ITCS 4145/5145 Assignment 2

Compiling and running MPI programs

Author: B. Wilkinson and Clayton S. Ferner. Modification date: September 10, 2012

In this assignment, the workpool computations done in Assignment 1 will be re-written in C using MPI message passing libraries and executed on the UNCW cluster. You will be able to compare the two approaches. The programming environment is Linux using a command line interface. 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 won't use the **mpirun** command directly from the command prompt. Instead, 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.

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 MPI 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.

Exercise 1 "Hello World" (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(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();
}
```

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.

Compilation:

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

#$ -S /bin/sh

#$ -pe orte 16          # Specify how many processors we want

# -- our name ---
#$ -N Hello             # Name for the job
#$ -l h_rt=00:01:00     # Request 1 minute to execute
#$ -cwd                  # Make sure that the .e and .o file arrive in
the working directory
#$ -j y                  # Merge the 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 `#$ -pe orte 4` 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 create separate output files for the stdout and stderr, but this isn't necessary.

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 tells SGE to run the hello program using MPI run and the same number of processors as indicated in the `#$ -pe orte 4` line above.

To submit your job to the 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 don't need. ***You will want to delete the output files you don't 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
```

Include in your submission document:

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

Exercise 2 Using multiple computers (5%)

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

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:

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

Exercise 3 Matrix Multiplication (55%)

Task 1: Creating the sequential 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/input3** to your directory. This file contains a large number of floating-point numbers that you can use as input.

Using the skeleton program in the Appendix, implement the code to compute matrix multiplication on matrices A and B. This skeleton program already has the code the read the input from the file and take time stamps using the system call **gettimeofday()**. Do not implement (yet) the other parts to run the algorithm in parallel. Compile the program using **gcc** like this:

```
gcc matrix.c -o matrix
```

Include a screenshot of this compilation in your submission document as well.

Run the sequential matrix multiplication and take a screenshot for your submission document using a command like this:

```
./matrix input3 > output.seq
```

This command will run the matrix multiplication using the data in the file **input3** and re-direct the output to the file **output.seq**.

Run the command and take a screenshot for you submission document:

```
wc output.seq
```

This command will produce the number of lines, words, and characters of the file. Since the file is so large, we do not want to see the output, especially since they are just numbers and don't really mean anything. Using the **wc** command will tell me that the output file is of the correct size. The number of lines should be something like 518, if the input size is 512.

Task 1: Creating the parallel version

Create another version of the program to implement matrix multiplication in parallel. 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 should be done sequentially (one processor only). Also, the second matrix needs to be scatter along the columns and not the rows. You need to think carefully about how to do this.

After you are able to compile successfully your matrix multiplication with **mpicc**, take a screenshot of the compilation for your submission document.

Task 3: 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 **input3** as another command line argument after your program on the last line of the submission file.

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

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

Run the command to execute the parallelized version of the matrix multiplication program using the number of processors in the range: 1, 4, 8, 16, 32. 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>
```

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 4: Record and analyze results

Record the elapse 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.

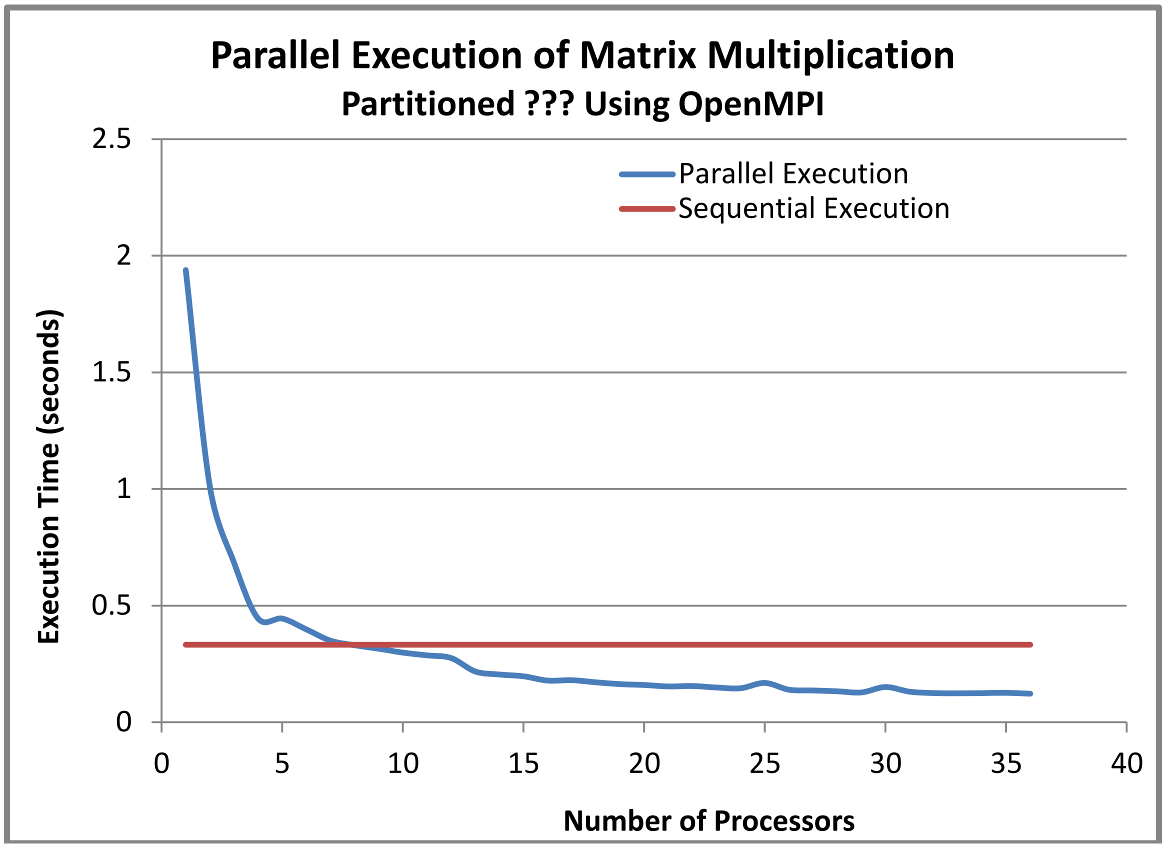

Figure 2: Example Execution Time Graph

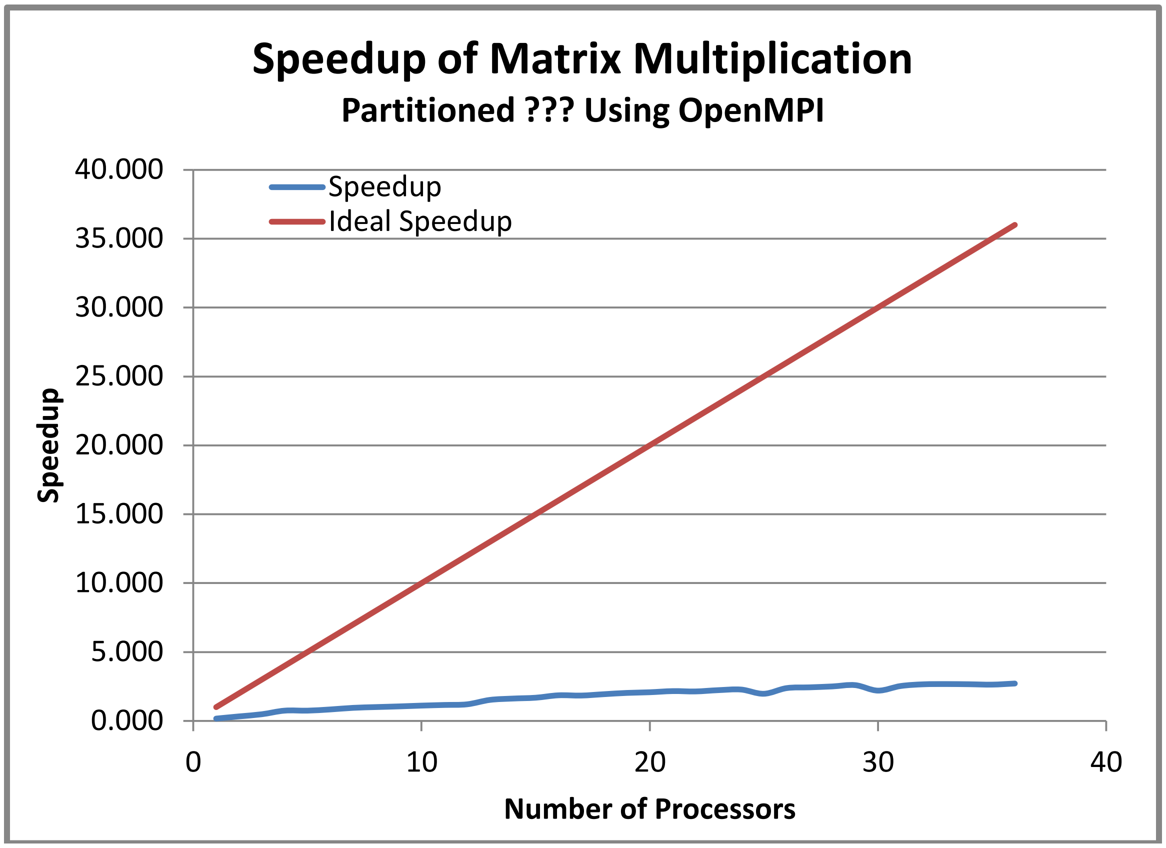

Figure 3: Example Speedup Graph

Include in your submission document:

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. Compilation of the program using **gcc**
    c. Results of running the **diff** command comparing the parallel output with the sequential outputs

Exercise 4 Workpool Version of Matrix Multiplication (15%)

Write a program to compute matrix multiplication using a workpool pattern similar to the way the Seeds framework implemented matrix multiplication in Assignment 1. In this implementation, instead of partitioning the work among the worker processes, each worker process will 1) ask the master for some work; 2) receive the data from the master; 3) compute the partial results; 4) return the partial results to the master; 4) repeat until the master says to stop. 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 compiler and executing the program correctly, run it on 4, 8, 16, and 32 processors. Compare the execution times with the sequential version using **diff** and create graphs of the execution time and speedup as you did before.

Compare and contrast using MPI and using the pattern programming approach.

Include in your submission document:

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
5. A comparison (written by you in your own words) comparing using MPI with using the pattern programming approach with the Seeds framework.

Grading

Every task and subtask specified will be allocated a score so make sure you clearly identify each part you did.

Assignment Submission

Produce a document that show that you successfully followed the instructions and performs all tasks by taking screenshots and include these screenshots in the document. (To include screenshots from Windows XP, select window, press Alt-Printscreen, and paste to file.) Provide insightful conclusions. 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;
    float 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]);
        return -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]);
        return -1;
    }


    // 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, "%f", &a[i][j]);

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

TODO: Add a barrier prior to the time stamp.

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

TODO: Scatter the input matrices a and b.

TODO: Add code to implement matrix multiplication (C=AxB) in parallel.

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, float 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(" %.2f", a[i][j]);
        }
        printf ("\n");
    }
    printf ("\n\n");
}
```