Parallel Programming Assignment 3
Compiling and running MPI programs

Author: Clayton S. Ferner and B. Wilkinson Modification date: October 11a, 2013

This assignment uses the UNC-Wilmington cluster **babbage.cis.uncw.edu**. The assignment starts with simple exercises to run an MPI hello world program on a single computer and on multiple computers to learn how to submit programs to the UNC-W cluster. The next part asks you to write an MPI program to perform matric multiplication using multiple computers. A skeleton program is given to you (Appendix) that you have fill in details. This is similar to Assignment 2 (Paraguin compiler) but now you have use lower-level MPI routines. Next you are asked to create your own matrix multiplication program using a load balancing workpool approach. This is similar to Assignment 1 (Seeds framework) but now you have use lower-level MPI routines. Lastly, you are asked to implement a stencil pattern, which is similar to Assignment 2 (Paraguin compiler) but with MPI routines.

An implementation of MPI called OpenMPI is installed on the UNCW cluster. In OpenMPI, there are two commands (scripts) that will be used mainly: **mpicc** to compile MPI programs and **mpirun** to execute a MPI program. However, you do not use the **mpirun** command directly from the command prompt. Instead, as in Assignment 2, you will submit a job to a job scheduler called Sun Grid Engine (SGE), which will allocate the processors and run your program. You may use any editor available such as **vi** or **nano** when asked to alter files. 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 Preliminary tasks

Task 1: Connect to babbage.cis.uncw.edu

Connect to babbage.cis.uncw.edu using Putty (or another ssh client).

Task 2: Check commands

First check the implementation and version of these commands with:

```
which mpicc
which mpirun
which qsub
which qstat
```

The full paths should be returned.

Task 3: 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 commands are:

```
mkdir mpi_assign
cd mpi_assign
```

All MPI commands will be issued from this directory.
You do not need to include anything from this stage of the assignment in the submission document.

Part 2 Executing programs on babbage.cis.uncw.edu (25%)

Task 1: Executing a simple 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(intargc, 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();
}
```

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 hello.c –o hello
```

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 the program, you need to submit a job to the Sun Grid Engine (SGE) job scheduler. To do this, you will need to create a file call hello.sge with the following contents:

```
#!/bin/sh
#
# Usage: qsubhello.sge

#$ -S /bin/sh

#$ -pe orte4         # Specify how many processors we want
#$ -N Hello          # Name for the job
#$ -l h_rt=00:01:00  # Request 1 minute to execute
#$ -cwd              # Make sure .e and .o files arrive in working directory
#$ -j y              # Merge standard out and standard error to one file

mpirun -np $NSLOTS ./hello
```

The first line of this script indicates that this is a shell script. The line **`#$ -peorte4`** indicates to SGE that you want 4 processing elements (processors). The line **`#$ -N Hello`** indicates that the output files should be named "HelloXXX", where XXX contains other characters to indicate what it is and the job number. The line **`#$ -cwd`** tells SGE to use the current directory and the working directory (so that you output files will go in the current directory). The line **`#$ -j y`** will merge the stdout and stderr files into one file called "Hello.oXXX", where XXX is the job number. Normally, SGE would have created separate output files for the stdout and stderr, but this is not necessary here.

The line **`#$ -l h_rt=00:01:00`** tells SGE to kill the job after a minute. This helps to keep the system clean of old jobs. If you find that a job is still in the queue after it has finished, you can delete it using **`qdel #`**, where **`#`** is the job number. If you expect your job to take longer than a minute to run, you will need to increase this time.

The last line of the submission file **`mpirun -np $NSLOTS ./hello`** tells SGE to run the hello program using MPI run and the same number of processors as indicated in the **`#$ -peorte4`** line above.

To submit your job tothe SGE, you enter the command:

```
qsub hello.sge
```

You job is given a job number, which you will use for other SGE command. The output of your program will be send to a filed called Hello.oXX, where XX is the job number. There will also be a file called Hello.poXX, which you do not need. ***You will want to delete the output files you do not need. Otherwise, your directory will fill up.***

To see the list of jobs that have been submitted and have not yet been terminated, use the command **`qstat`**. If you find that a job is still in the queue after it has finished, you can delete it using **`qdel #`**, where # is the job number.

After your job has complete, you should get output in the file Hello.oXX similar to:

```
Hello world from master process 0 running on compute-0-0.local
Message from process = 1 : Hello world from process 1 running on compute-0-0.local
Message from process = 2 : Hello world from process 2 running on compute-0-0.local
Message from process = 3 : Hello world from process 3 running on compute-0-0.local
```

Task 2 Using multiple computers

a. In the previous exercise, only one computer was used even though we specified 4 processing elements. This is because there are multiple cores on each computer. Modify your SGE script to run the program on 8, 12, and 16 processors. Notice that the output from the processes is in order of process number.

b. 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 submit it to the scheduler. 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 copy of your hello world program
2. A copy of your job submission file
3. A copy of the output of your program
4. A screenshot or screenshots showing:
    a. Compilation of the hello world program
    b. Submitting the job to SGE
    c. **qstat** showing the job in the queue
    d. a directory listing (**ls** command) showing the output file exists

Task 2

1. A copy of the output of your program on the largest number of processors you were able to use (e.g. 32) before and after the modification in (a)
2. A copy of your hello world program after the modification in (b)
3. Answers to the questions in task 2 (b).

Part 3 Matrix Multiplication (25%)

Task 1: Creating the parallel version

Create a new directory under the assignment directory to do this task. First you will need some input. Copy the file **/home/cferner/working/work1/input2x512x512Doubles** to your directory. This file contains 2 matrices of floating-point numbers that are 512x512 which you can use as input. Also copy the file **/home/cferner/working/work1/output512x512mult** to your directory. This file contains the answers of multiplying the two matrices in the input file.

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

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

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.

Task 2: Run the parallel version

Using the **hello.sge** file as a template, make a job submission file for matrix multiplication (call it **matrix.sge**). Besides changing the names in the job submission file, you will also need to add **input2x512x512Doubles** as another command line argument after your program on the last line of the submission file.

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, …. Rename each output file from Matrix.o### to output.par.<P>, where <P> is the number of processors used. The compare the output of your parallel program with the sequential version using the command:

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

Use the sequential time given in the file **/home/cferner/working/work1/output512x512mult**.

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.

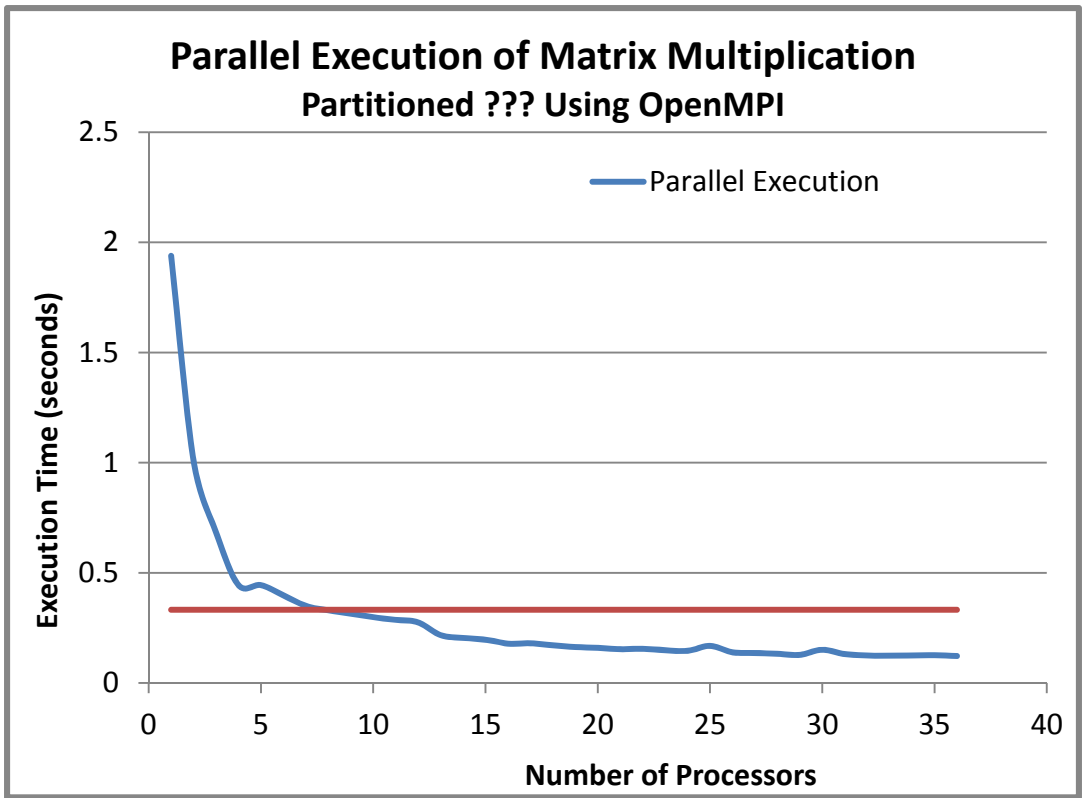

Figure 2: Example Execution Time Graph

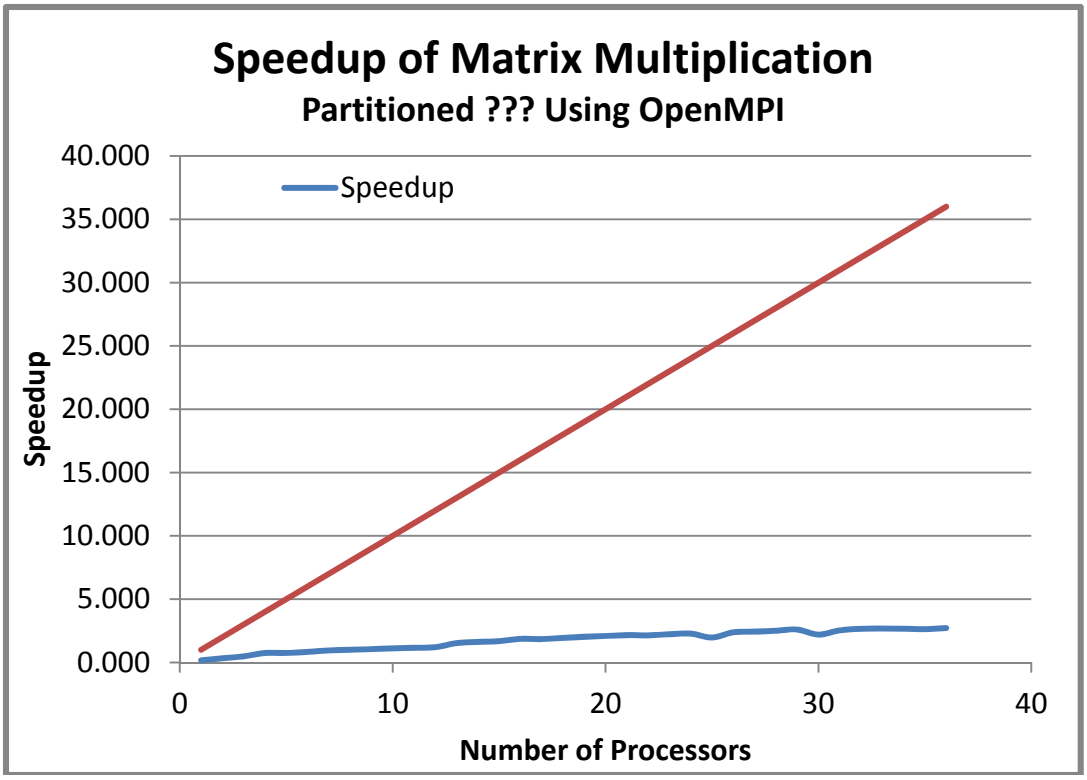

Figure 3: Example Speedup Graph

Include in your submission document for Part 3:

1. A copy of your matrix multiplication program
2. A copy of your job submission file
3. A copy of your execution time and speedup graphs
4. 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 Implementing patterns - a Workpool (25%)

In Part 3, the input arrays are divided statically across all the processors. A fixed group of rows of array **A** and a fixed group of columns of array **B** are sent to each processor. Then the master waits until all the processors return their results. However, a workpool 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).

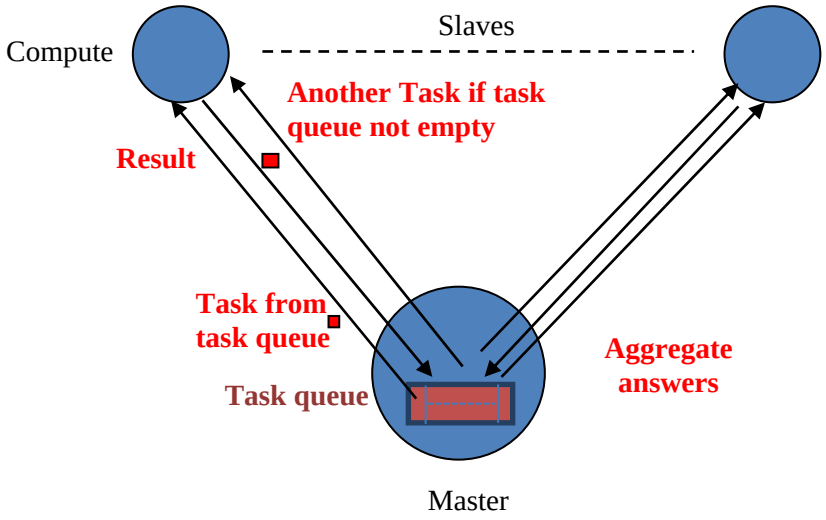

Figure 4: Workpool with a task queue

Implement a matrix multiplication workpool with a task queue. Each task processes one row and one column of the matrices and produces one result element as shown in Figure 5:

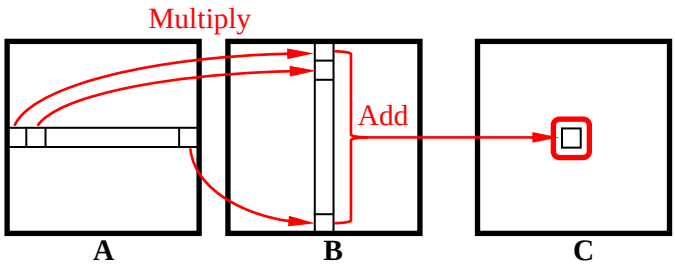

Figure 5 Matrix multiplication – each task

The master process is responsible for *all the code currently in the appendix* plus: 1) receiving requests from workers for work; 2) sending data to the workers to process; 3) receiving partial results and gathering them into a final answer; and 4) informing the workers when it is time to stop.

After you are successful in compiling and executing the program correctly on a single computer, run it on 4, 8, 12, and 16 processors. Compare the execution times with the sequential version using **diff** and create graphs of the execution time and speedup as you did before.

Include in your submission document for Part 4:

1. A copy of your workpool version of matrix multiplication program
2. A copy of your job submission file
3. A copy of your execution time and speedup graphs
4. A screenshot or screenshots showing:
    a. The outputs of the **diff** command

Part 5 Implementing patterns – a Stencil Pattern (25%)

Implement the Jacobi Iteration algorithm to simulate the heat distribution in a 10ft. x 10ft. room with a 4ft. wide fireplace centered along one wall. The heat of the fireplace is considered to be 100º C (although this is too cold for a fire), while the rest of the room is initially 0º C. Your program will perform the same computations as you did in Assignment 2; however, this time you need to implement the stencil pattern that was done for you by the Paraguin compiler.

The improved Jacobi Iteration algorithm shown in class is below:

```
int main(int argc, char *argv[])
{
    int i, j, current, next;
    double A[2][N][M];
    ...
    // A[0] is initialized with data somehow and duplicated into A[1]
    ...
    current = 0;
    next = (current + 1) % 2;
    for (time = 0; time < MAX_ITERATION; time++) {
        for (i = 1; i < N-1; i++)
            for (j = 1; j < M-1; j++)
                A[next][i][j] = (A[current][i-1][j] + A[current][i+1][j] +
                            A[current][i][j-1] + A[current][i][j+1]) * 0.25;
                current = next;
                next = (current + 1) % 2;
    }
    // Final result is in A[current]
    ...
}
```

You will need to turn this into a parallel algorithm using MPI. You will need to do:

1) Initialize the data
2) Broadcast the array to all processors
3) Before each computation of the new values
   a. Each processor (except the last one) will send its last row to the processor with rank one more than its own rank.
   b. Each processor (except the first one) will receive the last row from the processor with rank one less than its own rank.
   c. Each processor (except the first one) will send its first row to the processor with rank one less than its own rank.
   d. Each processor (except the last one) will receive the first row from the processor with rank one more than its own rank.
4) After the computation is complete, gather the partial results back on the master processor and print the results

After you are successful in compiling the program correctly, run it on 4, 8, 12, and 16 processors. Compare the results run on different number of processors.

Include in your submission document for Part 5:

1. A copy of your MPI solution of the stencil pattern program
2. A copy of your job submission file
3. A screenshot or screenshots showing your results.

Grading

Every task and subtask specified will be allocated a score so make sure you clearly identify each part/task you did. Make sure you include everything that is specified in the "Include in your submission document" section at the end of each part.

Assignment Submission

Produce a single pdf document that show that you successfully followed the instructions and performs all tasks by taking screenshots and include these screenshots in the document. Submit by the due date as described on the course home page. **Include all code, not as screen shots of the listings but complete properly documented code listing.**

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])
{
    inti, 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");
}
```