000, 1000000 };

int n\_test\_num = 5;

int source;

MPI\_Status status;

double tave;

int test;

int test\_num = 10;

double tmax;

double tmin;

double wtime;

double \*x;

if ( id == 0 )

{

printf ( "\n" );

printf ( " Timings based on %d experiments\n", test\_num );

printf ( " N double precision values were sent\n" );

printf ( " in a ring transmission starting and ending at process 0\n" );

printf ( " and using a total of %d processes.\n", p );

printf ( "\n" );

printf ( " N T min T ave T max\n" );

printf ( "\n" );

}

/\*

Choose message size.

\*/

for ( i = 0; i < n\_test\_num; i++ )

{

n = n\_test[i];

x = ( double \* ) malloc ( n \* sizeof ( double ) );

/\*

Process 0 sends very first message,

then waits to receive the "echo" that has gone around the world.

\*/

if ( id == 0 )

{

dest = 1;

source = p - 1;

tave = 0.0;

tmin = 1.0E+30;

tmax = 0.0;

for ( test = 1; test <= test\_num; test++ )

{

/\*

Just in case, set the entries of X in a way that identifies

which iteration of the test is being carried out.

\*/

for ( j = 0; j < n; j++ )

{

x[j] = ( double ) ( test + j );

}

wtime = MPI\_Wtime ( );

MPI\_Send ( x, n, MPI\_DOUBLE, dest, 0, MPI\_COMM\_WORLD );

MPI\_Recv ( x, n, MPI\_DOUBLE, source, 0, MPI\_COMM\_WORLD, &status );

wtime = MPI\_Wtime ( ) - wtime;

/\*

Record the time it took.

\*/

tave = tave + wtime;

if ( wtime < tmin )

{

tmin = wtime;

}

if ( wtime > tmax )

{

tmax = wtime;

}

}

tave = tave / ( double ) ( test\_num );

printf ( " %8d %14.6g %14.6g %14.6g\n", n, tmin, tave, tmax );

}

/\*

Worker ID must receive first from ID-1, then send to ID+1.

\*/

else

{

source = id - 1;

dest = ( ( id + 1 ) % p );

for ( test = 1; test <= test\_num; test++ )

{

MPI\_Recv ( x, n, MPI\_DOUBLE, source, 0, MPI\_COMM\_WORLD, &status );

MPI\_Send ( x, n, MPI\_DOUBLE, dest, 0, MPI\_COMM\_WORLD );

}

}

free ( x );

}

return;

}

**Program 3**

**SEARCH\_MPI** is a C program which searches integers between A and B for a value J such that F(J) = C, using the MPI parallel programming environment.

For the particular example considered here, the solution will be F(1,674,924,981) = 45.

# include <stdlib.h>

# include <stdio.h>

# include <time.h>

# include "mpi.h"

int main ( int argc, char \*argv[] );

int search ( int a, int b, int c, int id, int p );

int f ( int i );

void timestamp ( void );

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main ( int argc, char \*argv[] )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

MAIN is the main program for SEARCH\_MPI.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

01 November 2012

Author:

John Burkardt

\*/

{

int a;

int b;

int c;

int fj;

int i4\_huge = 2147483647;

int id;

int j;

int p;

double wtime;

/\*

Initialize MPI.

\*/

MPI\_Init ( &argc, &argv );

/\*

Get this processor's ID.

\*/

MPI\_Comm\_rank ( MPI\_COMM\_WORLD, &id );

/\*

Get the number of processes.

\*/

MPI\_Comm\_size ( MPI\_COMM\_WORLD, &p );

a = 1;

b = i4\_huge;

c = 45;

if ( id == 0 )

{

timestamp ( );

printf ( "\n" );

printf ( "SEARCH\_MPI:\n" );

printf ( " C/MPI version\n" );

printf ( " Search the integers from A to B\n" );

printf ( " for a value J such that F(J) = C.\n" );

printf ( "\n" );

printf ( " A = %d\n", a );

printf ( " B = %d\n", b );

printf ( " C = %d\n", c );

}

wtime = MPI\_Wtime ( );

j = search ( a, b, c, id, p );

wtime = MPI\_Wtime ( ) - wtime;

if ( j != -1 )

{

printf ( "\n" );

printf ( " Process %d found J = %d\n", id, j );

printf ( " Verify F(J) = %d\n", f ( j ) );

}

if ( id == 0 )

{

printf ( " Elapsed wallclock time is %g\n", wtime );

}

/\*

Terminate MPI.

\*/

MPI\_Finalize ( );

/\*

Terminate.

\*/

if ( id == 0 )

{

printf ( "\n" );

printf ( "SEARCH\_MPI:\n" );

printf ( " Normal end of execution.\n" );

printf ( "\n" );

timestamp ( );

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int search ( int a, int b, int c, int id, int p )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

SEARCH searches integers in [A,B] for a J so that F(J) = C.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

01 November 2012

Author:

John Burkardt

Parameters:

Input, int A, B, the search range.

Input, int C, the desired function value.

Input, int ID, the process ID.

Input, int P, the number of processes.

Output, int SEARCH, the computed solution, or -1

if no solution was found.

\*/

{

int fi;

int i;

int j;

j = -1;

/\*

i = i + p can take us "over top" so that i becomes negative!

So we have to be more careful here!

\*/

for ( i = a + id; 0 < i && i <= b; i = i + p )

{

fi = f ( i );

if ( fi == c )

{

j = i;

break;

}

}

return j;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int f ( int i )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

F is the function we are analyzing.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

22 October 2012

Author:

John Burkardt

Parameters:

Input, int I, the argument.

Input, int F, the value.

\*/

{

int i4\_huge = 2147483647;

int j;

int k;

int value;

value = i;

for ( j = 1; j <= 5; j++ )

{

k = value / 127773;

value = 16807 \* ( value - k \* 127773 ) - k \* 2836;

if ( value <= 0 )

{

value = value + i4\_huge;

}

}

return value;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void timestamp ( void )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

TIMESTAMP prints the current YMDHMS date as a time stamp.

Example:

31 May 2001 09:45:54 AM

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

24 September 2003

Author:

John Burkardt

Parameters:

None

\*/

{

# define TIME\_SIZE 40

static char time\_buffer[TIME\_SIZE];

const struct tm \*tm;

time\_t now;

now = time ( NULL );

tm = localtime ( &now );

strftime ( time\_buffer, TIME\_SIZE, "%d %B %Y %I:%M:%S %p", tm );

printf ( "%s\n", time\_buffer );

return;

# undef TIME\_SIZE

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double cpu\_time ( void )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

CPU\_TIME returns the current reading on the CPU clock.

Discussion:

The CPU time measurements available through this routine are often

not very accurate. In some cases, the accuracy is no better than

a hundredth of a second.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

06 June 2005

Author:

John Burkardt

Parameters:

Output, double CPU\_TIME, the current reading of the CPU clock, in seconds.

\*/

{

double value;

value = ( double ) clock ( )

/ ( double ) CLOCKS\_PER\_SEC;

return value;

}

**Program 4**

**MULTITASK\_MPI** is a C program which demonstrates how to "multitask", that is, to execute several unrelated and distinct tasks simultaneously, using MPI for parallel execution.

In this example, there is a "master" process, identified as process 0, and two worker processes, 1 and 2. Process 0 does nothing but choose the input for the worker processes, transmit it, and wait for the computed results to be returned. The programs executed by process 1 and by process 2 are quite different.

While the typical MPI model has all the worker processes executing the same program, this example shows that that is not necessary.

# include <stdio.h>

# include <stdlib.h>

# include <math.h>

# include <time.h>

# include "mpi.h"

int main ( int argc, char \*argv[] );

void p0\_set\_input ( int \*input1, int \*input2 );

void p0\_send\_input ( int input1, int input2 );

void p0\_receive\_output ( int \*output1, int \*output2 );

int p1\_receive\_input ( void );

int p1\_compute\_output ( int input1 );

void p1\_send\_output ( int output1 );

int p2\_receive\_input ( void );

int p2\_compute\_output ( int input2 );

void p2\_send\_output ( int output2 );

void timestamp ( void );

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main ( int argc, char \*argv[] )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

MAIN is the main program for MPI\_MULTITASK.

Discussion:

Message tag 1: P0 sends input to P1

Message tag 2: P0 sends input to P2

Message tag 3: P1 sends output to P0.

Message tag 4: P2 sends output to P0.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

21 October 2011

Author:

John Burkardt

\*/

{

int id;

int ierr;

int input1;

int input2;

int output1;

int output2;

int p;

double wtime;

/\*

Process 0 is the "monitor".

It chooses the inputs, and sends them to the workers.

It waits for the outputs.

It plots the outputs.

\*/

ierr = MPI\_Init ( &argc, &argv );

ierr = MPI\_Comm\_rank ( MPI\_COMM\_WORLD, &id );

ierr = MPI\_Comm\_size ( MPI\_COMM\_WORLD, &p );

/\*

Make sure we have enough processes.

\*/

if ( p < 3 )

{

printf ( "\n" );

printf ( "MPI\_MULTITASK - Fatal error!\n" );

printf ( " Number of available processes must be at least 3!\n" );

ierr = MPI\_Finalize ( );

exit ( 1 );

}

/\*

Run program P0 on process 0, and so on.

\*/

if ( id == 0 )

{

timestamp ( );

printf ( "\n" );

printf ( "MPI\_MULTITASK:\n" );

printf ( " C / MPI version\n" );

wtime = MPI\_Wtime ( );

p0\_set\_input ( &input1, &input2 );

p0\_send\_input ( input1, input2 );

p0\_receive\_output ( &output1, &output2 );

wtime = MPI\_Wtime ( ) - wtime;

printf ( " Process 0 time = %g\n", wtime );

ierr = MPI\_Finalize ( );

printf ( "\n" );

printf ( "MPI\_MULTITASK:\n" );

printf ( " Normal end of execution.\n" );

timestamp ( );

}

/\*

Process 1 works on task 1.

It receives input from process 0.

It computes the output.

It sends the output to process 0.

\*/

else if ( id == 1 )

{

wtime = MPI\_Wtime ( );

input1 = p1\_receive\_input ( );

output1 = p1\_compute\_output ( input1 );

p1\_send\_output ( output1 );

wtime = MPI\_Wtime ( ) - wtime;

printf ( " Process 1 time = %g\n", wtime );

ierr = MPI\_Finalize ( );

}

/\*

Process 2 works on task 2.

It receives input from process 0.

It computes the output.

It sends the output to process 0.

\*/

else if ( id == 2 )

{

wtime = MPI\_Wtime ( );

input2 = p2\_receive\_input ( );

output2 = p2\_compute\_output ( input2 );

p2\_send\_output ( output2 );

wtime = MPI\_Wtime ( ) - wtime;

printf ( " Process 2 time = %g\n", wtime );

ierr = MPI\_Finalize ( );

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void p0\_set\_input ( int \*input1, int \*input2 )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

P0\_SET\_INPUT sets input.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

21 October 2011

Author:

John Burkardt

Parameters:

Output, int \*INPUT1, \*INPUT2, the values of two

inputs used by tasks 1 and 2.

\*/

{

\*input1 = 10000000;

\*input2 = 100000;

printf ( "\n" );

printf ( "P0\_SET\_PARAMETERS:\n" );

printf ( " Set INPUT1 = %d\n", \*input1 );

printf ( " INPUT2 = %d\n", \*input2 );

return;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void p0\_send\_input ( int input1, int input2 )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

P0\_SEND\_INPUT sends input to processes 1 and 2.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

21 October 2011

Author:

John Burkardt

Parameters:

Input, int INPUT1, INPUT2, the values of two

inputs used by tasks 1 and 2.

\*/

{

int id;

int ierr;

int tag;

id = 1;

tag = 1;

ierr = MPI\_Send ( &input1, 1, MPI\_INT, id, tag, MPI\_COMM\_WORLD );

id = 2;

tag = 2;

ierr = MPI\_Send ( &input2, 1, MPI\_INT, id, tag, MPI\_COMM\_WORLD );

return;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void p0\_receive\_output ( int \*output1, int \*output2 )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

P0\_RECEIVE\_OUTPUT receives output from processes 1 and 2.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

21 October 2011

Author:

John Burkardt

Parameters:

Output, int OUTPUT1, OUTPUT2, the values of the

outputs of tasks 1 and 2.

\*/

{

int ierr;

int output;

int output\_received;

int source;

MPI\_Status status;

output\_received = 0;

/\*

Loop until every worker has checked in.

\*/

while ( output\_received < 2 )

{

/\*

Receive the next message that arrives.

\*/

ierr = MPI\_Recv ( &output, 1, MPI\_INT, MPI\_ANY\_SOURCE, MPI\_ANY\_TAG,

MPI\_COMM\_WORLD, &status );

/\*

The actual source of the message is saved in STATUS.

\*/

source = status.MPI\_SOURCE;

/\*

Save the value in OUTPUT1 or OUTPUT2.

\*/

if ( source == 1 )

{

\*output1 = output;

}

else

{

\*output2 = output;

}

output\_received = output\_received + 1;

}

printf ( "\n" );

printf ( " Process 1 returned OUTPUT1 = %d\n", \*output1 );

printf ( " Process 2 returned OUTPUT2 = %d\n", \*output2 );

return;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int p1\_receive\_input ( void )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

P1\_RECEIVE\_INPUT receives input from process 0.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

21 October 2011

Author:

John Burkardt

Parameters:

Output, int P1\_RECEIVE\_INPUT, the value of the parameter.

\*/

{

int id;

int ierr;

int input1;

MPI\_Status status;

int tag;

id = 0;

tag = 1;

ierr = MPI\_Recv ( &input1, 1, MPI\_INT, id, tag, MPI\_COMM\_WORLD, &status );

return input1;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int p1\_compute\_output ( int input1 )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

P1\_COMPUTE\_OUTPUT carries out computation number 1.

Discussion:

No MPI calls occur in this function.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

21 October 2011

Author:

John Burkardt

Parameters:

Input, int INPUT1, the problem input.

Output, int P1\_COMPUTE\_OUTPUT1, the problem output.

\*/

{

int i;

int j;

int k;

int output1;

output1 = 0;

for ( i = 2; i <= input1; i++ )

{

j = i;

k = 0;

while ( 1 < j )

{

if ( ( j % 2 ) == 0 )

{

j = j / 2;

}

else

{

j = 3 \* j + 1;

}

k = k + 1;

}

if ( output1 < k )

{

output1 = k;

}

}

return output1;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void p1\_send\_output ( int output1 )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

P1\_SEND\_OUTPUT sends output to process 0.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

21 October 2011

Author:

John Burkardt

Parameters:

Input, int OUTPUT1, the problem output.

\*/

{

int id;

int ierr;

int tag;

id = 0;

tag = 3;

ierr = MPI\_Send ( &output1, 1, MPI\_INT, id, tag, MPI\_COMM\_WORLD );

return;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int p2\_receive\_input ( void )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

P2\_RECEIVE\_INPUT receives input from process 0.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

21 October 2011

Author:

John Burkardt

Parameters:

Output, int P2\_RECEIVE\_INPUT, the value of the parameter.

\*/

{

int id;

int ierr;

int input2;

MPI\_Status status;

int tag;

id = 0;

tag = 2;

ierr = MPI\_Recv ( &input2, 1, MPI\_INT, id, tag, MPI\_COMM\_WORLD, &status );

return input2;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int p2\_compute\_output ( int input2 )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

P2\_COMPUTE\_OUTPUT carries out computation number 2.

Discussion:

No MPI calls occur in this function.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

20 October 2011

Author:

John Burkardt

Parameters:

Input, int INPUT2, the problem input.

Output, int P2\_COMPUTE\_OUTPUT, the problem output.

\*/

{

int i;

int j;

int output2;

int prime;

output2 = 0;

for ( i = 2; i <= input2; i++ )

{

prime = 1;

for ( j = 2; j < i; j++ )

{

if ( ( i % j ) == 0 )

{

prime = 0;

break;

}

}

if ( prime )

{

output2 = output2 + 1;

}

}

return output2;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void p2\_send\_output ( int output2 )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

P2\_SEND\_OUTPUT sends output to process 0.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

21 October 2011

Author:

John Burkardt

Parameters:

Input, int OUTPUT2, the problem output.

\*/

{

int id;

int ierr;

int tag;

id = 0;

tag = 4;

ierr = MPI\_Send ( &output2, 1, MPI\_INT, id, tag, MPI\_COMM\_WORLD );

return;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void timestamp ( void )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

TIMESTAMP prints the current YMDHMS date as a time stamp.

Example:

31 May 2001 09:45:54 AM

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

24 September 2003

Author:

John Burkardt

Parameters:

None

\*/

{

# define TIME\_SIZE 40

static char time\_buffer[TIME\_SIZE];

const struct tm \*tm;

time\_t now;

now = time ( NULL );

tm = localtime ( &now );

strftime ( time\_buffer, TIME\_SIZE, "%d %B %Y %I:%M:%S %p", tm );

printf ( "%s\n", time\_buffer );

return;

# undef TIME\_SIZE

}

**Program 5**

**PRIME\_MPI** is a C program which counts the number of primes between 1 and N, using MPI to carry out the calculation in parallel.

The algorithm is completely naive. For each integer I, it simply checks whether any smaller J evenly divides it. The total amount of work for a given N is thus roughly proportional to 1/2\*N^2.

This program is mainly a starting point for investigations into parallelization.

# include <stdlib.h>

# include <stdio.h>

# include <math.h>

# include <time.h>

# include "mpi.h"

int main ( int argc, char \*argv[] );

int prime\_number ( int n, int id, int p );

void timestamp ( void );

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main ( int argc, char \*argv[] )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

MAIN is the main program for PRIME\_MPI.

Discussion:

This program calls a version of PRIME\_NUMBER that includes

MPI calls for parallel processing.

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

07 August 2009

Author:

John Burkardt

\*/

{

int i;

int id;

int ierr;

int n;

int n\_factor;

int n\_hi;

int n\_lo;

int p;

int primes;

int primes\_part;

double wtime;

n\_lo = 1;

n\_hi = 262144;

n\_factor = 2;

/\*

Initialize MPI.

\*/

ierr = MPI\_Init ( &argc, &argv );

/\*

Get the number of processes.

\*/

ierr = MPI\_Comm\_size ( MPI\_COMM\_WORLD, &p );

/\*

Determine this processes's rank.

\*/

ierr = MPI\_Comm\_rank ( MPI\_COMM\_WORLD, &id );

if ( id == 0 )

{

timestamp ( );

printf ( "\n" );

printf ( "PRIME\_MPI\n" );

printf ( " C/MPI version\n" );

printf ( "\n" );

printf ( " An MPI example program to count the number of primes.\n" );

printf ( " The number of processes is %d\n", p );

printf ( "\n" );

printf ( " N Pi Time\n" );

printf ( "\n" );

}

n = n\_lo;

while ( n <= n\_hi )

{

if ( id == 0 )

{

wtime = MPI\_Wtime ( );

}

ierr = MPI\_Bcast ( &n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD );

primes\_part = prime\_number ( n, id, p );

ierr = MPI\_Reduce ( &primes\_part, &primes, 1, MPI\_INT, MPI\_SUM, 0,

MPI\_COMM\_WORLD );

if ( id == 0 )

{

wtime = MPI\_Wtime ( ) - wtime;

printf ( " %8d %8d %14f\n", n, primes, wtime );

}

n = n \* n\_factor;

}

/\*

Terminate MPI.

\*/

ierr = MPI\_Finalize ( );

/\*

Terminate.

\*/

if ( id == 0 )

{

printf ( "\n");

printf ( "PRIME\_MPI - Master process:\n");

printf ( " Normal end of execution.\n");

printf ( "\n" );

timestamp ( );

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int prime\_number ( int n, int id, int p )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

PRIME\_NUMBER returns the number of primes between 1 and N.

Discussion:

In order to divide the work up evenly among P processors, processor

ID starts at 2+ID and skips by P.

A naive algorithm is used.

Mathematica can return the number of primes less than or equal to N

by the command PrimePi[N].

N PRIME\_NUMBER

1 0

10 4

100 25

1,000 168

10,000 1,229

100,000 9,592

1,000,000 78,498

10,000,000 664,579

100,000,000 5,761,455

1,000,000,000 50,847,534

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

21 May 2009

Author:

John Burkardt

Parameters:

Input, int N, the maximum number to check.

Input, int ID, the ID of this process,

between 0 and P-1.

Input, int P, the number of processes.

Output, int PRIME\_NUMBER, the number of prime numbers up to N.

\*/

{

int i;

int j;

int prime;

int total;

total = 0;

for ( i = 2 + id; i <= n; i = i + p )

{

prime = 1;

for ( j = 2; j < i; j++ )

{

if ( ( i % j ) == 0 )

{

prime = 0;

break;

}

}

total = total + prime;

}

return total;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void timestamp ( void )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Purpose:

TIMESTAMP prints the current YMDHMS date as a time stamp.

Example:

31 May 2001 09:45:54 AM

Licensing:

This code is distributed under the GNU LGPL license.

Modified:

24 September 2003

Author:

John Burkardt

Parameters:

None

\*/

{

# define TIME\_SIZE 40

static char time\_buffer[TIME\_SIZE];

const struct tm \*tm;

size\_t len;

time\_t now;

now = time ( NULL );

tm = localtime ( &now );

len = strftime ( time\_buffer, TIME\_SIZE, "%d %B %Y %I:%M:%S %p", tm );

printf ( "%s\n", time\_buffer );

return;

# undef TIME\_SIZE

}