Heat Equation

Assignment Two

This declaration certifies that this project was written by us and in our own words, except for the use of code from your own examples as we saw fit to take. We are conscious that the incorporation of material from other works or paraphrases of such material without acknowledgement will be treated as plagiarism, subject to the custom and usage of the subject according to the University Regulations on Conduct of Examinations.

Heat Equation

Assignment Two

# Modeling and Analysis of the Heat Equation

# Introduction

Given the initial assignment specifications, our objective was to model the heat equation a set of PDEs into a system of ODEs and represent it visually. The model had to show how heat transferred through objects taking into account a number of constraints such as object type, density, energy input, temperature, etc. Our visualizations had to include working Matlab code as well as code in C++. Furthermore, our C++ code had to include both a serial solution as well as a solution using the OpenMP and MPI APIs that took advantage of shared and distributed memory structures. Finally, we had to visualize the C++ code in Paraview using the outputted VTK file format.

We went through the assignment in a straightforward manner as per assignment specification. Most of the code was easily available in lecture examples 15.3 and 21.4 which became the basis for our analysis. The use of Sparse matrices to save space was seen, which I thought was a novel idea. Plotting a tetrahedral structure in Matlab also proved tricky as we had difficulty choosing between the ‘trisurf’ and ‘tetrahedral’ functions. Then there was the failure of Paraview to function properly when we tried opening up VTK files in it.

But we’ll get to that as you move forward in the report. For now, we present to you our results and analysis of Assignment 2 – modeling the Heat equation.

# Derivation

Taking our initial equation: -

We first apply the weak form derivation:

Next we expand the equation:

With the result, we then use Einstein summation notation:

And Re-arrange this equation to get:

Then we rearrange the function by adding in the weight functions to get:

And then

So while p and q are in the range 1 to 4 for our tetrahedron element, the overall system modeling equation becomes:

From the given information, **Ne** will be the total number of elements, and **Qcpu** and **Tair** are both constants.

With that in play we then re-arrange the equation in the form of our standard format:

Which will give us:

The derivations shown above are the discretization part of our partial differential equations; the next step is determining the key values that will be put into the formula:

To calculate the integration of the shape function for the M and K matrices in our standard format, it requires the use of the integration formulae for the integration of the shape function:

The integration formulas are defined as:

And

Where is the volume of the tetrahedron volume and is the tetrahedron face area of the element.

M is a 4 by 4 matrix because p and q are in the range from 1 to 4. In the case when p and q are equal we’d use:

And where p and q are not equal:

So the M matrix of each element will be:

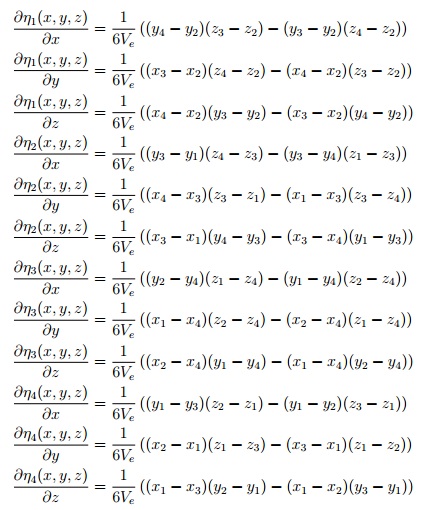
And is the volume of the tetrahedron element.

When calculating the robin condition part of the K matrix, we used the integration formulae for the surface area, and as we know the each surface area is a triangular shape so there will be three points that make up a face. In this case the p and q values are in the range from 1 to 3, when p and q values are equal:

And when p and q are not equal:

And the matrix of the shape function will be the 3 by 3 matrix, so this part of the matrix will be

- for the part of the K matrix. Taken from lecture notes on page 349, to calculate, the above will follow the following derivations:



To simplify our effort whilst coding, we also made a G matrix that would perform all of the elements calculation of each point of the element. Thus, the G matrix for the tetrahedron element is given by:

G=

[((y(4)-y(2))\*(z(3)-z(2))-(y(3)-y(2))\*(z(4)-z(2))), ((y(3)-y(1))\*(z(4)-z(3))-(y(3)-y(4))\*(z(1)-z(3))), ((y(2)-y(4))\*(z(1)-z(4))-(y(1)-y(4))\*(z(2)-z(4))), ((y(1)-y(3))\*(z(2)-z(1))-(y(1)-y(2))\*(z(3)-z(1)));

((x(3)-x(2))\*(z(4)-z(2))-(x(4)-x(2))\*(z(3)-z(2))), ((x(4)-x(3))\*(z(3)-z(1))-(x(1)-x(3))\*(z(3)-z(4))), ((x(1)-x(4))\*(z(2)-z(4))-(x(2)-x(4))\*(z(1)-z(4))), ((x(2)-x(1))\*(z(1)-z(3))-(x(3)-x(1))\*(z(1)-z(2)));

((x(4)-x(2))\*(y(3)-y(2))-(x(3)-x(2))\*(y(4)-y(2))), ((x(3)-x(1))\*(y(4)-y(3))-(x(3)-x(4))\*(y(1)-y(3))), ((x(2)-x(4))\*(y(1)-y(4))-(x(1)-x(4))\*(y(2)-y(4))), ((x(1)-x(3))\*(y(2)-y(1))-(x(1)-x(2))\*(y(3)-y(1)))];

In the loop, p and q will represent two points in the element and the calculation for each element of this part would be:

Looping all 4 points in the element for p from 1 to 4 and q from 1 to 4

Gp = [G(1,p), G(2,p),G(3,p)];

Gq = [G(1,q), G(2,q),G(3,q)];

for calculating the weight function of s where we only included the shape integration formulae:

The final part of the derivation before coding was to calculate the volume of tetrahedron element and face areas of each face in the element:

% Area of a triangular element with coordinates

% (x1, y1, z1), (x2, y2, z2), (x3, y3, z3):

Gamma = sqrt(((y2-y1)\*(z3-z1) - (z2-z1)\*(y3-y1))^2 ...

+ ((z2-z1)\*(x3-x1) - (x2-x1)\*(z3-z1))^2 ...

+ ((x2-x1)\*(y3-y1) - (y2-y1)\*(x3-x1))^2)/2;

and

% Volume of a tetrahedral element with coordinates

% (x1, y1, z1), (x2, y2, z2), (x3, y3, z3), (x4, y4, z4):

Omega = abs( x1\*y2\*z3 - x1\*y3\*z2 - x2\*y1\*z3 ...

+ x2\*y3\*z1 + x3\*y1\*z2 - x3\*y2\*z1 ...

- x1\*y2\*z4 + x1\*y4\*z2 + x2\*y1\*z4 ...

- x2\*y4\*z1 - x4\*y1\*z2 + x4\*y2\*z1 ...

+ x1\*y3\*z4 - x1\*y4\*z3 - x3\*y1\*z4 ...

+ x3\*y4\*z1 + x4\*y1\*z3 - x4\*y3\*z1 ...

- x2\*y3\*z4 + x2\*y4\*z3 + x3\*y2\*z4 ...

- x3\*y4\*z2 - x4\*y2\*z3 + x4\*y3\*z2 ) /6;

## MATLAB ANALYSIS:

After discretize the original equation by Finite Element Method, the ODE of the original equation will be solved by implicit Euler’s method. In the MATLAB analysis, the discretized format of the original equation will be in the form of:

M, K and s will be constructed in the CPU\_HEAT function by looping over all of the tetrahedron elements with analysis on the boundary conditions.

The first part of the MATLAB analysis is to write a function that can return M, K and s values. Initially, the volume of each tetrahedron element and surface area of each surface are calculated and stored in arrays. Then M and part of the K will be calculated by looping around all elements and based on the geometrical information of that particular element.

For the K matrix construction, refer to the derivation section, part is one part of the Robin condition of the CUP fin. With consideration on both the Neumann and Robin boundary faces, the s values and will be calculated by looping all of the faces of the sides with boundary conditions.

Implicit Euler’s method is applied in the time marching loop that analyzed the equation from 0s to 100s. By applying Implicit Euler’s method, the equation will be transformed into the form

which can be written in the form:

And

In the time marching loop T could be caudated from T values from previous time step by calculating T=b/A. The MATLAB analysis is calculated on the BOX grid and the final result are plotted in figure 1,2 and 3.

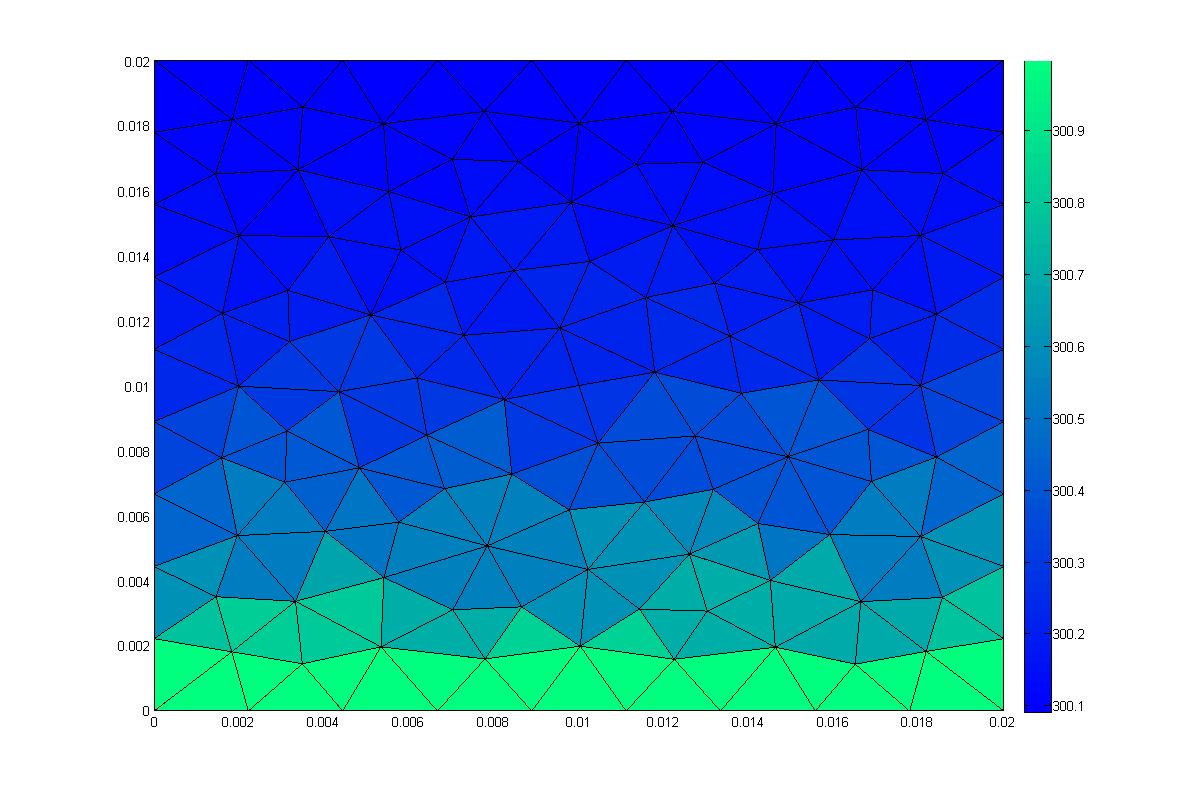


Figure 1 Neumann boundary plot at t~0s

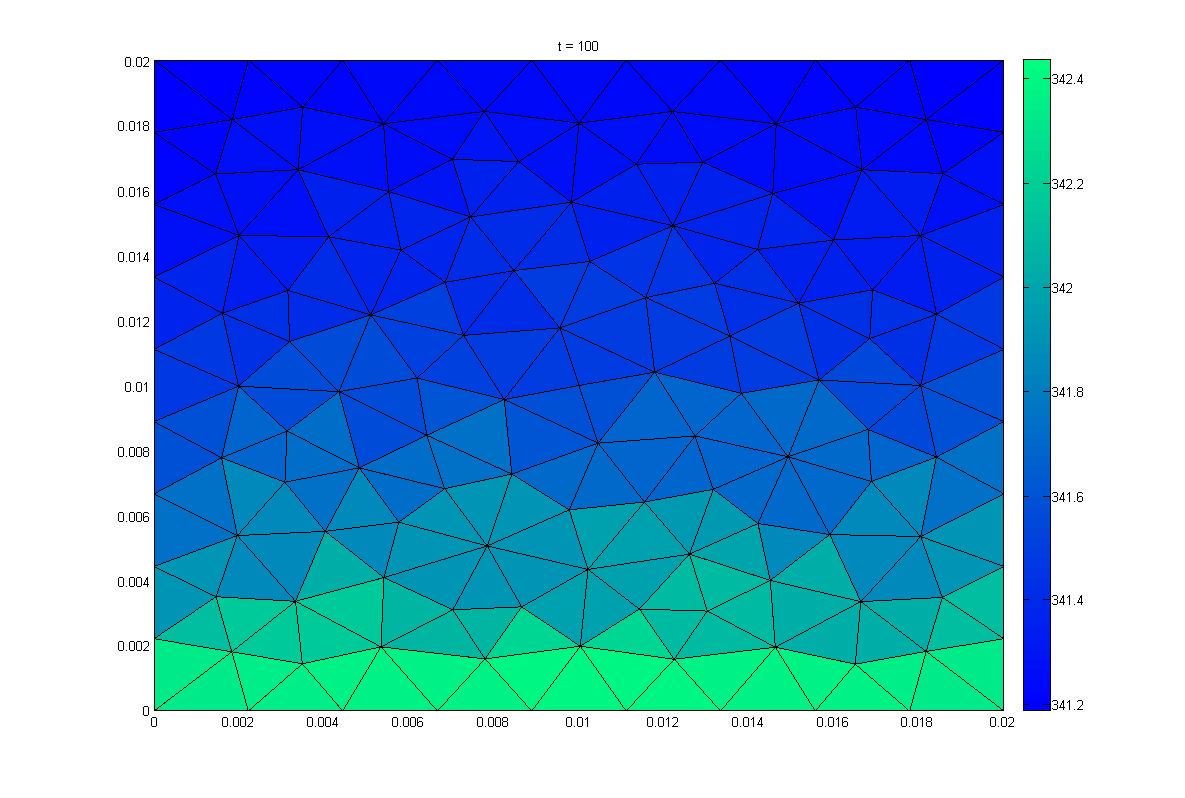


Figure 2 Neumann boundary plot at t=100s

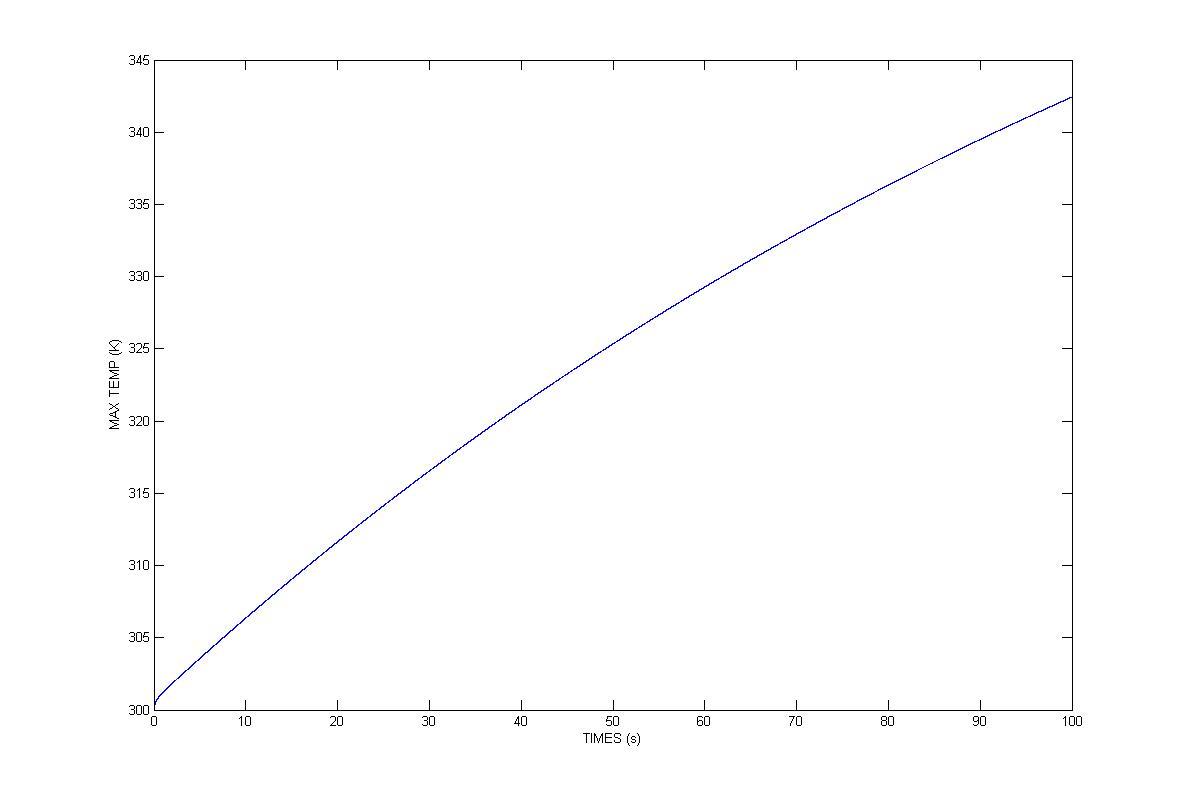


Figure 3 Maximum temperature vs. time

## SERIAL C++ ANALYSIS:

With Matlab working and out of our way, we now had the task of converting our Matlab code into workable C++ code. Furthermore, we had to achieve a peak temperature of around 342 degreesas shown in our Matlab model visualization.

Converting the code was pretty straightforward. Most of the code such as the class definitions were available from Ex 21.4 and were put as they were. However, because we were using the Robin instead of Dirichlet boundary conditions, the use of variable points such as fixed or free was not required. As such, we removed them from the program and modified our functions.

Unlike example 21.4 where we did not have a tetrahedron to model, the number of points, faces, and elements would increase by one. As mentioned in the derivation we’d now have 4 elemental nodes with 3 faces and 3 points each(x, y, z). These changes can be seen in our readData, writeData, and assembleSystem functions, examples of which are provided below:

for(int p=0; p<N\_p; p++)

{

file << setw(6) << setprecision(5) << Points[p][0] << "\t" << Points[p][1] << "\t" << Points[p][2] << "\t" << T[p] << endl;

}

file << "CELLS " << N\_e << " " << 4\*N\_e << endl;

for(int e=0; e<N\_e; e++)

{

file << "3\t" << Elements[e][0] << "\t" << Elements[e][1] << "\t" << Elements[e][2] << "\t" << Elements[e][3] << endl;

}

### Figure 3.1 writeData function showing the use of 3 Points and 4 Elements per node

for(int e=0; e<N\_e; e++)

{

for(int p=0; p<4; p++)

{

x[p] = Points[Elements[e][p]][0];

y[p] = Points[Elements[e][p]][1];

z[p] = Points[Elements[e][p]][2];

}

Omega[e] = Omega[e] = fabs( x[0]\*y[1]\*z[2] - x[0]\*y[2]\*z[1] - x[1]\*y[0]\*z[2]

+ x[1]\*y[2]\*z[0] + x[2]\*y[0]\*z[1] - x[2]\*y[1]\*z[0]

- x[0]\*y[1]\*z[3] + x[0]\*y[3]\*z[1] + x[1]\*y[0]\*z[3]

- x[1]\*y[3]\*z[0] - x[3]\*y[0]\*z[1] + x[3]\*y[1]\*z[0]

+ x[0]\*y[2]\*z[3] - x[0]\*y[3]\*z[2] - x[2]\*y[0]\*z[3]

+ x[2]\*y[3]\*z[0] + x[3]\*y[0]\*z[2] - x[3]\*y[2]\*z[0]

- x[1]\*y[2]\*z[3] + x[1]\*y[3]\*z[2] + x[2]\*y[1]\*z[3]

- x[2]\*y[3]\*z[1] - x[3]\*y[1]\*z[2] + x[3]\*y[2]\*z[1] ) /6;

}

### Figure 3.2 showing part of the assembleSystem function where we calculate the Elemental areas by looping over all x,y,z coordinates for all 4 elements

The assembleSystem function assembles values read in from the data file into our standard M = KT + s function. After looping over all Elements to calculate Face and Elemental areas, it assembles the values for M, K, and s matrices.

// Outer loop over each node

for(int p=0; p<4; p++)

{

m = Nodes[p];

Gp[0] = G[0][p];

Gp[1] = G[1][p];

Gp[2] = G[2][p];

// Inner loop over each node

for(int q=0; q<4; q++)

{

n = Nodes[q];

Gq[0] = G[0][q];

Gq[1] = G[1][q];

Gq[2] = G[2][q];

M(m, n) += M\_e[p][q]\*Omega[e]\*roh\*cap/20.0

K(m, n) -= k\*(Gp[0]\*Gq[0]+Gp[1]\*Gq[1]+Gp[2]\*Gq[2])/(36.0\*Omega[e]);

}

}

### Figure 3.3 shows how matricies M and K are assembled within the assembleSystem function.

Following the formation of these matrices we decided to apply boundary conditions. As shape coordinates and values are read in from our grid file, using the boundary condition segment we define which coordinates are of type “Neumann” and which are of type “robin” through a series of If statements. Following this, we’d mark their nodes by looping over each individual face and thus add them to our Node matrices.

Finally, we’d call our Solve matrix which would calculate the initial residue and solve our heat equation via the Conjugate gradient method.

// Compute the initial residual

A.multiply(AT, T);

for(m=0; m<N\_row; m++)

{

r\_old[m] = b[m] - AT[m];// residual

d[m] = r\_old[m];

r\_oldTr\_old+= r\_old[m]\*r\_old[m];

}

r\_norm = sqrt(r\_oldTr\_old);

### Figure 3.4 shows how we calculate the initial residual

The output of our program was given in terms of a series of VTK files that took in coordinate point values, Elements values, and the temperature gradients throughout the system. It was observed that the highest temperature noted in our VTK files was around 341 degrees, a reasonably close estimate.

## PARALLIZED C++ Analysis

With working C++ code, from henceforth we’d refer to as ‘serial code’; we now had the task of trying to optimize it. At our disposal were C++ APIs called Open MP and MPI which specialized in parallelizing code to make it run faster on machines with multiple cores and large amounts of RAM. OpenMP was an API that accomplished this by using shared memory architecture. Multiple process threads could access the same memory pool of data and process code to quickly come to an output. MPI on the other hand was an API that used distributed memory architecture, designed specifically for distributed computer environments. Basically data was stored over a variety of memory locations and was accessed by individual threads through a series of calls (sends and receives).

Our objective in this section of the assignment was to take our serial code and implement an OpenMPI design, followed by an analysis of scaling runs after optimizing the code. Unlike assignment 1 where we had to design a system of sends and receives to solve our shallow water equation, in this case, most of the MPI code was readily available in example 21.4, which we took and implemented in our design.

We realized that there were no free or fixed bools within our derivation subject to the boundary conditions, so as such we removed them from both the functions and the improvised MPI code. After such we were left with the task of renaming the seral code’s variables such as N\_e and N\_p to variables such as myN\_e and myN\_p.

In the main function, after initializing the following functions

**MPI\_Init(&argc, &argv);**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myID);**

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &N\_Processes);**

Depending on the number of N\_processes we set in the srun command, the MPI divides the program into separate threads with distributed memory and their own data. The MPI\_Comm\_rank sets the process threads apart via a separate MyID (rank). The only way these functions can communicate with each other is at the common boundary points using a function called exhangeData. This function allows for communication between boundary points and elements that were originally next to each other, but for the sake of an MPI implementation, our grid was broken apart into several smaller grids splitting these up. The borders of each of these smaller grids would now need to communicate with each other to ensure data uniformity when showing the movement of heat across the whole structure. That’s where the exchangeData function comes in to ensure that our new “inter-process” boundaries maintain a locational uniformity amongst them and each other that will also allow for accurate mapping of heat through the whole grid.

void exchangeData(double\* v, Boundary\* Boundaries, int myN\_b) {

int yourID = 0;

int tag = 0;

MPI\_Status status;

#pragma omp parallel default(shared) private(m)

{

#pragma omp for schedule(static)

for (int b = 0; b < myN\_b; b++) {

if (Boundaries[b].type\_ == "interprocess") {

for (int p = 0; p < Boundaries[b].N\_; p++) {

buffer[p] = v[Boundaries[b].indices\_[p]];

}

yourID = static\_cast<int>(Boundaries[b].value\_);

MPI\_Bsend(buffer, Boundaries[b].N\_, MPI\_DOUBLE, yourID, tag,

MPI\_COMM\_WORLD);

}

}

#pragma omp for schedule(static)

for (int b = 0; b < myN\_b; b++) {

if (Boundaries[b].type\_ == "interprocess") {

yourID = static\_cast<int>(Boundaries[b].value\_);

MPI\_Recv(buffer, Boundaries[b].N\_, MPI\_DOUBLE, yourID, tag,

MPI\_COMM\_WORLD, &status);

for (int p = 0; p < Boundaries[b].N\_; p++) {

v[Boundaries[b].indices\_[p]] += buffer[p];

}

}

}

### Figure 4.1 showing the exchangeData function that maintains node uniformity between broken up grid points

We implemented OpenMP over the MPI by introducing a number of slave threads that would work under each Master MPI thread. The master can identified via its MPI rank myID, which other than 0 would be a worker styled master thread that would handle one section of the grid with its own memory access.

Most of our OpenMP centered around the solve function where we created a series of (m) worker threads in the beginning of the function using the command –

**#pragma omp parallel default(shared) private(m)**

Followed by the command

**#pragma omp for schedule(static)**

for each successive for loop and the

**#pragma omp single**

Command for any part of the code that we couldn’t divide over a number of threads. We implemented the same techniques in the exchangeData function when we were cycling over “inter-process” nodes, an example of which can be seen below:

#pragma omp parallel default(shared) private(m)

{

for(m=0; m<N\_row; m++)

{

r\_old[m] = b[m] - AT[m];// residual

d[m] = r\_old[m];

// r\_oldTr\_old+= r\_old[m]\*r\_old[m];

}

#pragma omp single

{

r\_oldTr\_old = computeInnerProduct(r\_old, r\_old, yourPoints, N\_row);

r\_norm = sqrt(r\_oldTr\_old);

}

// Conjugate Gradient iterative loop

while(r\_norm>tolerance && k<maxIterations){

#pragma omp single

{

A.multiply(Ad, d);

exchangeData(Ad, Boundaries, myN\_b);

dTAd = computeInnerProduct(d, Ad, yourPoints, N\_row);

}

alpha = r\_oldTr\_old/dTAd;

#pragma omp for schedule(static)

for(m=0; m<N\_row; m++)

{

dTAd += d[m]\*Ad[m];

}

### Figure 4.2 showing the implementation of OpenMP in the solve function.

More OpenMP parallel programming was done in the main function as well as in small amounts all over the program where required but we stayed clear of the main timestep program. Variables at each time-step were dependent on their previous values thus, we were not able to apply OpenMP there.