Sobel Edge Detector

Parallel Computer Architecture: Assignment 4

Stefen Lagos
Benjamin Wheeler

Notes

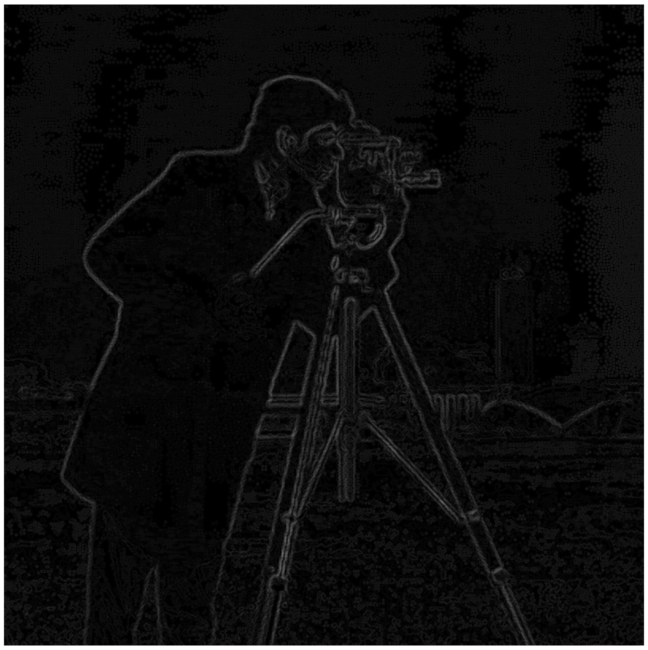
As per the README:

- Use GNU Make to build and run the project.
- The `make run` target will build and run the executable.
- The `make clean` target will remove build artifacts.

Part 1

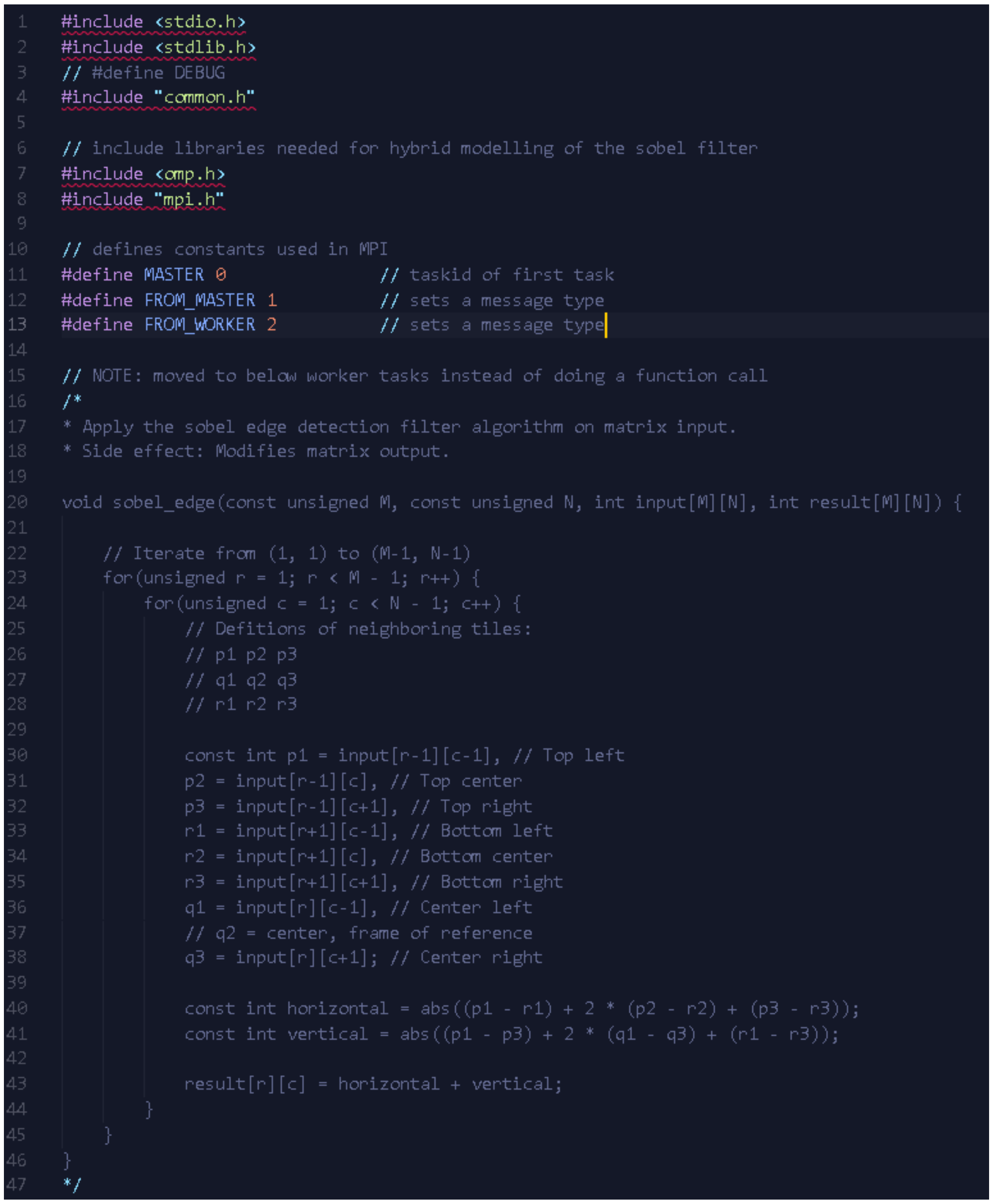
We implemented our own sobel edge detection algorithm based on the given formulae in the provided slides.

Output image:



Part 2

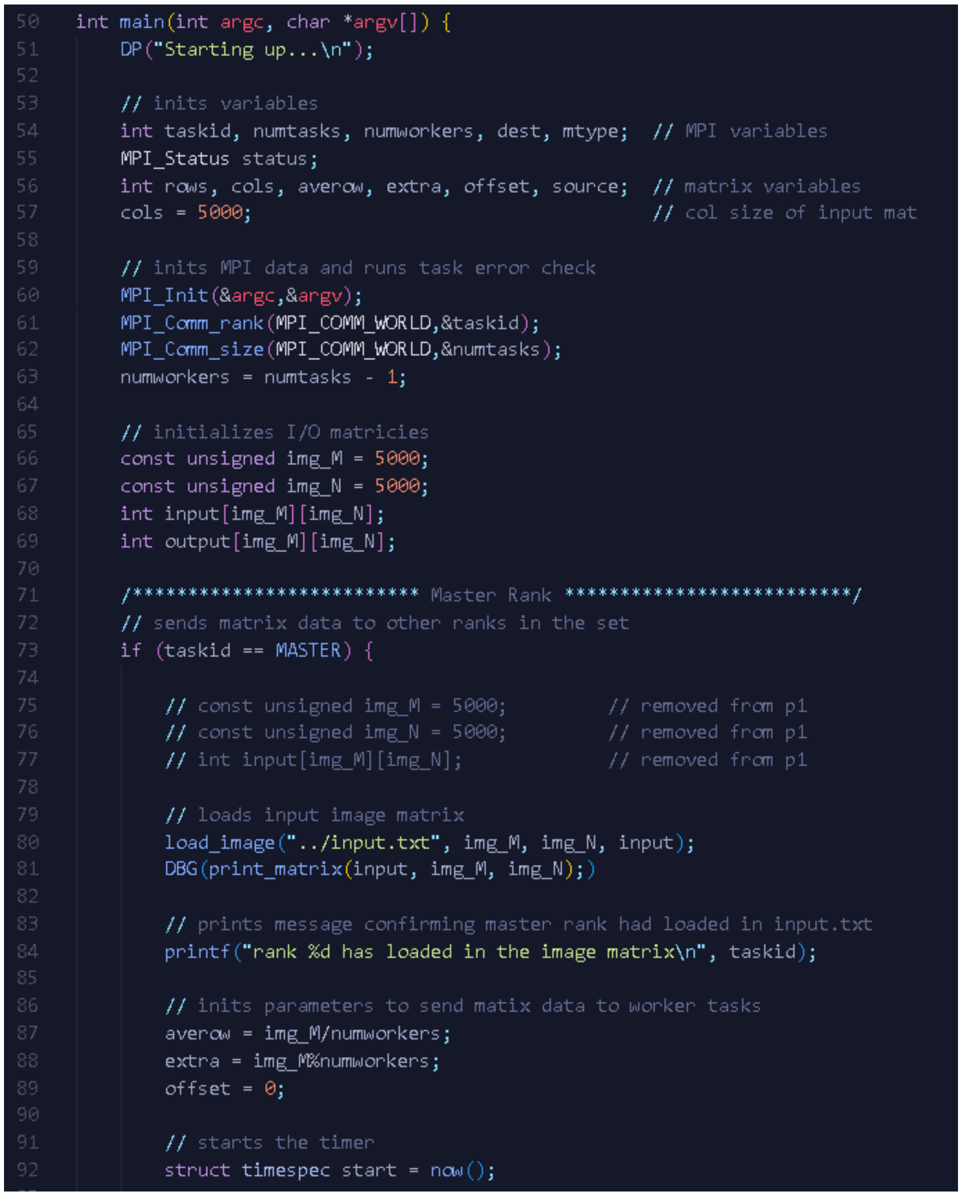
The modified code from Part 1 revolved mostly around the introduction of the MPI and OpenMP constructs. To begin, appropriate header libraries were included in the main C file to allow for the usage of both MPI and OpenMPI. Next, the majority of the “main.c” body was moved inside one master task, as to not have the ‘txt’ input file loaded into the system by every rank (i.e., doing this inside the master task region ensures the image is loaded only once) and debug print messages run by every task. These changes are shown below:



```c
#include <stdio.h>
#include <stdlib.h>
// #define DEBUG
#include "common.h"

// include libraries needed for hybrid modelling of the sobel filter
#include <omp.h>
#include "mpi.h"

// defines constants used in MPI
#define MASTER 0                // taskid of first task
#define FROM_MASTER 1           // sets a message type
#define FROM_WORKER 2           // sets a message type

// NOTE: moved to below worker tasks instead of doing a function call
/*
* Apply the sobel edge detection filter algorithm on matrix input.
* Side effect: Modifies matrix output.

void sobel_edge(const unsigned M, const unsigned N, int input[M][N], int result[M][N]) {

    // Iterate from (1, 1) to (M-1, N-1)
    for(unsigned r = 1; r < M - 1; r++) {
        for(unsigned c = 1; c < N - 1; c++) {
            // Defitions of neighboring tiles:
            // p1 p2 p3
            // q1 q2 q3
            // r1 r2 r3

            const int p1 = input[r-1][c-1], // Top left
            p2 = input[r-1][c], // Top center
            p3 = input[r-1][c+1], // Top right
            r1 = input[r+1][c-1], // Bottom left
            r2 = input[r+1][c], // Bottom center
            r3 = input[r+1][c+1], // Bottom right
            q1 = input[r][c-1], // Center left
            // q2 = center, frame of reference
            q3 = input[r][c+1]; // Center right

            const int horizontal = abs((p1 - r1) + 2 * (p2 - r2) + (p3 - r3));
            const int vertical = abs((p1 - p3) + 2 * (q1 - q3) + (r1 - r3));

            result[r][c] = horizontal + vertical;
        }
    }
}
*/
```

Additionally note that the sobel_edge function was commented out. This is because instead of a function call, in the main body of the C-file, an if-else chain was used to specify if the operating region was either correlating to a master or a worker. Worker task regions ran the body of this algorithm instead of calling it through a function.

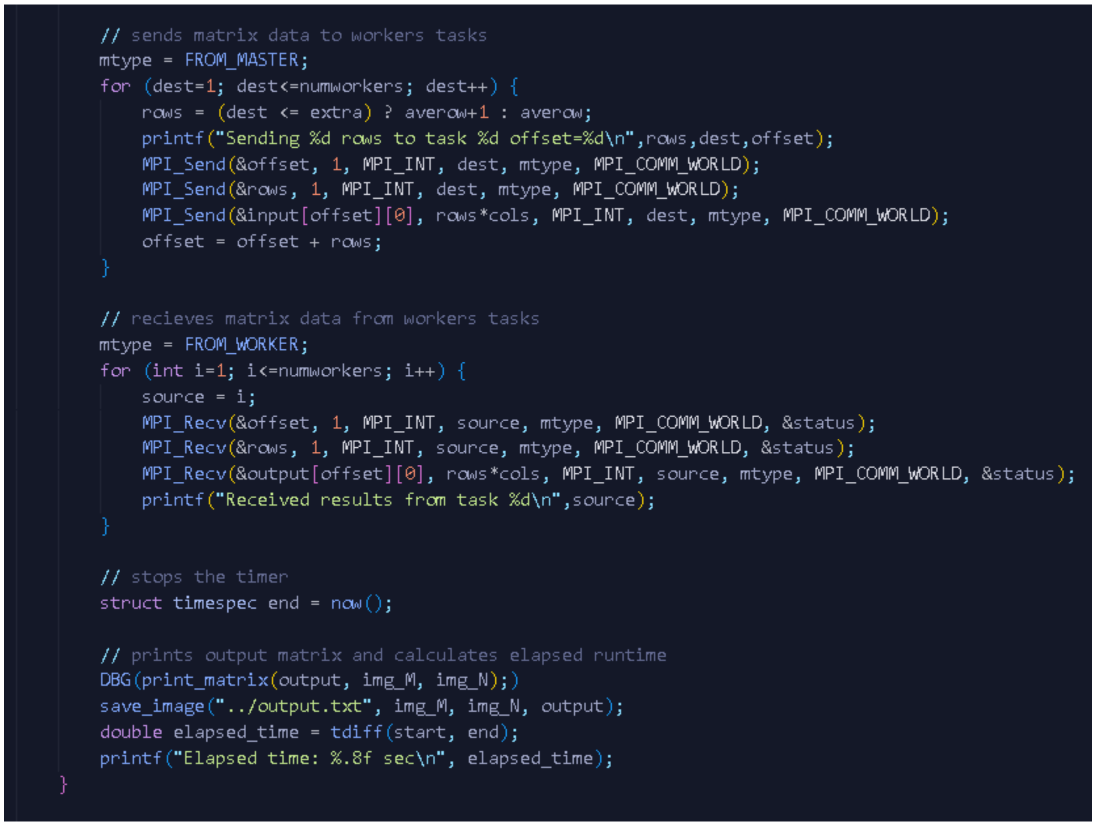
The below shows the beginning of the master task region, explained above:



```c
int main(int argc, char *argv[]) {
    DP("Starting up...\n");

    // inits variables
    int taskid, numtasks, numworkers, dest, mtype;  // MPI variables
    MPI_Status status;
    int rows, cols, averow, extra, offset, source;  // matrix variables
    cols = 5000;                                    // col size of input mat

    // inits MPI data and runs task error check
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    numworkers = numtasks - 1;

    // initializes I/O matricies
    const unsigned img_M = 5000;
    const unsigned img_N = 5000;
    int input[img_M][img_N];
    int output[img_M][img_N];

    /************************* Master Rank *************************/
    // sends matrix data to other ranks in the set
    if (taskid == MASTER) {

        // const unsigned img_M = 5000;        // removed from p1
        // const unsigned img_N = 5000;        // removed from p1
        // int input[img_M][img_N];            // removed from p1

        // loads input image matrix
        load_image("../input.txt", img_M, img_N, input);
        DBG(print_matrix(input, img_M, img_N);)

        // prints message confirming master rank had loaded in input.txt
        printf("rank %d has loaded in the image matrix\n", taskid);

        // inits parameters to send matix data to worker tasks
        averow = img_M/numworkers;
        extra = img_M%numworkers;
        offset = 0;

        // starts the timer
        struct timespec start = now();
```

As explained above, much of the default MPI configuration (including variable initialization and code to acquire total rank count and task ID) is left outside of the master region. The master region holds the code needed to load the input matrix and print debug messages. The master region additionally holds code required to start/stop the timer as to not have several conflicting timer calculations in the final output log file.

The master task begins by calculating the average row count per task and seeing if any tasks require any extra rows, similarly to Lab 2. Then, matrix chunk data correlating to an offset and a row count of the input matrix is sent to the worker tasks. After the worker tasks run the sobel algorithm on their respective chunks, data is received by the master and the final output matrix is saved as a text file. The final run time is additionally calculated. This is all shown below:

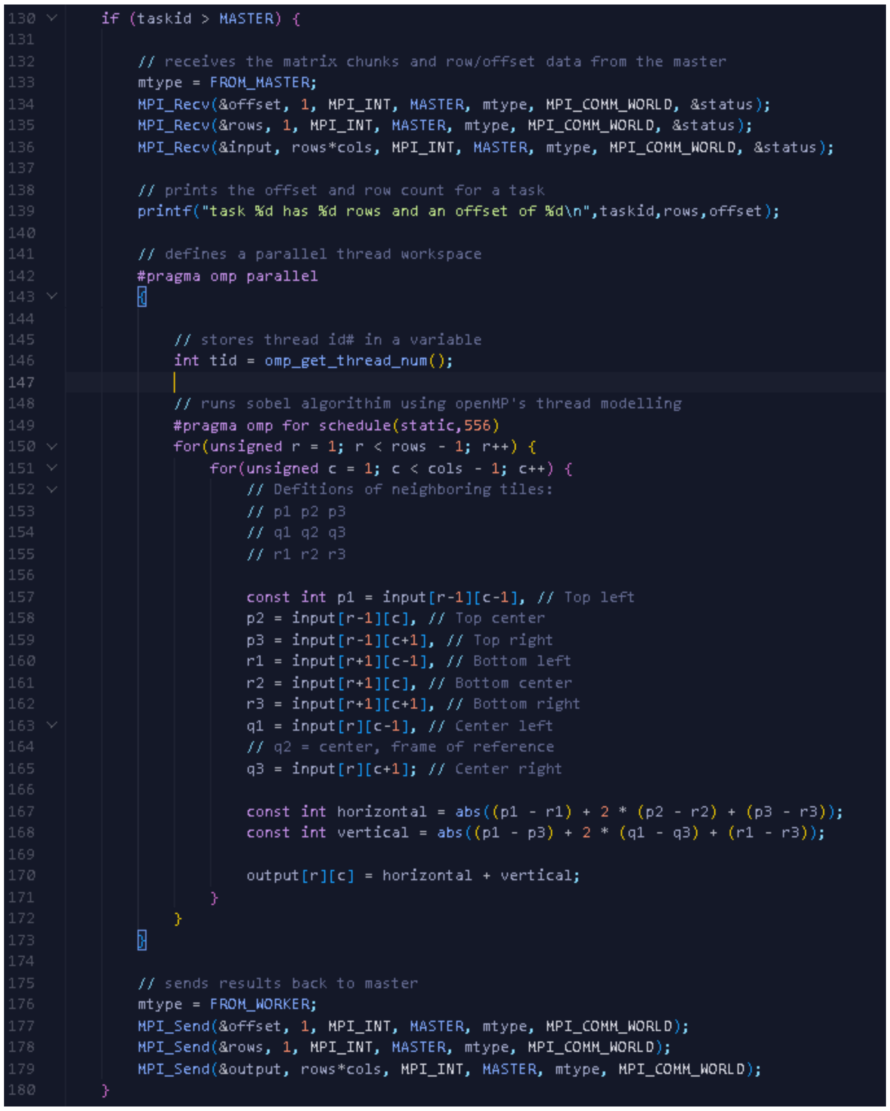


```c
        // sends matrix data to workers tasks
        mtype = FROM_MASTER;
        for (dest=1; dest<=numworkers; dest++) {
            rows = (dest <= extra) ? averow+1 : averow;
            printf("Sending %d rows to task %d offset=%d\n",rows,dest,offset);
            MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
            MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
            MPI_Send(&input[offset][0], rows*cols, MPI_INT, dest, mtype, MPI_COMM_WORLD);
            offset = offset + rows;
        }

        // recieves matrix data from workers tasks
        mtype = FROM_WORKER;
        for (int i=1; i<=numworkers; i++) {
            source = i;
            MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
            MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
            MPI_Recv(&output[offset][0], rows*cols, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
            printf("Received results from task %d\n",source);
        }

        // stops the timer
        struct timespec end = now();

        // prints output matrix and calculates elapsed runtime
        DBG(print_matrix(output, img_M, img_N);)
        save_image("../output.txt", img_M, img_N, output);
        double elapsed_time = tdiff(start, end);
        printf("Elapsed time: %.8f sec\n", elapsed_time);
    }
```

The worker task receives the sent input matrix chunk data and defines a parallel region. Through this, each task can run a set of threads to speed up calculation time by further parallelizing the application of the sobel filter to a task’s respective matrix chunk.

The tasks use the threads in the parallel region to update the output matrix. The resulting offset, row, and output matrix data is then sent back to the master task and received (shown above in the master region). The body of the worker region is shown below:

|
v



```c
if (taskid > MASTER) {

    // receives the matrix chunks and row/offset data from the master
    mtype = FROM_MASTER;
    MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&input, rows*cols, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);

    // prints the offset and row count for a task
    printf("task %d has %d rows and an offset of %d\n",taskid,rows,offset);

    // defines a parallel thread workspace
    #pragma omp parallel
    {

        // stores thread id# in a variable
        int tid = omp_get_thread_num();

        // runs sobel algorithim using openMP's thread modelling
        #pragma omp for schedule(static,556)
        for(unsigned r = 1; r < rows - 1; r++) {
            for(unsigned c = 1; c < cols - 1; c++) {
                // Defitions of neighboring tiles:
                // p1 p2 p3
                // q1 q2 q3
                // r1 r2 r3

                const int p1 = input[r-1][c-1], // Top left
                p2 = input[r-1][c], // Top center
                p3 = input[r-1][c+1], // Top right
                r1 = input[r+1][c-1], // Bottom left
                r2 = input[r+1][c], // Bottom center
                r3 = input[r+1][c+1], // Bottom right
                q1 = input[r][c-1], // Center left
                // q2 = center, frame of reference
                q3 = input[r][c+1]; // Center right

                const int horizontal = abs((p1 - r1) + 2 * (p2 - r2) + (p3 - r3));
                const int vertical = abs((p1 - p3) + 2 * (q1 - q3) + (r1 - r3));

                output[r][c] = horizontal + vertical;
            }
        }
    }

    // sends results back to master
    mtype = FROM_WORKER;
    MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
    MPI_Send(&output, rows*cols, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
}
```

The best configuration of MPI and OpenMP parameters to get the fastest run time was found through a series of tests. The tests were run in this ordering:

1. 1 node, 4 ranks
2. 2 nodes, 2 ranks, no socket specification
3. 2 nodes, 2 ranks, tasks/socket = 1
4. **2 nodes, 2 ranks, tasks/socket = 2**
5. 2 nodes, 2 ranks, tasks/socket = 3
6. 2 nodes, 2 ranks, tasks/socket = 4
7. 4 nodes, 1 rank, no socket specification

From the above, the fastest run time was found to be configuration 4 (2 nodes, 2 ranks, tasks/socket = 2). In attempts to further increase run time, OpenMP configurations were changed with configuration 4 acting as a baseline. This way, performance enhancements could be added to the already fastest configuration.

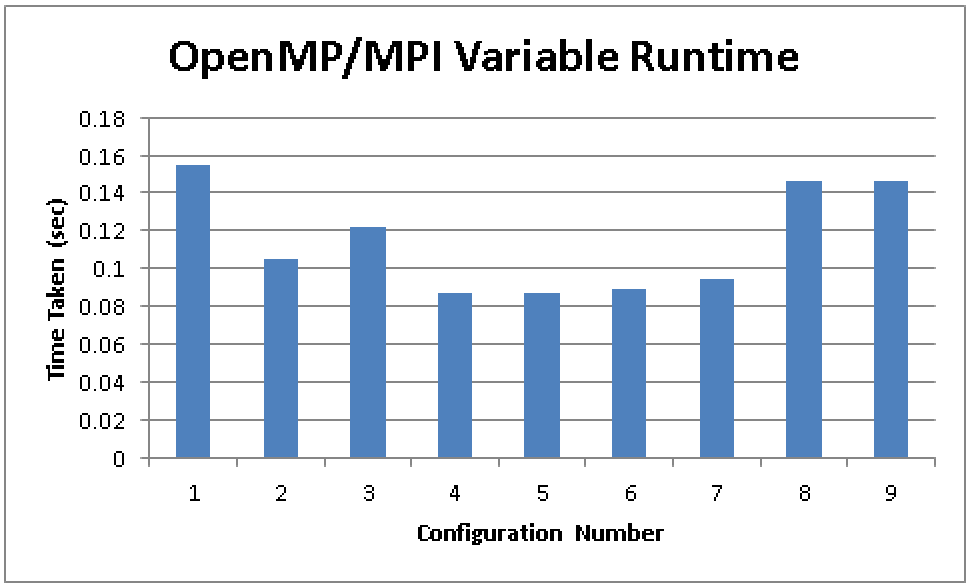
All above configurations started off with a default chunk size and static scheduling. To experiment and see if speeds could be increased, tests 7 and 8 were done as follows:

8. 2 nodes, 2 ranks, tasks/socket = 2, dynamic scheduling
9. 2 nodes, 2 ranks, tasks/socket = 2, static scheduling, chunksize = 556

The reason for the chunksize being 556 is to evenly divide each of the 3 worker tasks' matrix chunks into 3 further distinct chunks, hopefully increasing performance.

The results of all above tests are shown in the below table. The below graph holds the runtime for each configuration, with configuration 4 clearly having the fastest runtime:

Node	Rank	Socket	Scheduling	Chunk size	Time Taken (sec)
1	4	N/A	Static	N/A	0.15500379
2	2	N/A	Static	N/A	0.10484074
2	2	1	Static	N/A	0.12202996
2	2	2	Static	N/A	0.08702277
2	2	3	Static	N/A	0.08717561
2	2	4	Static	N/A	0.08898421
4	1	N/A	Static	N/A	0.09450152
2	2	2	Dynamic	N/A	0.14589309
2	2	2	Static	556	0.14584411



Therefore, based on the above justification with the various tests and charted data, it can be concluded that the configuration with 2 nodes, 2 ranks, 2 tasks/socket, and default/static scheduling (i.e. **configuration 4**) is the best performing configuration.

Final output image from Part 2: