Assignment 4
The Suzaku Framework

Authors: R. Payne and B. Wilkinson
Modification date: April 4, 2014 (minor clarifications in red)

Overview

The purpose of this project is to implement a task queue workpool programming pattern using the Suzaku framework to solve block matrix multiplication. The Suzaku framework implements the workpool pattern for matrix multiplication using macros and routines and produces MPI code. Thus this is another way to produce pattern based programs without actually writing the message-passing code yourself. The actual Suzaku code is hidden in a provided object file. Later you will be asked to write this code yourself.

You should first try to use your own computer for this assignment. All that will be needed is an MPI compiler, which you used in a previous assignment. Several versions of the suzaku files are given on the course home page for different OS's/compilers. If none work for you, you may use the UNCC gridgw cluster to do this assignment.

Suzaku is a new project so watch for announcements on the home page for any corrections/clarifications.

Part 1 Preliminary tasks

Task 1: Setup

Download the Suzaku object file **suzaku.o** from the class website (Assignment 4).

Task 2: Check MPI installation

First check your MPI implementation with the commands with:

```
which mpicc
which mpiexec
```

The full paths showing the location of mpicc and mpiexec should be returned.

Task 3: Set up a directory for the assignments

Make a directory called Suzaku that will be used for the Suzaku programs in this course, and cd into that directory. The commands are:

```
mkdir Suzaku
```

```
cd Suzaku
```

All 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 Hello World source code (25%)

Task 1: Executing a simple Hello World program

Create a C program called hello.c in the Suzaku directory. This program is given here:

```
#include "suzaku.h"

void compute(double a[N][N], double b[N][N], double c[N][N], int index, int
blksize){

}

int main(int argc, char **argv){
      int p, rank;

      MPI_START(&p, &rank, &argc, &argv);

      for(int i = 0; i < 10; i++){
            printf("Hello world from process: %i \n", rank);
      }

      MPI_Finalize();
      return 0;
}
```

This program outputs "Hello world from process: " and the process number from which the line of code is executing.

Compilation:

Place the `suzaku.o` object file and `suzaku.h` header file from the home page with your source code. To compile the program use the command:

```
mpicc -std=c99 -c -o hello.o hello.c
mpicc -std=c99 hello.o suzaku.o -o helloworld
```

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.

The first command with the –c option will create an object file (.o) rather than the final executable. Make sure you include the –c. The second command will link the object file with the suzaku object file to create the final executable.

Because int i is declared inside the for loop, an additional flag of "-std=c99" may be needed to compile the hello world program if the underlying compiler is gcc, i.e. use:

```
mpicc -std=c99 ….
```

"-std= … " determines the language standard, in this case ISO C99.

Execution:

To execute the program, use the following commands in terminal:

```
mpiexec –n # ./helloworld
```

where the "#" is the number of processors that the program will execute using.
After your program is complete, you should have output similar to the following (if you specified 2 processors):

```
Hello world from process: 0
Hello world from process: 0
Hello world from process: 0
Hello world from process: 0
Hello world from process: 0
Hello world from process: 0
Hello world from process: 0
Hello world from process: 0
Hello world from process: 0
Hello world from process: 0
Hello world from process: 1
Hello world from process: 1
Hello world from process: 1
Hello world from process: 1
Hello world from process: 1
Hello world from process: 1
Hello world from process: 1
Hello world from process: 1
Hello world from process: 1
Hello world from process: 1
```

Include in your submission document for Part 2:

Task 1

1. A screenshot of the output using 1, 2, and 4 processors The source with the compute function and the sequential completed (not as a screenshot).

2. A screenshot of the directory containing the source files and the executable.

Part 3 Block Matrix Multiplication using Suzaku (25%)

Task 1: Creating the sequential matrix multiplication version

First you will need some input. Download the file `input2x512x512Doubles` to your directory. This file contains two matrices of floating point numbers that are 512x512 which you can use as input. Also copy the file `output512x512mult` to your directory. This 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()`. The Suzaku framework solves block matrix multiplication using a task queue to implement a workpool pattern. The master and worker process functions accept an argument for the block size which is defined at the top of the source code. You need to fill in the `compute` function and the case if there is only one processor. The `compute` function is going to have to handle the computation of rows/columns starting from the index and going block size number of iterations. Since the processor equal to one case is sequential, it only requires the straight-forward solution.

Before adding any code, compile the source with the following commands:

```
mpicc –c matrixmulti.c –o matrixmulti.o
mpicc matrixmulti.o suzaku.o –o matrixmulti
```

Then run the program with the following command:

```
mpiexec –n 1 ./matrixmulti input2x512x512Doubles >output
```

The output file will contain the resultant matrix. Right now, a diff command comparing the `output` file against the `output512x512mult` file would show that the output is not correct. The command to check the difference between the files is:

```
diff output512x512mult output
```

Note: You can output the difference to a text file by adding `>diffOutput`

Fill in the code for the processor equal to one case, then check your output against the `output512x512mult` file to verify that you have the correct resultant matrix.

Task 2: Creating the parallel block matrix multiplication version

Fill in the code for the `compute` function. The compute function will contain a for loop that iterates over the block size worth of columns and rows. The function also should record each result to the “c” matrix with the assignment operator. Since the matrices are

passed by reference, the values will be appropriately stored, thus no need for a return value.

Using the `diff` command, you will be able to check your computation against the sequential output. You should have output that resembles:

```
< elapsed_time= 1.218369 (seconds)
---
> elapsed_time= 6.643332 (seconds)
```

Task 3: Record and analyze results

Record the elapsed times for the parallelized program running on the different number of processors and block sizes as well as the elapse time for the sequential version. It is not necessary to test different block sizes on the p=1 case. 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.

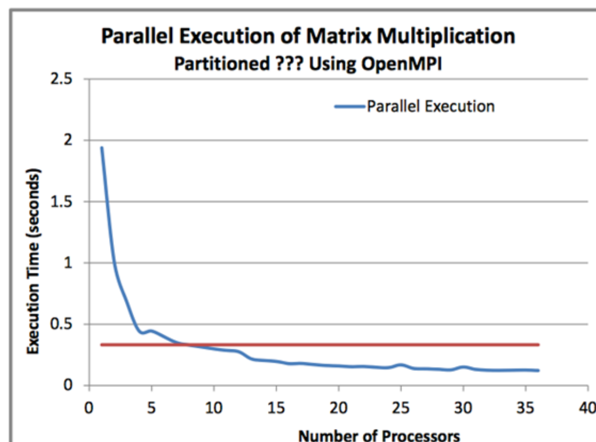

Figure 2: Example Execution Time Graph

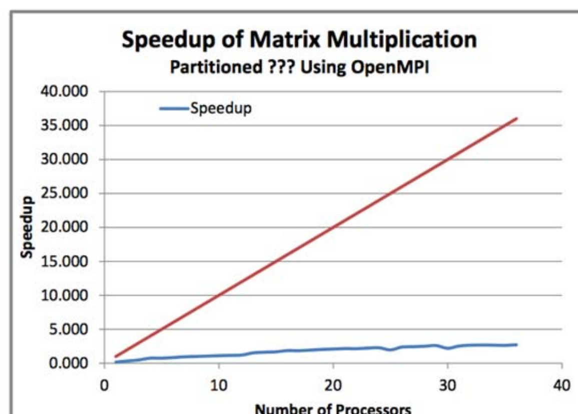

Figure 3: Example Speedup Graph

Include in your submission document for Part 3:

Task 1

1. A screenshot of compiling your source, linking the object file with the suzaku object, and executing to an output file.

Task 2

3. The source with the compute function and the sequential completed(not as a screenshot).

Task 3

1 Screenshots of the execution times and speed up graphs for p = 1, 2, 4, 8 and block sizes of 1, 2, 4, and 8.

4. The processor speed and number of cores of the machine that you are running the program from.

5. An explanation of the results. Why is the block size affecting the speed? How might your processor be impacting the results.

Part 4 Replacing the Master/Worker Functions with MPI (25%)

In Part 3, the Suzaku framework used a task queue workpool (*Figure 4*) to manage the work being sent between processors. The task queue simply sends a row(s) of A and the worker receives the row(s), computes the results, and sends the results to the master process to store in the C matrix. The master process starts from the beginning of the A matrix and sends a block size worth of rows at a time, until all rows have been sent out. Then the master process sends a termination signal to the workers.

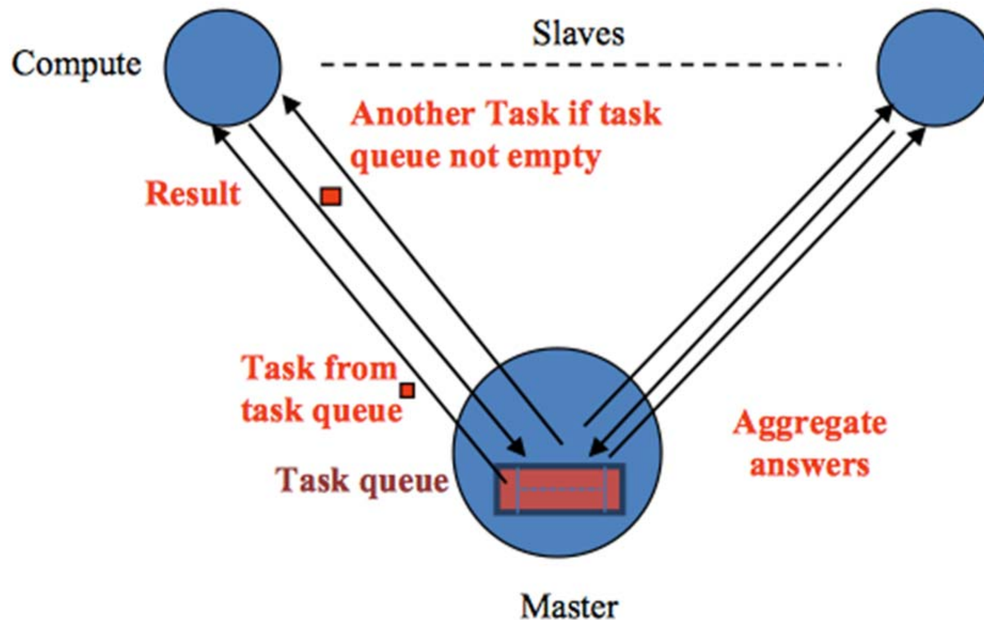

Figure 4: Workpool with a task queue

Task 1: Replacing the `workerProcess` Function

For this part, you will write functions using MPI send and receive calls with the code you wrote from part 3. Start out by writing the worker process first, where the termination signal is an integer N. You will not need to modify the main function, other than calling your new worker process. Remember, the worker has to receive work before it does any computation, so the worker will wait for the index of the work, then the associated row(s)(Hint: 2 separate receives). The worker calls the `compute` function, then sends the results back to the master process. After the worker process is complete, verify that it works by compiling and running your program.

Task 2: Replacing the `masterProcess` Function

Suzaku's master process uses a counter based on the amount of work that has yet to be accomplished. First, it gives out work to each processor then waits for a processor to

return with a result. After it receives the result, it employs the returning worker by sending it another piece of work to accomplish. The result is a task queue workpool where none of the processors are sitting around waiting for other processors.

Task 3: Record and analyze results

Just as in part 3 task 3, record and graph the speed up and execution times using p = 1, 2, 4, 8 and block sizes 1, 2, 4, 8.

Include in your submission document for Part 4:

Task 1 & 2

1. A screenshot of compiling your source, linking the object file with the suzaku object, and executing to an output file.
2. The source code containing your completed functions(not as screenshot).

Task 3

1. Screenshots of the execution times and speed up graphs for p = 1, 2, 4, 8 and block sizes of 1, 2, 4, and 8.

2. The speed and number of cores the processor that you record your results from. **Must be the same machine as the one used in Part 3.**

3. A comparison to the results you obtained in Part 3.

Part 5 Block Matrix Multiplication with B^T (25%)

In the previous parts, the **B** matrix was broadcasted to each process. This was a method to deal with the issue of needing to send out columns rather than rows. Another method of dealing with this same problem is to transpose the **B** matrix. Recall that matrix transposition swaps the elements in the *i*-th row and *j*-th column to the *j*-th row and the *i*-th column. The result is a matrix such that the rows and columns have been switched. Since memory is stored contigiously in an array, transposing the matrix enables us to send rows instead of broadcasting the entire array. The compute function needs to be adjusted such that the **C** matrix contains the solution for **A** x **B**, and not **A** x $\mathbf{B}^{\mathbf{T}}$(transposed).

Task 1: Transpose the B Matrix

Using your code from Part 3, remove the function call `mpiBroadcastArrayOfDoubles(*b)` from the source. You are allowed to use MPI commands and are able to remove other functions to accomplish this task, however, you must keep the following function calls in place:

```
compute(...)
MPI_START(...)
startTimer(...)
masterProcess(...)
workerProcess(...)
stopTimer(...)
printResults(...)
```

There are multiple ways to accomplish the transposition and completing the block matrix multiplication using the Suzaku framework. Link and compile your source just as before. Use the difference command to verify that the output matrix is correct.

Task 2: Record and Analyze the results

Just as in part 3 task 3, record and graph the speed up and execution times using p = 1, 2, 4, 8 and block sizes 1, 2, 4, 8.

Include in your submission document for Part 5:

Task 1

1. A screenshot of compiling your source, linking the object file with the suzaku object, executing to an output file, and comparing your output file using the diff command.
2. The source code containing your solution for transposing the B matrix (not as screenshot).

Task 2

1. Screenshots of the execution times and speed up graphs for p = 1, 2, 4, 8 and block sizes of 1, 2, 4, and 8.

2. The speed and number of cores the processor that you record your results from. **Must be the same machine as the one used in Part 3 & 4.**

3. A comparison to the results you obtained in Part 3 & 4.

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 A – Block Matrix Multiplication Skeleton Program

```
#include "suzaku.h"
#define BLKSIZE 1

void compute(double a[N][N], double b[N][N], double c[N][N], int index, int
blksize){
      //INSERT MATRIX MULTIPLICATION TO BE DONE BY WORKERS HERE
      //FOR P > 1
}

int main(int argc, char *argv[]) {
      int i, j, k, error = 0;
      int p, rank = 0;
      double a[N][N], b[N][N], c[N][N];
      double sum;

      MPI_START(&p, &rank, &argc, &argv);
      readInputFile(argc, argv, &error, a, b);
      startTimer(rank);

      if(p == 1){
            //INSERT CODE FOR MATRIX MULTIPLICATION ON 1 PROCESSOR
      }
      else{
            //Send out the b array to the workers
            mpiBroadcastArrayOfDoubles(*b);
            //Uses a task queue to issue work and rows from a as workers come
            //back with completed work
            masterProcess(a, c, p, rank, BLKSIZE);
            //Fetches work and returns the results calculated by the compute
            //function
            workerProcess(a, b, c, rank, BLKSIZE);
      }

      stopTimer(rank);
      printResults("C =", c, rank);

      return 0;
}
```

Appendix B – Suzaku Interface

```
//Broadcasts a 2 demisional array to the processes
void mpiBroadcastArrayOfDoubles(double *array);
//Manages the work flow of the workers
void masterProcess(double array1[N][N], double array2[N][N], int p, int rank, int blksize);
//Workers receive work and call the compute function provided to get results
void workerProcess(double array1[N][N], double array2[N][N], double array3[N][N], int rank, int blksize);
//Print the results of a matrix of double
void printResults(char *prompt, double array[N][N], int rank);
//Takes the input arguments from the command line and reads the filename to
//read the values into the arrays
void readInputFile(int argc, char *argv[], int *error, double array1[N][N], double array2[N][N]);
//Starts the measurement of execution time by calling timeofday
void startTimer(int rank);
//Stops the measurement of execution time and displays the calculated time
//also using timeofday and terminates MPI
void stopTimer(int rank);
//The function that needs to be defined in the workpool implementation,
//this function is called by the workerProcess function
void compute(double array1[N][N], double array2[N][N], double array3[N][N], int index, int blksize);
```