Distributed Memory Programming With MPI
Computer Lab Exercises

Advanced Computational Science II
John Burkardt
Department of Scientific Computing
Florida State University

http://people.sc.fsu.edu/~jburkardt/classes/acs2_2012/mpi/mpi_exercises.pdf

23 October 2012

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

Although MPI is typically used on clusters, most of our lab exercises will be carried out directly on the lab machines, and in fact, each person will be using a single machine. In this case, the parallelism will come from regarding each core on the machine as a separate process with its own memory. Our last in-class exercise, however, will use the HPC cluster for comparison.

The lab exercises include:

1. *bashrc*: add MPI to your .bashrc file;
2. *hello*: compile and run a simple program;
3. *quad*: estimate an integral using quadrature;
4. *prime*: count the prime numbers from 2 to N;
5. *prime_hpc*: run prime on the FSU HPC;
6. *heat*: the 1D heat equation for distributed memory.
7. *search*: your assignment, search for solutions to an integer equation.

For most of the exercises, there is a source code program from which you can start. 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 next Tuesday**.

1 BASHRC: Add MPI to Your .bashrc File

In order to use MPI on the lab machines, we need to issue a command that defines which version of MPI we want to use and the names and locations of various useful files and programs. The command is the usual horrible impossible-to-remember string:

```
module load openmpi-x86_64
```

You have to issue this command every time you log onto a lab machine and want to use MPI.

A better solution is to copy this command into your **.bashrc** file, because the computer can do a much better job of remembering the command. That way, it will automatically set up MPI for you every time you log in.

To do this, be sure you are in your main directory. Use your favorite editor to update or create the **.bashrc** file, and insert the necessary line:

```
gedit .bashrc
  (insert the line "module load openmpi-x86_64")
  (then save and exit)
```

The **.bashrc** file only gets referenced when you log in, so the new definitions won't automatically take effect until next login. To force them to go into effect now, type

```
source .bashrc
```

The main thing this will do is define the MPI versions of the compilers, including:

```
mpicc
mpic++
mpif77
mpif90
```

To check that the compilers have been defined, try a command like

```
which mpicc
```

If the system knows how to find the MPI C compiler, it will print out its location. If not, it will print nothing. (If your **which** command comes back empty-handed, let me know!)

Assuming this has been done, we've at least made it easy to invoke MPI on the lab machines!

2 HELLO: Compile and run a simple program

Get a copy of the *hello* program. One location is:

```
http://people.sc.fsu.edu/~jburkardt/classes/acs2_2012/mpi/hello/hello.html
```

This program doesn't do any parallel processing, but it does call MPI functions, so it's a simple starting example. It shows:

- 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 or termination call.

Although the program is written in legal C or FORTRAN, you cannot compile it with the basic C or FORTRAN compiler, because the include file is not in the default location. In other words, a command like

```
gcc hello.c
```

will produce lots of error messages! Instead, we want to compile with the appropriate MPI compiler.

Once you've created an executable file called *a.out*, rename it to *hello*:

```
mv a.out hello
```

Where can we run this program? By great luck, our lab machines are dual processor quad core systems, which means there are eight cores available. An MPI program can run on eight cores on one computer just as well as on one core on each of eight computers. So let's use the lab machines!

How do we run the program in parallel? Remember, we want to run several copies, we want them to know how to communicate with each other, we want them to be assigned distinct MPI ID's. Simply typing **./hello** is not going to do this! Instead, we need to use a command that can take care of all these details. Since we are running all the copies on one machine, the only missing information we need to specify is *how many copies are to be run.* The command we need is called **mpirun** and it works like this:

```
mpirun -np 2 ./hello
```

The **-np 2** switch tells **mpirun** how many synchronized copies of the program are to be run. Since we didn't specify any information about *where* they are to run, they will run on the local machine. Notice that, with this command, you get a hello from two separate copies of the program, as you should expect!

3 QUAD: Using More Processes

Get a copy of the *quad* program. One location is:

```
http://people.sc.fsu.edu/~jburkardt/classes/acs2_2012/mpi/quad/quad.html
```

This program estimates an integral $\int_a^b f(x)\,dx$ using an average of function values at **n** equally spaced points. This is a natural application for parallel processing. Each process can evaluate the function at a subset of the points, and at the end the partial sums can be added together to get the integral estimate.

3.1 Run QUAD, varying the number of processes

Right now, we're not interested in the answer from the program, but in how long it takes to get that answer. So we set the value of **n** high enough that we can expect the program to at least roughly a second to run.

The **quad** program expects you to enter the number **n**. You can enter it on the command line:

```
quad 1000
```

or else it will ask for you input at run time:

```
quad
  Enter N, the number of intervals: 1000
```

What is the exact **mpirun** command that will run your program with 8 processes **and** set the value of **n** to 1000? (If you can't figure this out, please ask!)

```
mpirun  ____  ____  ____  ____
```

Try running the program with an increasing number of processes **p**, and a fixed number **n=10,000** of intervals. *Remember <u>not</u> to use a comma when you input the value of* **n** *to the program!*

```
 p      Time
--     ------
 1     ------
 2     ------
 4     ------
 8     ------
16     ------
```

Can you make any conclusions from these results?

3.2 Run QUAD, varying the amount of work

Fix the number of processes to 8, and vary the number of intervals. Make a series of runs with increasing values of **n**, the number of evaluation points. What is the behavior of the error? In particular, as we multiply **n** by 10, what happens to the error? Decide this by taking the ratio of the error **E** to the error **E1000** we see on the first calculation.

```
     n      E           E / E1000
  ----     ------      ----------
 1,000     ______      1.00_____
 2,000     ______      _________
 4,000     ______      _________
 8,000     ______      _________
16,000     ______      _________
```

We might expect that if we double **n**, the error will decrease by a factor of 1/2 ("everything is linear") or 1/4 ("sometimes, numerical analysis pays off"). Roughly what pattern do you see?

4 PRIME: Count the prime numbers from 2 to N

In this exercise, you are given a working sequential program that is very close to being an MPI program. You'll try to convert it to MPI. The program counts the primes between 2 and 100,000.

Get a copy of the *prime* program. One location is:

```
http://people.sc.fsu.edu/~jburkardt/classes/acs2_2012/mpi/prime/prime.html
```

4.1 Run PRIME sequentially

Compile and run *prime* sequentially, that is, use the regular gnu compilers to compile, and *do not* use **mpirun** to run. 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.

You can estimate the time that the program takes to run by using the **time** command. For instance:

```
gcc prime.c
mv a.out prime
time ./prime

(program output appears)

real  0m1.650s   <-- 1.65 seconds
user  0m1.648s
sys   0m0.001s
```

Record the time it took for the program to run:

```
Serial program ran in __________ seconds.
```

4.2 Make an MPI version of PRIME

Because the program has already been broken up into chunks, we can try to make a parallel version in which each chunk is done by a separate process.

Taking slow and careful steps, convert the program so that it uses MPI. What follows is one way to organize your work:
We always have to do these simple things:

- invoke the MPI include file;
- call **MPI_Initalize()**, **MPI_Comm_size()** and **MPI_Comm_rank()** before stuff happens;
- call **MPI_Finalize()** after stuff happens.

Clean up the print statements:

- Before the loop, there are some print statements. Only process 0 should print these;
- Inside the loop, there is a print statement. Let it stay as it is;
- After the loop, there are some print statements. Only process 0 should print these;

Remove the loop, and use MPI_Reduce to collect the results:

- Remove the beginning and ending of the loop which used to define **id**; the value of **id** is now set by the call to **MPI_Comm_rank()**;
- an **MPI_Reduce()** command should collect the partial sum to process 0;

Compute the time:

- Before the loop begins, process 0 prints some stuff. Include a call to **MPI_Wtime()** there, to start the timer;
- After the loop, process 0 prints some stuff. Call **MPI_Wtime()** again, to update the time, and then print it.

Once you think you've gotten the program converted to MPI, try to run it again with 4 and 8 processes. How does your time compare with the serial run?

```
serial       __________
4 processes  __________
8 processes  __________
```

Have we solved the PRIME problem? Here are some things to think about:

```
\item{If we increase the top number from 100,000 to 1,000,000, does the
problem get 10 times harder, and take 10 times as long?  (No, it takes much longer!);}
\item{Since each process has to check roughly the same number of potential primes,
does this mean they all have about the same amount of work? (No, the last process
works much harder than the first one!)}
```

5 PRIME_HPC: Run PRIME on the FSU HPC

Assuming there is time, and assuming the HPC is available, we can try to run our prime programs on the HPC cluster.

Copy the file "prime_batch.sh" from the same directory where you got the prime program.

To work on the cluster, you must have an account on the FSU HPC system. Moreover, you must know how to use **sftp** to transfer files up there, and **ssh** to set up an interactive login session. Because I prefer to edit files on my home system, I usually have both an **ssh** and **sftp** window open at the same time. If anything's wrong with a file on the HPC, I *get* it back to the home system, edit it on my local friendly editor, and then *put* the updated copy back.

Open sftp and ssh windows, and connect them both to the HPC system. My HPC account is on sc.hpc.fsu.edu. Copy your prime program, and the prime_batch.sh file to the HPC.

Once you're logged in, you want to compile your program. The FSU HPC system includes several choices for the compiler, and several "flavors" of MPI. For our work, we will choose the GNU compilers, and OpenMPI. To do this, we must issue the following command on the HPC, every time we plan to do some MPI work:

```
module load gnu-openmpi
```

(You could stick this command into the **.bashrc** file on the HPC, just like we did for the lab machines.)

Now compile your copy of the *prime* program using the appropriate command:

```
mpicc prime.c
mpiCC prime.cpp
mpif77 prime.f
mpif90 prime.f90
```

This will create an executable called *a.out*, which you should rename to *prime*:

```
mv a.out prime
```

The HPC login node is not supposed to be used for computation. It is shared by all the people who are trying to work on the cluster. While we could issue an **mpirun** command now, it would inconvenience all the other people who are logged in and doing editing and other tasks. Instead, we need to submit our job to the queueing system, which will sent it to a computational node instead.

Take a look at the script. There is probably nothing you have to change. The lines of text beginning with #MOAB are commands to the queueing system, specifying a time limit, number of processors, a job name and so on.

The commands that don't begin with a # are the commands that we can imagine entering interactively. In particular:

- **module load gnu-openmpi** sets up OpenMP with gnu compilers;
- **cd $PBS_O_WORKDIR** runs the job from "this" directory;
- **mpirun -np 8 ./prime > prime_output.txt** runs our program.

The number of processors requested by the **ppn=8** statement should match the number of MPI processes requested by the **mpirun -np 8** statement!

Note that the HPC cluster we are using in the classroom queue has 32 cores on each node, so you can run up to 32 MPI processes here.

Submit the script on the HPC using the command:

```
msub prime_batch.sh
```

To see the status of your job, you can issue one of the commands:

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

If your program has run successfully, the output will come back as a file with a name like *prime_batch.o12345*.

6 HEAT: The 1D Heat Equation

Insert MPI calls into a version of the *heat* program, so that the resulting coded runs using distributed memory.

Get a copy of the *heat* program. One location is:

```
http://people.sc.fsu.edu/~jburkardt/classes/acs2_2012/mpi/heat/heat.html
```

The program estimates the heat function $h(x,t)$, which is defined on the spatial interval $0 <= x <= 1$ and the time interval $0 <= t <= 10$.

Our version of the heat equation has the form

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

with initial condition

$$h(x,0) = 95.0 \tag{2}$$

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 discussed 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.

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 processes 1 through **p**-1 send their values of $h(1)$ to the left, and processes 0 through **p**-2 receive these values into the variable $h(n+1)$.

Run the program using 8 processes.

7 SEARCH: search for solutions to an integer equation.

We are given a function $f(i)$, defined only for positive integer inputs. We want to search the integers $a \leq i \leq b$ seeking a value j such that $f(j) = c$. We believe there is exactly one solution. This is a perfect opportunity for parallel programming.

Get a copy of the *search* program. One location is:

`http://people.sc.fsu.edu/~jburkardt/classes/acs2_2012/mpi/search/search.html`

You should be able to compile and run this program sequentially. It prints out an estimate of the running time for you as well. You might want to record this value.

Your assignment: Modify a copy of the *search* program so that it will run in parallel under MPI. It should also print out the wall clock time, as measured by **MPI_Wtime()**;

To get credit for this lab, turn in three files to Heng Dai by Tuesday, 30 October 2012:

1. your revised source code;
2. the output from a run using 1 process;
3. the output from a run using 8 processes.