Parallel Programming Assignment 2
Compiling and running MPI programs

Author: B. Wilkinson and Clayton S. Ferner Modification date: Jan 30, 2014 (minor clarification Feb 17, 2014)

In this assignment, you will write and execute run MPI programs. ***For Spring 2014, we are going to try a new approach so watch for announcements on the home page for any corrections/clarifications.*** First you will test your programs on your own computer. This will require you to install an implementation of MPI on your own computer. Later you will test the programs on the UNC-C cci-grid0x.uncc.ed cluster. This approach reduces issues of faulty user programs running on the cluster affecting response time. Users can also do local editing, possibly with an IDE such as Eclipse-PTP (although for now, we do not explore Eclipse-PTP here, except as extra credit).

Part 1 is installing an implementation of MPI.

Part 2 has simple exercises to run an MPI hello world program on a single computer and on multiple computers to learn how to compile and execute programs

Part 3 asks you to write an MPI program to perform matric multiplication. A skeleton program is given to you (Appendix) that you have fill in details.

Part 4 asks you to test your MPI programs on the UNC-C cluster and compare execution times.

Part 5 asks you to create your own load balancing workpool program. This is similar to Assignment 1 (Seeds framework) but now you have use lower-level MPI routines.

When downloading files with a browser, be careful to make sure the file name extension and/or the contents have not been altered with HTML tags. Also be careful when copying and pasting code that unwanted characters are not copied (especially from Word documents).

Part 1 Installing MPI on your computer (20%)

You may do this any way you wish. There are two widely available implementations of MPI:

1. OpenMPI http://www.open-mpi.org/

2. MPICH http://www.mpich.org/

with Linux/Mac and Windows versions[1]. The Windows version in MPICH is a Microsoft product and a C compiler will be needed (comes with Visual Studio). A gcc-like C compiler for Windows can also be obtained from MinGW, http://www.mingw.org/

For Windows, an alternative approach is to provide a Linux environment of your computer and install a Linux version of MPI, for example with:

1. Cygwin http://www.cygwin.com/

[1] Binary support for Windows OpenMPI has been discontinued, see http://www.open-mpi.org/

2. VirtualBox https://www.virtualbox.org/

The Cygwin site already provides OpenMPI and Cygwin can be installed with OpenMPI libraries at the same time simply by selecting OpenMPI libraries during installation process. For VirtualBox, first install VirtualBox, then a distribution of Linux such as Ubuntu with say OpenMPI libraries. In both cases, installation can take a long time.

The following instructions are written for a Linux/Mac environment.

Check commands

Check the implementation and version with the commands:

```
which mpicc
which mpiexec
```

The full paths should be returned.

Set up a directory for the assignments

Make a directory called **mpi_assign** that will be used for the MPI programs in this course, and **cd** into that directory The Linux commands are:

```
mkdir mpi_assign
cd mpi_assign
```

All MPI commands will be issued from this directory.

Include in your submission document for Part 1:

Describe exactly what you did to install MPI with sufficient details for others to be able to do the same.

Part 2 Executing a simple Hello World program (20%)

Task 1: Hello World program

Create a C program called **hello.c** in the **mpi_assign** directory. This program is given below:

```
#include <stdio.h>
#include <string.h>
#include <stddef.h>
#include <stdlib.h>
#include "mpi.h"

main(int argc, char **argv ) {
    char message[256];
    int i,rank, size, tag=99;
    char machine_name[256];
    MPI_Status status;

    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    gethostname(machine_name, 255);

    if(rank == 0) {
        printf ("Hello world from master process %d running on %s\n",rank,
                                                    machine_name);
        for (i = 1; i < size; i++) {
            MPI_Recv(message, 256, MPI_CHAR, i, tag, MPI_COMM_WORLD, &status);
            printf("Message from process = %d : %s\n", i, message);
        }
    } else {
        sprintf(message, "Hello world from process %d running on %s",rank,
                    machine_name);
        MPI_Send(message, 256, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return(0);
}
```

The program sends the message "Hello, world from process # running on hostname XXXX" from each slave process (workers, rank != 0) to the master process (rank = 0). The master process receives the messages and then prints the messages to stdout.

Compile and execute the hello world program using four processes in total.

Compilation:

To compile the program use the command:

```
mpicc –o hello hello.c
```

which uses the **gcc** compiler to links in the libraries and create an executable **hello**, and hence all the usual flags that can be used with **gcc** can be used with **mpicc**.

Execution:

To execute an MPI program, the usual command is **mpiexec** with the specific path to the executable, i.e.:

```
mpiexec -n 4 ./hello
```

where **./** denotes the path to current directory (or give its full path) otherwise, you will get a "File not found" error.

So far, four instances of the program will execute just on one computer. You should get the output:

```
Hello world from master process 0 running on …
Message from process = 1 : Hello world from process 1 running on …
Message from process = 2 : Hello world from process 2 running on …
Message from process = 3 : Hello world from process 3 running on …
```

The order of messages may be different; it will depend upon how the four processes are scheduled on one computer. Comment on the how MPI processes map to processors/cores.

Task 2 Experiments with Code

Modify the hello world program by specifying the rank and tag of the receive operation to MPI_ANY_SOURCE and MPI_ANY_TAG, respectively. Recompile the program and execute. Is the output in order of process number? Why did the first version of hello world sort the output by process number but not the second?

Include in your submission document for Part 2:

Task 1

1. A screenshot or screenshots showing:
    a. Compilation of the hello world program
    b. Executing the program with its output

Task 2

1. Results of changing the rank and tag
2. Answers to the questions.

Part 3 Matrix Multiplication (20%)

Task 1: Creating MPI program

Obtain the input test file and output file from:

http://coitweb.uncc.edu/~abw/ITCS4145S14/Assignments/MatrixTestFiles/index.html

Copy the files to your **mpi_assign** directory. The input file contains two matrices of floating-point numbers that are 512x512 which you can use as input. The output file contains the answers of multiplying the two matrices in the input file.

Create a program using the skeleton program show in the Appendix. This skeleton program already has the code the read the input from the file and take time stamps using the system call **gettimeofday()**. You need to fill in the sections labeled "**TODO**". These sections include:

1) Scatter the input data to all the processors
2) Implement Matrix Multiplication in parallel
3) Gather the partial results back to the master process

Note: all of the code currently in the appendix (in black) should be done sequentially (one processor only). All of the code you add for the "TODO"'s should be done in parallel.

After you are able to compile successfully your matrix multiplication with **mpicc**, take a screenshot of the compilation for your submission document.
Also create and compile a sequential version of the program (i.e. without MPI).

Task 2: Run the parallel version

The program reads the input file. You will need to add the name of that file (**input2x512x512Doubles**) as the final command line argument after your program on the **mpiexec** command for example

```
mpiexec -n 4 ./hello input2x512x512Doubles
```

Run the command to execute the parallelized version of the matrix multiplication program using the number of processors in the range: 1, 4, 8, 12, 16, …. Then compare the output of your parallel program with the sequential version using the **diff** command [2]:

```
diff output.seq output.par.<P>
```

Sample output is shown in Figure 1:

```
1c1
<elapsed_time= 0.332111 (seconds)
---
>elapsed_time= 0.298292 (seconds)
```

*Figure 1: Example output from **diff** command*

If your program is implemented correctly, it should output the same answers as the sequential version of the program. If so, then the result of the **diff** command should be only 4 lines that look something like Figure 1. This means that the output is the same for both program but the execution time is different. If the **diff** command produces output that consists of many lines of numbers, then your parallelized matrix multiplication is not producing the same answers as the sequential version. You need to figure out why not and fix it. When your parallelized program can produce the same output as the sequential version, include snapshots of the output of the **diff** commands comparing all of the parallel output files with the sequential output file. Take a screenshot of the output of the diff commands comparing the parallel output with the sequential output for your submission document.

Task 3: Record and analyze results

Record the elapsed times for the parallelized program running on the different number of processors as well as the elapse time for the sequential version. Create a graph of these results using a graphing program, such as a spreadsheet. You have to create a graph of the execution times compared with sequential execution and the speedup curve with linear speedup. These graphs should look something like *Figure 2* and *Figure 3*, but the shape of the curves do not. Your curves may show something entirely different. The figures below are just examples. Make sure that you provide axes labels, a legend (of there is more than one line) and a title to the graphs. Include copies of the graphs in your submission document.

[2] On Windows, use the FC command.

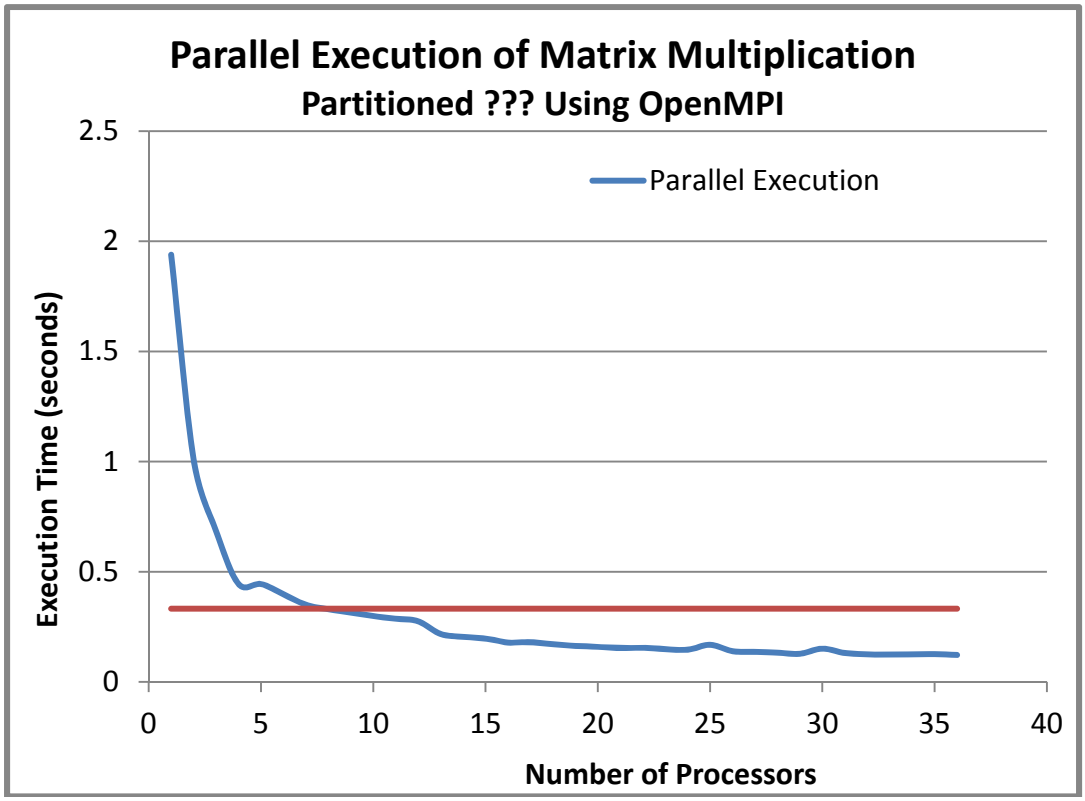

Figure 2: Example Execution Time Graph

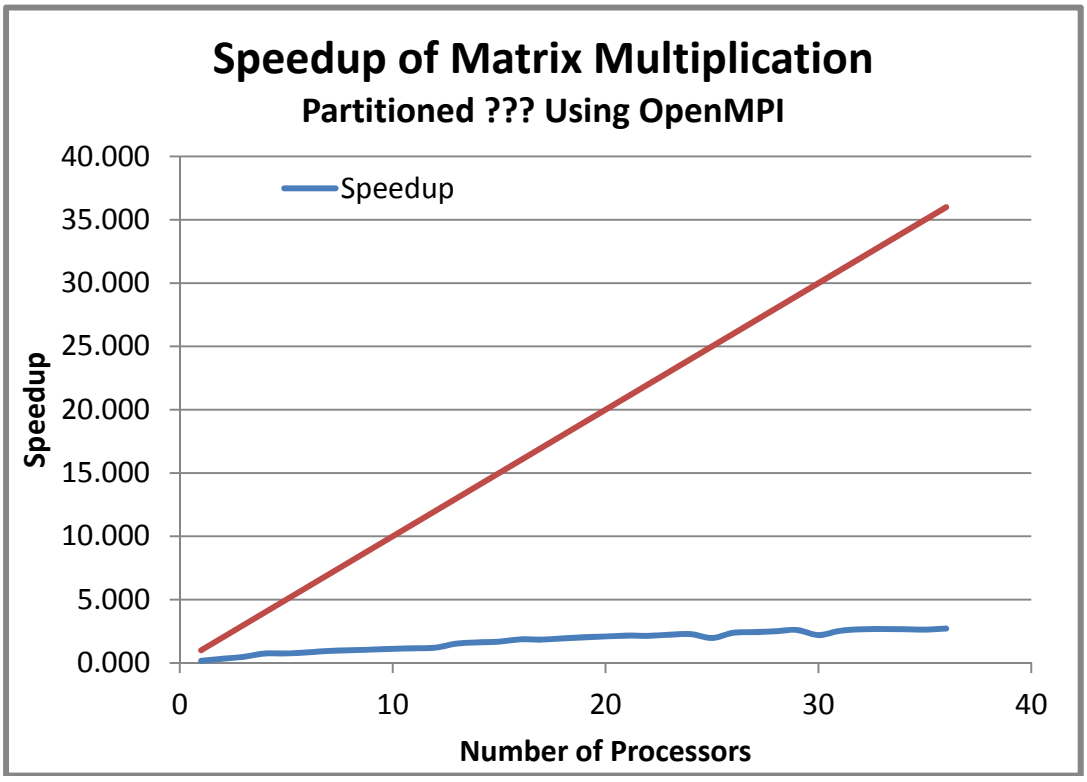

Figure 3: Example Speedup Graph

Include in your submission document for Part 3:

1. A copy of your matrix multiplication program
2. A copy of your execution time and speedup graphs
3. A screenshot or screenshots showing:
    a. Compilation of the program using **mpicc**
    b. Results of running the **diff** command comparing the parallel output with the sequential outputs

Part 4 Using the UNC-C cci-grid0x cluster (20%)

Now we will use the UNC-Cluster. ***Carefully*** review the notes on this cluster on the course home page:

http://coitweb.uncc.edu/~abw/ITCS4145S14/Assignments/UNCC_Cluster.pdf

Task 1 Connect onto the cci-grid0x cluster

Connect to **cci-gridgw.uncc.edu** using Putty (or another ssh client).

Check the MPI installed with the commands:

```
which mpicc
which mpiexec
```

The full paths should be returned.

Task 2 Executing MPI programs on a single computer

Make a directory in your home directory called **mpi_assign** that will be used for the MPI programs, and **cd** into that directory. Transfer all your MPI source programs **hello.c** and **matrix.c** from the previous parts to that directory.

Compile and execute the MPI programs:

- Hello world program, **hello.c** (from Part 2)
- Matrix multiplication program, **matrix.c** (from Part 3)

on **cci-gridgw.uncc.edu.** Establish the results are correct.

Task 3 Using multiple computers

The usual way MPI identifies the computers (machines) that it can use for executing an MPI program is by listing them in a file, and using the **-machinefile** flag with **mpiexec** (**mpiexec.hydra** on the cluster), i.e, if the file containing the list of machines is called **machines**:

```
mpiexec.hydra -machinefile machines -n 4 ./a.out
```

would run **a.out** on the cluster with four processes and each process would execute on one of the machines in the list. By default, MPI cycles through the list of machines giving processes to machines in a round robin fashion. (One can also specify the number of processes on a particular machine by adding that number after the machine name.)

Because of the way the cluster is set up, internal compute nodes communicate using local names. Create a file called machines containing the list of machines, *using the local names*, e.g.

cci-grid05
cci-grid07
cci-grid08
cci-gridgw.uncc.edu

To execute the **hello.c** program on the computers specified in the machines file with 8 processes, the command is:

```
mpiexec.hydra -machinefile machines -n 8 ./hello
```

Run the program with 8, 16, and 32 processors. Notice that the output from the processes is in order of process number.

Include in your submission document for Part 3:

1. A screenshot showing the execution of the **hello.c** program on **cci-gridgw.uncc.edu**. from your account
2. A screenshot showing the execution of the **matrix.c** program on **cci-gridgw.uncc.edu** from your account
3. A screenshot showing the execution of the **hello.c** program using multiple nodes of the cluster (**cci-grid05, cci-grid07, cci-grid08, and cci-gridgw.uncc.edu)**.
4. A screenshot showing the execution of the **matrix.c** program using multiple nodes of the cluster (**cci-grid05, cci-grid07, cci-grid08, and cci-gridgw.uncc.edu)**.
5. For matrix multiplication, in both cases show the results of running the **diff** command comparing the parallel output with the sequential outputs

Part 5 Implementing patterns - Workpool (20%)

The workpool pattern can provide powerful load balancing whereby when a processor returns one result, it is given further work to do. *Figure 4* shows such a workpool with a task queue. Individual tasks are given to the slaves. When a slave finishes and returns the result, it is given another task from the task queue, until the task queue is empty. At that point, the master waits until all outstanding results are returned (the termination condition – task queue empty and all result collected).

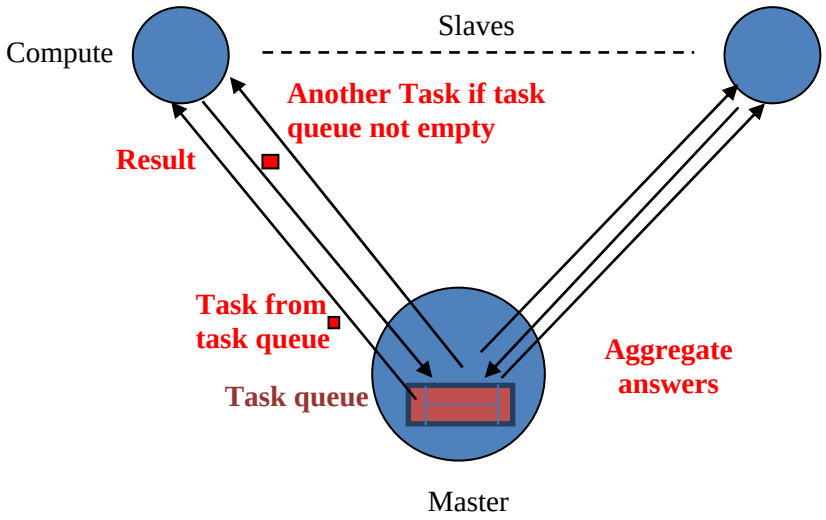

Figure 4: Workpool with a task queue

First write a sequential program in C to perform the Monte Carlo π calculation described in Assignment 1 (Seeds) using 10,000 samples. The program is to output both the value of π, its error from the exact value, and the execution time (done by instrumenting the code).

Then write an MPI program to implement a workpool version for the Monte Carlo π calculation, with 10,000 samples and *N* processes where *N* can be input.

Execute both programs on your own computer and on the cluster. On the cluster, run the MPI program with 4, 8, 12, and 16 processes. Compare the execution times with a sequential version and create graphs of the execution time and speedup as you did before.

Include in your submission document for Part 5:

1. Sequential C program for the Monte Carlo π calculation
   a. Copy of your source program
   b. Copy of the specified output both locally and on the cluster

2. MPI program for the Monte Carlo π calculation
   a. Copy of your source program
   b. Copy of the specified output both locally and on the cluster
   c. Speedup graphs with 4, 8, 12, and 16 processes

Extra Credit for everyone (up to 15%)

Demonstrate using Eclipse-PTP to compile and execute MPI programs on your computer or/and the cluster. *Give full details so that others might do the same.*

Assignment Submission

Produce a **single** pdf document that show that you successfully followed the instructions and performs all tasks. Max 30 pages.

Grading: Every task and subtask specified will be allocated a score so make sure you clearly identify each part/task you did (Part 1, Task 1 etc.). Make sure you include at least everything that is specified in the "Include in your submission document" section at the end of each part. **The items specified to include is not exhaustive. You may add additional materials to demonstrate you did the tasks. Include conclusions. Original code given in the assignment is not needed.**

Appendix – Matrix Multiplication Skeleton Program

```
#define N 512

#include <stdio.h>
#include <math.h>
#include <sys/time.h>

print_results(char *prompt, float a[N][N]);

int main(int argc, char *argv[])
{
    int i, j, k, blksz, error = 0;
    double a[N][N], b[N][N], c[N][N];
    char *usage = "Usage: %s file\n";
    FILE *fd;
    double elapsed_time, start_time, end_time;
    struct timeval tv1, tv2;

    if (argc< 2) {
        fprintf (stderr, usage, argv[0]);
        error = -1;
    }

    if ((fd = fopen (argv[1], "r")) == NULL) {
        fprintf (stderr, "%s: Cannot open file %s for reading.\n",
                                argv[0], argv[1]);
        fprintf (stderr, usage, argv[0]);
        error = -1;
    }
```

TODO: Broadcast the error. Have all processes terminate if the error is non-zero. (Be sure to use MPI_Finalize).

```
    // Read input from file for matrices a and b.
    // The I/O is not timed because this I/O needs
    // to be done regardless of whether this program
    // is run sequentially on one processor or in
    // parallel on many processors. Therefore, it is
    // irrelevant when considering speedup.

    for (i = 0; i< N; i++)
        for (j = 0; j < N; j++)
            fscanf (fd, "%lf", &a[i][j]);

    for (i = 0; i< N; i++)
        for (j = 0; j < N; j++)
            fscanf (fd, "%lf", &b[i][j]);
```

TODO: Add a barrier prior to the time stamp.

```
    // Take a time stamp
    gettimeofday(&tv1, NULL);
```

TODO: Scatter the input matrix a.

TODO: Broadcast the input matrix b.

TODO: Add code to implement matrix multiplication (C=AxB) in parallel. Each processors should compute a set of rows of the resulting matrix.

TODO: Gather partial result back to the master process.

```
    // Take a time stamp.  This won't happen until after the master
    // process has gathered all the input from the other processes.
    gettimeofday(&tv2, NULL);

    elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
                      ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
    printf ("elapsed_time=\t%lf (seconds)\n", elapsed_time);

    // print results
    print_results("C = ", c);
}

print_results(char *prompt, double a[N][N])
{
    int i, j;

    printf ("\n\n%s\n", prompt);

    for (i = 0; i< N; i++) {
        for (j = 0; j < N; j++) {
            printf(" %.2lf", a[i][j]);
        }
        printf ("\n");
    }
    printf ("\n\n");
}
```