

# Distributed Memory Programming With MPI Exercises

John Burkardt  
Interdisciplinary Center for Applied Mathematics &  
Information Technology Department  
Virginia Tech

.....  
Advanced Computational Science II  
Department of Scientific Computing  
Florida State University

.....  
<http://people.sc.fsu.edu/~jburkardt/presentations/...>  
... mpi\_exercises\_2008\_acs2.pdf

22 September 2008

This hands on session introduces MPI, which can be used to write parallel programs on distributed memory systems.

We will use the HPC system to do our computation. This means that in general, you have to log in to a remote system, transfer your program there, compile it, and then submit a request to the queuing system to have your job run.

I will show you how you can do some preliminary debugging on your local system, by using a "stub" library.

The exercises include:

1. HELLO: compile and run a simple program;
2. QUAD: estimate an integral using quadrature;
3. MC: estimate a multidimensional integral using the Monte Carlo method;
4. PRIME: sum the prime numbers from 2 to N
5. HEAT: set up the 1D heat equation for distributed memory

For most of the exercises, there is a source code program that you can use. The source code is generally available in a C, C++, FORTRAN77 and FORTRAN90 version, so you can stick with your favorite language.

At the end of these exercises, you are to work on an assignment **to be handed in by Friday**, involving the heat equation.

## 1 HELLO: Compile and run a simple program

Get a copy of the **hello** program and the shell script that submits it. One location is:

[http://www.scs.fsu.edu/~burkardt/vt2/fsu\\_mpi\\_2008/hello/hello.html](http://www.scs.fsu.edu/~burkardt/vt2/fsu_mpi_2008/hello/hello.html)

This program does almost nothing, so it shows you the bare minimum you need to get an MPI program running, including:

- the invocation of the “include” file.
- the initialization call
- How you find out how many processes are available.
- How you find out the ID of your process
- How the process ID can be used to control your actions.
- the finalization call

## 1.1 Compile HELLO with a “stub” library

If you try to compile your *hello* program, you will probably get error messages. Why?

Possibly because the compiler can’t find the include file you are asking for, and the include file is necessary to define the meaning of the symbolic constants, and the type of the MPI functions.

However, if the compiler supports MPI, then the complains will come at link/load time, when it tries to create the executable. At that point, it is very possible that it cannot find a library containing the MPI functions.

This is a shame, since it would be very convenient, for debugging, to be able to compile our programs on a simple local machine, just to check for simple errors. But if we really don’t need to run the program locally, then all we need is an include file and a library that are good enough to keep the compiler and loader/linker happy.

This is sometimes called a “stub” library, since it contains hollowed out versions of MPI functions, just the name and a **return** statement.

To see what I am talking about, try to compile your hello program on your local machine. For instance, if you are using *hello.c*, then depending on what compiler is available, try one of the following: the commands

```
cc hello.c
gcc hello.c
icc hello.c
```

If the compiler exists, it will try to compile your program, and it will probably complain.

Now get a copy of the corresponding MPI stub library, from [Get a copy of the \*\*hello\*\* programs](http://www.scs.fsu.edu/~burkardt/vt2/fsu_mpi_2008/mpi_stubs/mpi_stubs.html). One location is:

```
http://www.scs.fsu.edu/~burkardt/vt2/fsu\_mpi\_2008/mpi\_stubs/mpi\_stubs.html
```

Add the stub library to your compile statement. For instance:

```
cc hello.c mpi_stubs.c
```

I believe your program should compile, load and, (only because the hello program is so simple), even run!

## 1.2 Compile HELLO with the real MPI library

After MPI.STUBS has helped you to diagnose and fix any compiler bugs while using your local system, it’s time to work with the real MPI library. This requires that you have an account on the FSU HPC system, that you transfer the file to that system using **sftp**, and that you login interactively to that system using **ssh**.

Once you’re logged in, you want to compile your program with the MPI library. The FSU HPC system includes several choices for the compiler, and several “flavors” of MPI. For our work, we want to choose the GNU compilers, and OpenMPI. To do this, we must issue the command

```
source /usr/local/profile.d/openmpi-gnu.sh
```

You can only compile after issuing this command!

You can make life simpler by using some kind of alias, which you would put in your **.bash\_profile** file, such as:

```
alias mpi_setup "source /usr/local/profile.d/openmpi-gnu.sh"
```

Then you just type **mpi\_setup** when you want to use MPI.

In any case, when you have set up MPI, try one of the commands:

```
mpicc hello.c
mpiCC hello.cc
mpif77 hello.f
mpif90 hello.f90
```

Rename your executable program to **hello**. Especially when you use a batch system to run programs, it's important to distinguish your programs by name.

```
mv a.out hello
```

### 1.3 Run HELLO using the Queueing System

From now on, all our programs will be run using the batch system. That means that we compile the program, give it a name, maybe put the program and its data in a special directory, and then send a request to the batch system. Then we wait until our request is processed.

Copy the script that submits the hello program. One location is:

```
http://www.scs.fsu.edu/~burkardt/vt2/fsu\_mpi\_2008/hello/hello.html
```

Take a look at the script. There is almost nothing you have to change. The interesting features include

- something that controls the total time allowed;
- two items that request a number of cpus, and declare (the same number) of OpenMP threads;
- a statement that actually runs your program;

Submit the script

```
msub hello.sh
```

You can issue one of the commands

```
showq
showq -u your-username-here
showq | grep your-username-here
```

to see the status of your job (unless it's already run.)

Congratulations! Now we can try the harder stuff.

## 2 QUAD: Estimate an integral using quadrature

One approximation to the integral  $\int_a^b f(x) dx$  uses an average of function values at equally spaced points. This is a natural application for parallel processing. Each process can work in a subinterval, and at the end the partial sums can be added together to get the integral estimate.

The exact value of the integral is  $\pi$ .

Get a copy of the **quad** program and the shell script that submits it. One location is:

[http://www.scs.fsu.edu/~burkardt/vt2/fsu\\_mpi\\_2008/quad/quad.html](http://www.scs.fsu.edu/~burkardt/vt2/fsu_mpi_2008/quad/quad.html)

By the way, you should be able to compile and run this program sequentially, using the **MPLSTUBS** library, if you want to try that. (The C++ version of the library is incomplete; and if you are using FORTRAN90 then you want to replace

```
use mpi
```

by

```
include 'mpif.h'
```

### 2.1 Run QUAD with MPI

Copy the program to the HPC system, compile it, name the executable **quad**, and then submit the shell script that runs the job. Examine the output file and make sure you are satisfied with the results.

### 2.2 Run QUAD, varying the number of processes

Fix the amount of work, **N**, to be 1,000,000. Then make a series of runs with 1, 2, 4, and 8 processes, call the elapsed times T1, T2, T4 and T8, and compute the ratios of T2/T1, T4/T1, and T8/T1.

P	Time	T / T1
--	-----	-----
1	-----	1.00
2	-----	-----
4	-----	-----
8	-----	-----

Do your timing results suggest that this program is using MPI efficiently?

### 2.3 Run QUAD, varying the amount of work

Fix the number of processors to 4. Then make a series of runs with increasing values of **N**, the number of evaluation points. This value was 1000 in the original version of the program. What is the behavior of the error? In particular, as we multiply **N** by 10, what happens to the error?

N	Error	E / E1000
-----	-----	-----
1000	-----	1.00
10000	-----	-----
100000	-----	-----
1000000	-----	-----
10000000	-----	-----

For a well behaved integrand function  $f(x)$ , you might expect the error to be reduced by a factor of 10 when **N** is multiplied by 10. Can you suggest why your results do not behave this way?

### 3 MC: Estimate an integral using the Monte Carlo method

We are going to write *another* integration program, this time using the Monte Carlo method in multiple dimensions.

#### 3.1 Run the MC program sequentially

You can do this on the local system. The program evaluates a function  $f(x)$  at  $N$  “random” points in the  $m$ -dimensional unit hypercube, and averages the values to estimate the integral. It knows the correct value of the integral, so it also prints the error in the estimate.

The program is set up to use a value of 1,000 for  $N$ . Run the program for an increasing number of  $N$  values and consider how the error decreases.

N	Error	E / E1000
-----	-----	-----
1000	-----	1.00
10000	-----	-----
100000	-----	-----
1000000	-----	-----
10000000	-----	-----

#### 3.2 Make a Parallel Version of MC

Make an MPI version of MC. Since the **QUAD** program is doing very similar things, you can get pretty far by studying that program, and figuring out the corresponding changes to make in the MC program.

However, there is one important difference. Each process calls the random number generator, using a **SEED** value. If you use the same seed on each process, then you will get no benefit from using MPI. Be very careful to figure out a way to use a different seed on each process.

To verify that you have done this correctly, have each process compute and print out its own local estimate for the integral.

Also, think about using one name for the “local” variables and another name for “global” variables. For instance, does the variable **SAMPLE\_NUM** count the number of samples used on one process, or the total over all processes? You should think of a way to name things so that process 0 can do its part of the work, and also collect all the work into a final result, without getting confused.

When you have your program working, compute the problem with **DIM\_NUM** = 4, and run it on 4 processes, and use a total of 1,000,000 samples, that is, 250,000 per process. Fill in the following table from your program:

Process 0 estimate	-----	using	250,000 points
Process 1 estimate	-----	using	250,000 points
Process 2 estimate	-----	using	250,000 points
Process 3 estimate	-----	using	250,000 points
Total estimate	-----	using all	1,000,000 points.

The estimates should all be different!

### 4 PRIME: Sum the prime numbers from 2 to N

In this exercise, you will be given a working sequential program, that is very close to being an MPI program. You’ll try to convert it to using MPI.

The task to be carried out is to identify every prime number between 2 and 100,000, and to compute the sum.

The correct answer is 454,396,537.

Get a copy of the **prime** program and the shell script that submits it. One location is:

[http://www.scs.fsu.edu/~burkardt/vt2/fsu\\_mpi\\_2008/prime/prime.html](http://www.scs.fsu.edu/~burkardt/vt2/fsu_mpi_2008/prime/prime.html)

## 4.1 Run PRIME sequentially

Compile and run the program sequentially. Notice that the computation is carried out in 10 parts, as though there were 10 processes involved. The code breaks up the interval [2,100000] into equal parts, and uses a loop to examine each subinterval separately. At the end, the results are summed up to get the final answer.

This code computes the CPU time required to perform the calculation over each subinterval. Notice that the times are not roughly equal. This suggests that the later processes have more work to do.

## 4.2 Make an MPI version of PRIME

Modify the program so that it runs under MPI.

You can look at the QUAD program to get an idea of what you need to do.

The simple changes:

- invoke the MPI include file
- call **MPI\_Initialize**, **MPI\_Comm\_size** and **MPI\_Comm\_rank**
- call **MPI\_Finalize** at the end

The sequential code used a loop to simulate parallel execution. Remove the loop. The loop index **ID** will be the value we already got from **MPI\_Comm\_rank**.

The code already has the correct formulas to compute the subinterval for process **P**, and the code in the loop is all correct.

When process **P** is done, it needs to send its partial result to process 0 for summation. Use the routine **MPI\_Reduce** to carry this out.

Usually, we only have the master process compute the wall clock time. For this problem, however, let's compute and print the wall clock time used by each worker as well. This will help us to see the variation in the amount of work done. We can even total all these wall clock times, and have the master process print that number as well.

The wall clock values being moved in this case are double precision real numbers. So in the reduction call, the datatype to use here is either **MPI\_DOUBLE** for C and C++, or **MPI\_DOUBLE\_PRECISION** for Fortran.

The only statement in the code that needs to be carried out by just one process is the one that prints out the final sum. This should look something like

```
if ( id == 0 )
    print n_lo, n_hi, total, wtime
```

## 4.3 Revise PRIME so you can Practice Message Passing

Let's try a little bit of message passing with this example.

Instead of calling **MPI\_Reduce**, let's try to handle the communication ourselves.

We begin with each process having a value **ID\_TOTAL**. Process 0 must receive a copy of this value from each process, and add it to a variable called **TOTAL** which is initialized to zero.

There are a couple ways to do this, and a couple tricky steps. However, in outline, your new code could look something like this:

```

if ( ID == 0 )
  for SOURCE = 1 to P - 1
    TAG = 1
    MPI_Recv ( ID_TOTAL, 1, MPI_INT, SOURCE, TAG, MPI_COMM_WORLD )
else
  TARGET = 0
  TAG = 1
  MPI_Send ( ID_TOTAL, 1, MPI_INT, TARGET, TAG, MPI_COMM_WORLD )

```

(Fortran users should replace `MPI_INT` by `MPI_INTEGER`, and add an `IERR` argument on the end of the calls).

I have left out details. How does each subtotal get added to the total, for instance? Since the loop doesn't start at 0, how does process 0 include its own subtotal in the total? (These aren't really hard questions, just easy to overlook!)

Run your revised program. Does it get the right answer?

## 5 ASSIGNMENT: a Distributed Version of HEAT

**Your assignment:** Insert MPI calls into a version of the `jheat` program, so that the resulting coded runs using distributed memory.

You will demonstrate that the code runs in parallel by running it on the HPC system, first using 1 node, and then using 8 nodes. (This means you must change both the `NODES=` and `-NP` lines in the shell script.)

Save the output files of the two runs.

Turn in three files to Chris Harden by Friday, 26 September 2008:

1. your revised source code
2. the output from the HPC run on 1 processor
3. the output from the HPC run on 8 processors

Until Friday afternoon, you may ask me for advice, guidance, or help.

Get a copy of the `heat` program and the shell script that submits it. One location is:

[http://www.scs.fsu.edu/~burkardt/vt2/fsu\\_mpi\\_2008/heat/heat.html](http://www.scs.fsu.edu/~burkardt/vt2/fsu_mpi_2008/heat/heat.html)

### 5.1 The problem being solved by the HEAT program

The program is intended to produce a discrete estimate for the heat function  $h(x, t)$ , which is defined on the spatial interval  $0 \leq x \leq 1$  and the time interval  $0 \leq t \leq 10$ .

Our version of the heat equation has the form

$$\frac{\partial h}{\partial t} - \frac{\partial^2 h}{\partial x^2} = 0. \quad (1)$$

with initial condition

$$h(x, 0) = 95.0 \quad (2)$$

and boundary conditions

$$h(0, t) = 100 + 10 * \sin(t) \quad (3)$$

$$h(1, t) = 75 \quad (4)$$

For the sequential code, we discretize the problem so that in space we have  $N + 2 = 12$  nodes, and in the time direction 101 time values.

For the parallel version, *each process* will use  $N + 2 = 12$  spatial nodes. Each process will identify its values as  $H(0)$  through  $H(N + 1)$ . As we discuss in class, entries 0 and  $N + 1$  are “special”, that is, the process does not compute them, but gets them from neighboring processes, or using boundary conditions.

If there are  $P$  processes, then there will be a total of  $P * N + 2$  nodes, if we ignore overlap. The extra two nodes are the nodes at the left and right endpoints.

As we discussed in class, at each step, a process computes the new value of  $H(I)$  based on the current values of  $H(I - 1)$ ,  $H(I)$ , and  $H(I + 1)$ . To complete the calculation, the process must “borrow” updated copies of  $H(0)$  and  $H(N + 1)$  from its left and right neighbors.

## 5.2 Finish the MPI Conversion

The program is almost completely written for MPI. However, in the routine **heat\_part**, where the actual computation takes place, three lines are missing. These lines carry out the second step in the exchange of data, in which processors 1 through  $P - 1$  send their values of  $H(1)$  to the left, and processors 0 through  $P - 2$  receive these values into the variable  $H(N + 1)$ .

The missing lines are indicated by question marks. Figure out what the missing lines should be, and insert them.

## 5.3 Run your HEAT program

Compile your program on the FSU HPC system. Run it twice, using first 1 process, and then 8.

Since the number of  $X$  points we use actually increases as we ask for more processors, the solution data will not be comparable, and the timings won't mean what they usually mean. The problem size is also rather too small to get an MPI speedup. So don't be concerned about the relative times of the two runs.

Send a copy of your source code AND the 2 output files to Chris Harden.