Transient 2D Heat Conduction Solver Report

Gary Johnson, A02079587, gary.r.johnson@aggiemail.usu.edu
Jared Knutti, A02311340, jtknutti@gmail.com
Tyler Lott, A02230980, tyler.lott@usu.edu

12 April 2021

Project Description

This project was to apply the power of distributed computing to a real world example of heat transfer using finite element methods. Finite element methods create a mesh of a surface. The points of that mesh are used as computational points which are solved for using a discretized differential equation. This differential equation can be anything from fluid flow to heat transfer.

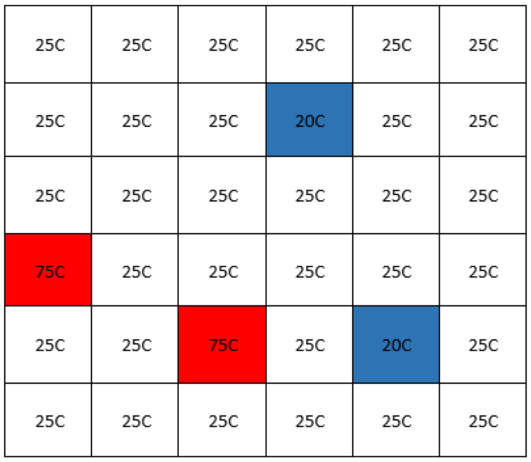
In our case we chose a 2d heat transfer equation, shown below.

$$q'' = -K(\frac{\delta T}{\delta x} + \frac{\delta T}{\delta y}) \tag{1}$$

This can be discretized into the following equation, where Fo is the Fourier number, T represents the mesh, t is the time step, and m and n are indexes of the T array.

$$T^{t+1}_{m,n} = F_o[T^t_{m+1,n} + T^t_{m-1,n} + T^t_{m,n+1} + T^t_{m,n-1}] + [1 - 4F_o]T^t_{m,n} \tag{2}$$

We also chose to use a simple square mesh, to make things simpler to distribute. We assumed insulated boundary conditions (meaning that there is no heat losses at the edges of the square). We also assumed boundary condition points on the plate that were of a constant temperature, we chose to use 4 of these points, two hot (75 degrees C), two cold (20 degrees C). We assumed all of the other points were at room temperature (25 degrees C).Using this information the mesh initialized was like the one below.



25C	25C	25C	25C	25C	25C
25C	25C	25C	20C	25C	25C
25C	25C	25C	25C	25C	25C
75C	25C	25C	25C	25C	25C
25C	25C	75C	25C	20C	25C
25C	25C	25C	25C	25C	25C

Though we would primarily be testing the mesh shown above and comparing it to a known solution, the program we aimed to create would have mesh size, timestep, distance between nodes of mesh, and thermal diffusivity of the material, as command line inputs to extend the usage of the solver to more cases and materials.

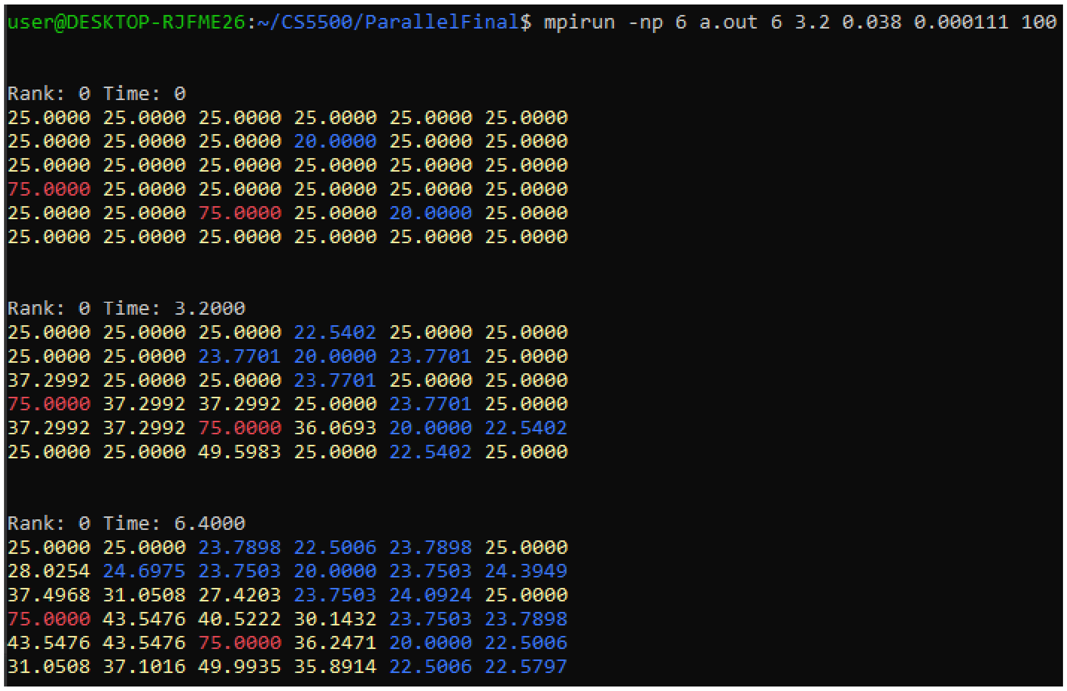
Overview of Effort

The program was separated into several parts to be completed. First, the mesh and boundary conditions initialization as well as the Fourier calculation to determine stability. Second, the data structures and communication architecture to be used. Third, visualization and comparison of results. Fourth testing and reporting on the findings.

The mesh was initialized by taking in command line arguments to determine the size of the mesh. The program uses a fixed square mesh. The simulation variables, time step, distance between nodes, and thermal diffusivity, and iterations to complete are also command line inputs. These are used by the program to calculate the Fourier number, which is central to the heat transfer calculation.

The second major part of this project was determining the communication architecture that was to be used. We decided to use a master-slave configuration. In this case, though, each process has a copy of the previous time step mesh. This way any process can calculate the row it is assigned by the master process. The processes are assigned a row, calculate the next time step for each node in the row, then send back the new row to the master. After each row is processed the master collects them all and distributes a new updated mesh to each process and repeats until the number of iterations is complete.

Visualization for our program was done through the command line using thresholds to color the output in three different categories, hot, cold, and neutral (red, blue, yellow). The program outputs the mesh in terminal as well as the time step and which process is printing (always master in the current implementation). The results from this are shown below.



```
user@DESKTOP-RJFME26:~/CS5500/ParallelFinal$ mpirun -np 6 a.out 6 3.2 0.038 0.000111 100


Rank: 0 Time: 0
25.0000 25.0000 25.0000 25.0000 25.0000 25.0000
25.0000 25.0000 25.0000 20.0000 25.0000 25.0000
25.0000 25.0000 25.0000 25.0000 25.0000 25.0000
75.0000 25.0000 25.0000 25.0000 25.0000 25.0000
25.0000 25.0000 75.0000 25.0000 20.0000 25.0000
25.0000 25.0000 25.0000 25.0000 25.0000 25.0000


Rank: 0 Time: 3.2000
25.0000 25.0000 25.0000 22.5402 25.0000 25.0000
25.0000 25.0000 23.7701 20.0000 23.7701 25.0000
37.2992 25.0000 25.0000 23.7701 25.0000 25.0000
75.0000 37.2992 37.2992 25.0000 23.7701 25.0000
37.2992 37.2992 75.0000 36.0693 20.0000 22.5402
25.0000 25.0000 49.5983 25.0000 22.5402 25.0000


Rank: 0 Time: 6.4000
25.0000 25.0000 23.7898 22.5006 23.7898 25.0000
28.0254 24.6975 23.7503 20.0000 23.7503 24.3949
37.4968 31.0508 27.4203 23.7503 24.0924 25.0000
75.0000 43.5476 40.5222 30.1432 23.7503 23.7898
43.5476 43.5476 75.0000 36.2471 20.0000 22.5006
31.0508 37.1016 49.9935 35.8914 22.5006 22.5797
```

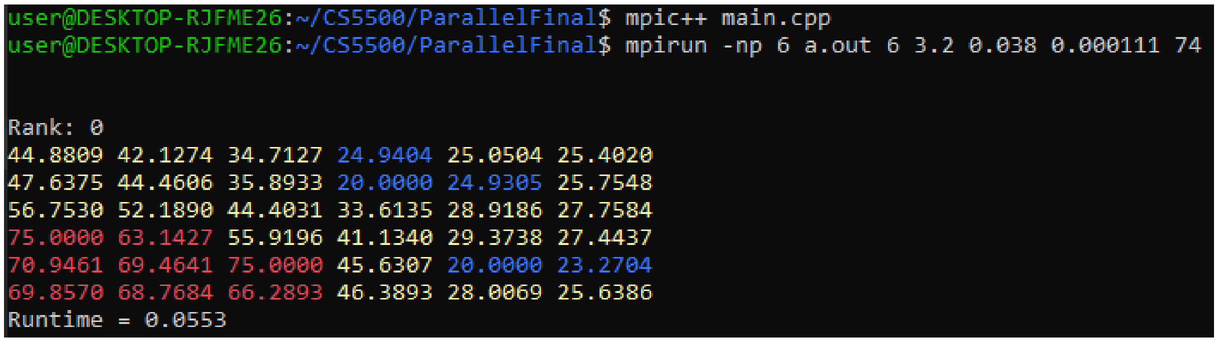
As the above example shows, the temperatures of each cell will gradually change based on the temperatures of the adjacent cells. Eventually the simulation will reach a steady state. This time stepping is what is advantageous of a transient model, we are able to see how the temperature in the plate develops. We are also able to solve some problems which have unstable steady state conditions. This case has a stable steady state, but some variations may reach a quasi steady state which fluctuates with some frequency, the final temperature may fluctuate between 30 and 35 degrees C with a sinusoidal function in time, this would not be able to be found by a steady state model, but can be modeled by a transient model.

Code Listing

All of the code is contained in the main.cpp file (Appendix A).

Description of Tests

First we made sure the code compiled and ran. The test and results for this are shown below.



```
user@DESKTOP-RJFME26:~/CS5500/ParallelFinal$ mpic++ main.cpp
user@DESKTOP-RJFME26:~/CS5500/ParallelFinal$ mpirun -np 6 a.out 6 3.2 0.038 0.000111 74


Rank: 0
44.8809 42.1274 34.7127 24.9404 25.0504 25.4020
47.6375 44.4606 35.8933 20.0000 24.9305 25.7548
56.7530 52.1890 44.4031 33.6135 28.9186 27.7584
75.0000 63.1427 55.9196 41.1340 29.3738 27.4437
70.9461 69.4641 75.0000 45.6307 20.0000 23.2704
69.8570 68.7684 66.2893 46.3893 28.0069 25.6386
Runtime = 0.0553
```

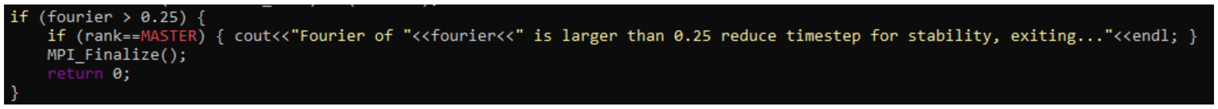
Command line arguments are:

- Mesh size
- Time step of simulation
- Distance between nodes in mesh
- Thermal diffusivity of the material
- Number of iterations

The second test was done in the program. We had to check for stability of the time step. The simulation is only computationally stable if the Fourier number is greater than 0.25 so this is checked as soon as the command line arguments are read in and the Fourier is calculated using the equation:

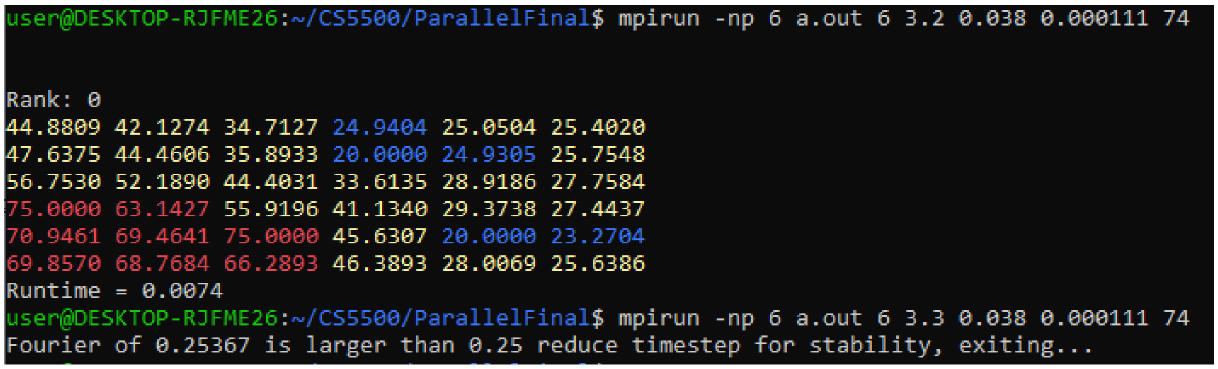
$$F_o = \frac{\alpha \triangle t}{\triangle L^2} \tag{3}$$

If the Fourier is greater than 0.25 the master process notifies the user and then the program exits.



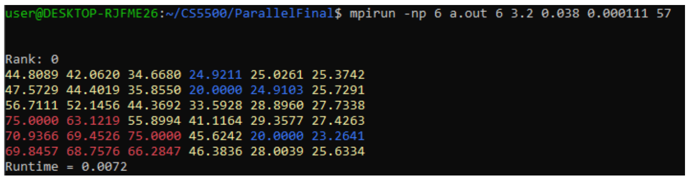
```
if (fourier > 0.25) {
    if (rank==MASTER) { cout<<"Fourier of "<<fourier<<" is larger than 0.25 reduce timestep for stability, exiting..."<<endl; }
    MPI_Finalize();
    return 0;
}
```

These are the results from a valid and invalid Fourier number.



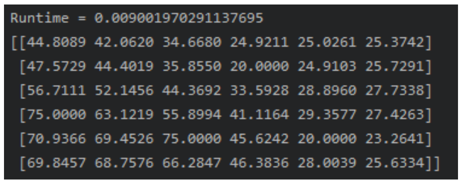
```
user@DESKTOP-RJFME26:~/CS5500/ParallelFinal$ mpirun -np 6 a.out 6 3.2 0.038 0.000111 74


Rank: 0
44.8809 42.1274 34.7127 24.9404 25.0504 25.4020
47.6375 44.4606 35.8933 20.0000 24.9305 25.7548
56.7530 52.1890 44.4031 33.6135 28.9186 27.7584
75.0000 63.1427 55.9196 41.1340 29.3738 27.4437
70.9461 69.4641 75.0000 45.6307 20.0000 23.2704
69.8570 68.7684 66.2893 46.3893 28.0069 25.6386
Runtime = 0.0074
user@DESKTOP-RJFME26:~/CS5500/ParallelFinal$ mpirun -np 6 a.out 6 3.3 0.038 0.000111 74
Fourier of 0.25367 is larger than 0.25 reduce timestep for stability, exiting...
```



```
user@DESKTOP-RJFME26:~/CS5500/ParallelFinal$ mpirun -np 6 a.out 6 3.2 0.038 0.000111 57


Rank: 0
44.8089 42.0620 34.6680 24.9211 25.0261 25.3742
47.5729 44.4019 35.8550 20.0000 24.9103 25.7291
56.7111 52.1456 44.3692 33.5928 28.8960 27.7338
75.0000 63.1219 55.8994 41.1164 29.3577 27.4263
70.9366 69.4526 75.0000 45.6242 20.0000 23.2641
69.8457 68.7576 66.2847 46.3836 28.0039 25.6334
Runtime = 0.0072
```



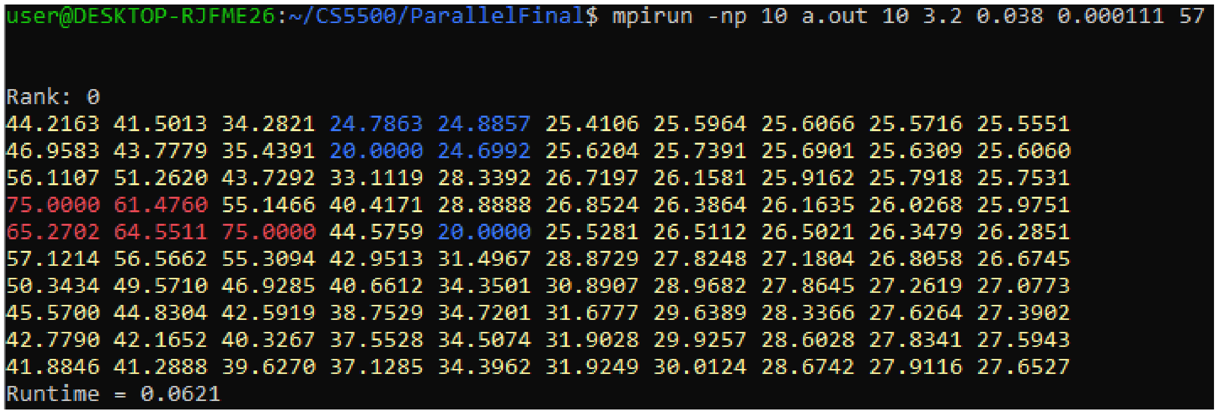
```
Runtime = 0.009001970291137695
[[44.8089 42.0620 34.6680 24.9211 25.0261 25.3742]
 [47.5729 44.4019 35.8550 20.0000 24.9103 25.7291]
 [56.7111 52.1456 44.3692 33.5928 28.8960 27.7338]
 [75.0000 63.1219 55.8994 41.1164 29.3577 27.4263]
 [70.9366 69.4526 75.0000 45.6242 20.0000 23.2641]
 [69.8457 68.7576 66.2847 46.3836 28.0039 25.6334]]
```

We then compared our results to the output of a solver written in python by Tyler that had been verified in a previous class.

Our project results (left). Verified solver results (right).

Seen in the results above, the parallel solver gets the exact same results as the verified solver. This is to be expected and means that both are numerically equivalent, despite being different in computation style. We also see that our program, in this case, performed the calculations a bit faster than the other solver.

The next tests we ran were to determine if the model could scale to larger meshes. We tested 6x6, 10x10, 100x100, and 1000x1000 meshes. Shown below is the 10x10 mesh, after this the models were too large to view in the terminal. For this size the mesh should be output and visualized in other software, thus they are not shown here. They were ran though and computed without any issues.



```
user@DESKTOP-RJFME26:~/CS5500/ParallelFinal$ mpirun -np 10 a.out 10 3.2 0.038 0.000111 57


Rank: 0
44.2163 41.5013 34.2821 24.7863 24.8857 25.4106 25.5964 25.6066 25.5716 25.5551
46.9583 43.7779 35.4391 20.0000 24.6992 25.6204 25.7391 25.6901 25.6309 25.6060
56.1107 51.2620 43.7292 33.1119 28.3392 26.7197 26.1581 25.9162 25.7918 25.7531
75.0000 61.4760 55.1466 40.4171 28.8888 26.8524 26.3864 26.1635 26.0268 25.9751
65.2702 64.5511 75.0000 44.5759 20.0000 25.5281 26.5112 26.5021 26.3479 26.2851
57.1214 56.5662 55.3094 42.9513 31.4967 28.8729 27.8248 27.1804 26.8058 26.6745
50.3434 49.5710 46.9285 40.6612 34.3501 30.8907 28.9682 27.8645 27.2619 27.0773
45.5700 44.8304 42.5919 38.7529 34.7201 31.6777 29.6389 28.3366 27.6264 27.3902
42.7790 42.1652 40.3267 37.5528 34.5074 31.9028 29.9257 28.6028 27.8341 27.5943
41.8846 41.2888 39.6270 37.1285 34.3962 31.9249 30.0124 28.6742 27.9116 27.6527
Runtime = 0.0621
```

Once verifying that the program was getting the correct results and was able to take in any mesh size, we focused on the performance of the program. We tested the program in two ways. First, we looked at the performance of the 100x100 mesh when distributed onto many cores (up to 12). The results for this are shown below.

No. cores	Runtime (s)
2	0.7384
4	0.3562
6	0.2797
8	0.2152
10	0.2908
12	0.2726

The results of this test show a sweet spot around 8 cores, where the run time is the shortest. This is because as the number of cores is increased the communication between cores becomes more of a time consuming task than the actual computation that the cores are doing. This may be able to be reduced by splitting the entire mesh into n number of sections and only communicating once done. That way each core only communicates with the master process once, reducing the number of sends and receives, compared to our architecture that sends the mesh line by line to the next available process.

The next time test we did was to compare the verified solver to our distributed program. A little background on the verified solver, it is built in python and uses linear algebra to solve directly for all parts of the mesh at once. Because of this, it would be expected that it will be faster, though it is limited by RAM, it has to fit multiple arrays of the size of the mesh into memory. For this reason we expected our program to have a larger mesh threshold, but perform slower than the verified model, especially at large mesh sizes. The results of the comparison are shown below.

Mesh Size	Verified Solver Time (s)	Distributed Solver Time (s)
6x6	0.0129	0.0066
10x10	0.0110	0.0098
100x100	0.0159	0.2090
1000x1000	0.3680	10.442

We see that the program is faster than the python solver in the small meshes but quickly gets slower and slower in comparison. It is likely that because of the overhead of python our program was able to be quicker in the first cases. As the mesh grew the speed of the direct solve vs iterative solving in our program became present. We also found that we couldn't max the mesh size of the python program without both programs becoming too slow to do any analysis at all. Therefore it would appear that for practical purposes, this architecture is easily beaten by a direct solve approach.

Concluding Remarks

In conclusion we were able to implement a distributed transient 2d heat conduction solver that was capable of getting results equivalent to a verified solver. Each process had access to a vector that represented a copper plate as well as steady temperature nodes. We utilized a master-slave configuration to distribute the rows within the vector to different processes. Processes iterated through the points on the node and solved for the temperature of each node in the row, then sent the row back to the master. Once each row was complete the master process collected all of the new rows and redistributed the vector representing the plate to each process, This is the end of one time step. This was repeated for the number of iterations that the user input, which, ideally, would be enough to solve until the plate reached a steady state temperature solution.

We found that the distributed program solution we had created was numerically as accurate as the verified solver used to compare against. We also found that our solver was faster with small mesh sizes but was drastically affected by increasing mesh sizes. Therefore this architecture is not viable for large scale solutions. After thinking of improvements we came to the conclusion that if we were to repeat this project we would use an architecture that split the mesh at the very beginning into n number of sections, and each section would only send its top row to the process above, and its bottom row to the process below. This would allow a ring pass like architecture to be applied and reduce the number of centralized passes. This would work because the computation only required the row above and below to compute any given node. This would likely be a more memory efficient as well as communication efficient way to complete the problem.

Appendix A

```
#include <mpi.h>
#include <chrono>
#include <vector>
#include <iomanip>
#include <algorithm>

using namespace std;

#define MCW MPI_COMM_WORLD
#define MASTER 0

//Tags
#define WORK 1
#define TERMINATE 2

const char * getColor(float temp){
    if (temp < 25) { return "\033[1;34m"; }
    if (temp >= 25 && temp <= 60) { return "\033[1;33m"; }
    if (temp > 60) { return "\033[1;31m"; }
    return "\033[0m";
}

// DEBUG helper function to check the values in the mesh
void printMesh(vector<vector<float>> mesh, int rank, float timestep) {
    cout << string(2,'\n');
    cout << "Rank:␣" << rank << "␣Time:␣"<<timestep<<endl;
    for (size_t row = 0; row < mesh.size(); row++) {
        for (size_t col = 0; col < mesh[row].size(); col++) {
            const char* color = getColor(mesh[row][col]);
            cout << std::fixed << std::setprecision(4) << color << mesh[row][col] << "␣" << "\033[0m";
        }
        cout << endl;
    }
}

// Sets the hot and cool constant temps to random cells
void setTemps(vector<vector<float>>& mesh, vector<vector<int>> hotCells, vector<vector<int>> coolCells, int rank, float hotTemp, floa

    // set the hot and cool cell coordinates
    for (int cell = 0; cell < hotCells.size(); cell++) {

        MPI_Bcast(hotCells[cell].data(), hotCells[cell].size(), MPI_INT, MASTER, MCW);
        MPI_Bcast(coolCells[cell].data(), coolCells[cell].size(), MPI_INT, MASTER, MCW);

        // Set constant temperature cells accordingly
        mesh[hotCells[cell][0]][hotCells[cell][1]] = hotTemp;
        mesh[coolCells[cell][0]][coolCells[cell][1]] = coolTemp;
    }
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    double start, end;
    int rank, size;
    MPI_Comm_rank(MCW, &rank);
    MPI_Comm_size(MCW, &size);
    // take in the command line variables
    // argv[1] = mesh_size
    // argv[2] = time step
    // argv[3] = distance between nodes
    // argv[4] = thermal diffusivity of the material
    // argv[5] = number of iterations
    int mesh_size, iterations;
    sscanf(argv[1], "%d", &mesh_size);
    sscanf(argv[5], "%d", &iterations);
    float dt, dx, therm_diff;
    sscanf(argv[2],"%f", &dt);
    sscanf(argv[3],"%f", &dx);
    sscanf(argv[4],"%f", &therm_diff);
//    float dt = 1.5, dx = 0.038, therm_diff = 0.000111; // dt-time step, dx-node separation, thermal diffusivity is for copper
    float fourier = (dt * therm_diff) / (dx * dx); // FOURIER MUST BE < 0.25 FOR STABILITY
    if (fourier > 0.25) {
        if (rank==MASTER) { cout<<"Fourier␣of␣"<<fourier<<"␣is␣larger␣than␣0.25␣reduce␣timestep␣for␣stability,␣exiting..."<<endl; }
        MPI_Finalize();
        return 0;
    }
    float hot = 75, cold = 20;

    srand(chrono::system_clock::now().time_since_epoch().count());  // seed rand() by current time

    vector<vector<float>> mesh(mesh_size, vector<float>(mesh_size, 25));  // initialize the mesh with the base temperature

    vector<vector<int>> hotCells{ {3, 0} , {4, 2} };  // mesh coordinates of the hot and cool cells
    vector<vector<int>> coolCells{ {1, 3} , {4, 4} };

    setTemps(mesh, hotCells, coolCells, rank, hot, cold);
    MPI_Barrier(MCW);

    start = MPI_Wtime();

    int timeStep = 0;
    while (timeStep < iterations) {

        if (rank == MASTER) {
            printMesh(mesh, rank, timeStep*dt);
            int currentRow = 0;
            int workingProcesses = 0;

            // Begin sending new work to each worker
            for (int worker = 1; worker < size; worker++) {
                // sends index
                MPI_Send(&currentRow, 1, MPI_INT, worker, WORK, MCW);
                currentRow++;
                workingProcesses++;
            }

            int rows = mesh.size();
            MPI_Status workerStatus;

            // Send rows until all rows have been completed
            while (workingProcesses > 0) {
                vector<float> receivedRow(mesh[0].size());

                // Receive row of work from worker. The row index is received as the tag
                MPI_Recv(receivedRow.data(), receivedRow.size(), MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MCW, &workerStatus);
                workingProcesses--;

                mesh[workerStatus.MPI_TAG] = receivedRow;
                int readyWorker = workerStatus.MPI_SOURCE;  // Send next row to worker that just finished last one

                if (currentRow < rows) {
                    // sends index
                    MPI_Send(&currentRow, 1, MPI_INT, readyWorker, WORK, MCW);
                    currentRow++;
                    workingProcesses++;
                } else {
                    MPI_Send(NULL, 0, MPI_INT, readyWorker, TERMINATE, MCW);
                }
            }
        } else {  // workers
            int row;
            bool working = true;
            MPI_Status workStatus;
            while (working) {
                // receives index.
                MPI_Recv(&row, 1, MPI_INT, MASTER, MPI_ANY_TAG, MCW, &workStatus);
                if (workStatus.MPI_TAG == TERMINATE) {
                    working = false;
                } else {
                    vector<float> updatedRow(mesh[0].size());
                    float a, b, c, d;

                    for (int i=0; i<mesh[0].size(); i++) {
                        vector<int> key = {row, i};
                        if ((std::find(hotCells.begin(), hotCells.end(), key) != hotCells.end()) ||
                             (std::find(coolCells.begin(), coolCells.end(), key) != coolCells.end())) { // check to see if a constant
                            updatedRow[i] = mesh[row][i];
                        } else {
                                if (row == 0) { // boundary conditions at edges is insulated meaning we use the inner point twice
                                    a = mesh[row+1][i];
                                } else {
                                    a = mesh[row-1][i]; }
                                if (row == mesh[0].size()-1) {
                                    b = mesh[row-1][i];
                                } else {
                                    b = mesh[row+1][i]; }
                                if (i == 0) {
                                    c = mesh[row][i+1];
                                } else {
                                    c = mesh[row][i-1]; }
                                if (i == mesh[0].size()-1) {
                                    d = mesh[row][i-1];
                                } else {
                                    d = mesh[row][i+1]; }
                                updatedRow[i] = fourier * (a + b + c + d) + (1 - 4 * fourier) * mesh[row][i];
                        }
                    }

                    // Send updated row back to master with the row index as the tag
                    MPI_Send(updatedRow.data(), updatedRow.size(), MPI_FLOAT, MASTER, row, MCW);
                }
            }
        }

        timeStep++;

        // Update each process's mesh
        for (int row = 0; row < mesh.size(); row++) {
            MPI_Bcast(mesh[row].data(), mesh[row].size(), MPI_FLOAT, MASTER, MCW);
            MPI_Barrier(MCW);
        }
    }

    MPI_Finalize();

    end = MPI_Wtime();
    if (rank==MASTER) { printMesh(mesh,rank, iterations*dt); cout<<"Runtime = "<<end-start<<endl; }

    return 0;
}
```