**ACS2\_MPI\_2013   
Computer Lab Exercises**

**John Burkardt  
Department of Scientific Computing  
Florida State University**

Applied Computational Science II, Computer Lab Instructions for Tuesday, 13 September 2016, 3:30-6:00pm, room 152 Dirac Science Library.

This file is available at ["http://people.sc.fsu.edu/~jburkardt/classes/acs2\_mpi\_2013/acs2\_mpi\_2013.html"](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/acs2_mpi_2013.html)

**Index**

* [Introduction](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/acs2_mpi_2013.html" \l "introduction)
* [Setup](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/acs2_mpi_2013.html" \l "setup)
* [hello](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/acs2_mpi_2013.html" \l "hello)
* [quad](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/acs2_mpi_2013.html" \l "quad)
* [prime](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/acs2_mpi_2013.html" \l "prime)
* [heat](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/acs2_mpi_2013.html" \l "heat)
* [search](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/acs2_mpi_2013.html" \l "search)

**Introduction**

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

Distributed memory systems include a single processor with multiple cores, as well as multiple processors, which may be in physically separate boxes, as long as it is possible for the processors to communicate. MPI can be used on clusters with hundreds of processors. For this lab, we will be satisfied with the very simple environment offered by our dual-processor 4 core machines, which to MPI can look like 8 separate computing objects.

The MPI skills you learn can also be used on FSU's Research Computing Center (RCC) cluster, which includes thousands of nodes. To do so, you would need to request an account on the RCC system, and learn a little about how to use the non-interactive batch system to execute jobs. MPI is the most widely used system for parallel programming, and can be useful on your laptop, desktop, or any research cluster that uses C, C++, or FORTRAN.

The exercises involve the following programs:

* *hello* will introduce you to MPI;
* *quad* will estimate an integral;
* *prime* counts the prime numbers from 2 to n;
* *heat* models the 1D heat equation;
* *search* looks for solutions to a simple integer equation.

For each exercise, there is a source code program that you can use. The source code is available in a C, C++, FORTRAN77 and FORTRAN90 version, so you can stick with your favorite language. The PYTHON language also has a feature that allows it to work with MPI, but I have not provided examples in that language. You can copy each code, in a language that suits you, by using a browser.

You may want to refer to <http://people.sc.fsu.edu/~jburkardt/presentations/acs2_mpi_2013.pdf>, the lecture notes on MPI.

**Setup**

Using MPI means starting with a C or FORTRAN program, adding an include statement, calling some special functions, and compiling the program. But instead of calling the regular compiler, we will call a special version of the compiler that knows how to put together an MPI executable. For instance, if we have a C program, then the compiler will be called **mpicc** instead of **gcc**.

That might seem complicated enough, but on our lab system, you don't automatically have access to the **mpicc** compiler. So if you type a command like

mpicc myprog.c

you might get back the response

bash: mpicc: command not found

One way to check a command without actually trying to execute it is to use the **which** command: it will tell you whether a command exists, and if so, it will even tell you the exact place in the UNIX file system where the command is stored. If not found, it will tell you all the places it looks. Try this command:

which mpicc

Depending on your computer and operating system, you might not have any compilers installed, or you might now have the MPI libraries, or everything might be there, but you might now have the shortcuts defined that allow you to access them easily. The easiest way to check is to type a command like *which mpicc* which essentially says "Hey, computer, do you know what mpicc means?. A negative response will be a long message that starts like this:

which: no mpicc in (/usr/local/bin:/usr/bin:/bin...)

while a positive response will be short, giving the exact location in the file system of the command that will be executed if you type the shortcut "mpicc":

/usr/lib64/openmpi/bin/mpicc

On our departmental computer lab system in room DSL 152, in order to use MPI commands like **mpicc**, you need to first issue the following command:

module load mpi/openmpi-x86\_64

Issue this command, and then type:

which mpicc

The shortcuts defining the MPI commands are now set up for you. (Thanks to Amir Tahmassebi for helping update this information.)

But the next time you log in, you will have to issue that command again. Instead of trying to remember the command, you can simply include it in a file of commands that are automatically carried out every time you log in. Go to your login directory, and use an editor to edit the file whose name is **.bashrc** - yes, there is a period at the beginning of this file name. Insert the "module" command into this file.

gedit .bashrc

(insert the line "module load mpi/openmpi-x86\_64")

(then save and exit)

From now on, when you log in, the MPI compilers will automatically be set up for you. But of course, being UNIX, there's one small problem - the compilers aren't set up for you during this login session, because you changed the file *after* you logged in. If the **which mpicc** command doesn't report the location of the MPI C compiler, force the **.bashrc** file to be run:

source .bashrc

and try the **which mpicc** command one last time.

**hello**

For our first exercise, we're just going to try to run a program that uses MPI. Pick a version of the *hello* program:

* [hello.c](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/hello/hello.c)
* [hello.cpp](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/hello/hello.cpp)
* [hello.f](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/hello/hello.f)
* [hello.f90](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/hello/hello.f90)

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.

We need to compile the program with the appropriate MPI compiler. These compilers include:

* mpicc
* mpic++
* mpif77
* mpif90

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

mv a.out hello

It is possible to run a single MPI program on all the lab machines at once, for instance, but that requires a certain amount of setup. Let's just consider the easy case where we run the program on the eight cores on a single lab machine. setup that would be necessary.

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. When you execute this command, you get a hello from two separate copies of the program, as you should expect!

This result may seem similar to what happened with OpenMP, but think about the fact that, with MPI, you could be getting "hello" from every machine in this lab. It's obvious that each machine in the lab has a separate processor, separate memory, and can only communicate with you if there is some kind of communication channel set up. These are the characteristics of a **distributed**system. Because we happen to run the programs on the same machine, we don't realize what the full potential of MPI is.

**quad**

Pick a version of the *quad* program, which will estimate an integral by evaluating a function at equally spaced points. Our MPI version of this algorithm will give a portion of the interval to each process, and sum the partial results at the end.

* [quad.c](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/quad/quad.c)
* [quad.cpp](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/quad/quad.cpp)
* [quad.f](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/quad/quad.f)
* [quad.f90](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/quad/quad.f90)

The program is all set up to run under MPI. Compile it, and run it using 4 processes. The program is supposed to tell you the estimated answer, and how long it took to execute. Unfortunately, we get the four different answers, and 4 times. We need to fix this.

First, notice that the function **MPI\_Reduce()** is being used to collect the partial integral from each process, and add them all together on process 0. That means only process 0 actually has the correct integral at the end, although all the processes have a variable called **q** and print it out. To avoid confusion, we only want process 0 to print its value.

Similarly, rather than have every program tell us the time it took, it's probably good enough just to have process 0 tell us its time.

We can take care of both of these problems by putting an **if()** statement around the statements that print the result and the time. Each MPI program requested its MPI ID and stored the result as the variable **id**. Therefore, you want to modify the program to read something like

if ( id == 0 ) then

print integral estimate q

print elapsed wallclock time wtime

end if

**Make this change to your program and rerun it.**

**prime**

Pick a version of the *prime* program, which counts the number of primes between 2 and N:

* [prime.c](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/prime/prime.c)
* [prime.cpp](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/prime/prime.cpp)
* [prime.f](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/prime/prime.f)
* [prime.f90](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/prime/prime.f90)

The program can be run sequentially, but it has been written in a way that suggests how it could be run in parallel. The program needs to check the integers from 2 to N. To do this, it first breaks up the integers into P sublists. Let's suppose P is 4, for instance, and N is 15:

0: 2, 6, 10, 14

1: 3, 7, 11, 15

2: 4, 8, 12

3: 5, 9, 13

Then it loops over the sublists, checking all numbers on list 0 first.

Who knows why someone would write a sequential program this way? But this format makes it very easy to see how the problem can be parallelized. If we have P processes, then we simply give one sublist to each process. At the end of execution, however, each process must take its subtotal and transmit it to process 0, which can compute the grand total and print it.

**Make the following simple changes:**

* invoke the MPI include file;
* call **MPI\_Initalize()**, **MPI\_Comm\_size()** and **MPI\_Comm\_rank()** before the parallel section.
* call **MPI\_Finalize()** after the parallel section.

The program has some print statements, but we only want process 0 to do output. **Change the program so only process 0 prints.**

The outer loop examines all the sublists. But now, we want each program to examing a single sublist. **Remove the outer loop on id.**

Once the inner loop on *i* is completed, the program has counted all its primes. But this is only a partial result, and we need to collect all the partial results on process 0, so it can print the answer.**Call MPI\_Reduce() to collect the results.**

If n = 100,000, the number of primes should be 9,592. **Compile and run your program on just 1 process, and check the answer.**

Assuming the program is right, we want to know if it is efficient. We do this by calling **MPI\_Wtime()** before and after the loop. But we only need process 0 to report this information. **Modify your program to include timing output, and run it for n = 100,000.**

Assuming your program is working OK, we want to increase the work level. Set **n** to 1,000,000. **Run the program with 1, 2 and 4 MPI processes.**

**heat**

Pick a version of the *heat* program:

* [heat.c](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/heat/heat.c)
* [heat.cpp](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/heat/heat.cpp)
* [heat.f](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/heat/heat.f)
* [heat.f90](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/heat/heat.f90)

Over a 1D interval, we have placed equally spaced nodes. The first and last nodes have boundary conditions, and the interior nodes have initial conditions. The temperature at each interior node changes over time according to the heat equation. A discretized version of that equation tells us that the new temperature at node i depends on the old temperatures at nodes **i-1**, **i** and **i+1**:

^ h(i,j+1)

T ^

I |

M |

E h(i-1,j)-------h(i,j)-------h(i,j+1)

|

+--------------SPACE--------------------->

The program estimates the heat function *h(x,t)*, defined on the spatial interval [0,1] and time interval [0,10].

Suppose we have **n\*p+2** nodes in the spatial direction, so that **n\*p** nodes are interior, and there is one node at the left and right boundaries. Suppose that we have **p** processes, and that we assign each process **n+2** nodes in such a way that there is overlap between successive processes. In the diagram below, **p** = 4, **n** = 5:

P0 0--1--2--3--4--5--6

P1 0--1--2--3--4--5--6

P2 0--1--2--3--4--5--6

P3 0--1--2--3--4--5--6

For each process, nodes 1 through 5 are interior nodes. Nodes 0 and 6 are either boundary nodes, or they really "belong" to a neighboring process.

Then let us suppose that each process wants to update the temperature at its interior nodes. Because each process includes nodes 0 and 6, it has enough information to update nodes 1 through 5. That means, one way or another, all nodes get updated.

However, when process **id** updates the values of nodes 1 and 5, these pieces of information need to be sent to the neighboring processes before they take the next step. Moreover, the neighbors**id-1** and **id+1** need to send to process ID updated information for nodes 0 and 6.

So after each update, we need to make two calls each to the MPI functions **MPI\_Send()** and **MPI\_Recv()**. The heat program is almost complete, but the second pair of calls to MPI\_Send() and MPI\_Recv has been left with question marks representing the arguments. **Edit your copy of by putting in the correct argument information in these two calls. Then run the program with 4 MPI processes.**

**search**

Pick a version of the *search* program:

* [search.c](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/search/search.c)
* [search.cpp](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/search/search.cpp)
* [search.f](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/search/search.f)
* [search.f90](http://people.sc.fsu.edu/~jburkardt/classes/acs2_mpi_2013/search/search.f90)

We are given a function *f(i)*, defined only for positive integer inputs. Given the value **c**, we want to search the integers **a ≤ i ≤ 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.

You should be able to compile and run this program sequentially. The sequential program prints out a timing estimate, but this uses a CPU time function. Your MPI version of the program must call **MPI\_Wtime()** instead.

**Your assignment:** Modify a copy of the *search* program so that instead of being given a set of bounds (a and b for example), it is instead given an array that enumerates all the integers in within those bounds (or generates a list within those bounds). Your code should then be configured to send sections of that list to different processors to run in parallel under MPI, and print out the solution **j**. Also, have process 0 (only) print out the wall clock time, as measured by **MPI\_Wtime()**. **Broadcasting the entire list to every processor will result in half credit for the code correctness portion of this lab.**

By 11:59pm, September 20th, submit the following to Ryan Learn:

* a copy of the revised search program;
* a table of the execution times for 1, 2, 4, and 8 processors .