Heat Equation

Assignment Two

Based on Example 15.3 and 21.4 from Dr. Steve Moore

Author :Sida Cheng 571224

Hashim Raza 651291

Heat Equation

Assignment Two

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.

Modeling and Analysis of the Heat Equation

$$\rho c \frac{dT}{dt} = k\nabla^2 \mathrm{T}$$

Derivation

Taking our initial equation:

$$\rho c \frac{dT}{dt} = k\nabla^2 \mathrm{T}$$

We first apply the weak form derivation:

$$\int_\Omega W(\rho c \frac{dT}{dt} - k\nabla^2 \mathrm{T}) d\Omega = 0$$

$$\int_\Omega W \rho c \frac{dT}{dt} d\Omega - \int_\Omega W k\nabla^2 \mathrm{T} d\Omega = 0$$

Next we expand the equation:

$$\int_\Omega W\rho c \frac{dT}{dt} d\Omega - (\int_\Omega k\nabla \cdot (W\nabla T) d\Omega - \int_\Omega k\nabla W \cdot \nabla T \, d\Omega) = 0$$

With the result, we then use Einstein's techniques:

$$\int_\Omega W\rho c \frac{dT}{dt} d\Omega - (\int_\Gamma kW\nabla T \cdot d\Gamma - \int_\Omega k\nabla W \cdot \nabla T \, d\Omega) = 0$$

And Re-arrange this equation to get:

$$\int_\Omega W\rho c \frac{dT}{dt} d\Omega + \int_\Omega k\nabla W \cdot \nabla T \, d\Omega = \int_\Gamma kW\nabla T \cdot d\Gamma$$

$$\int_\Gamma kW\nabla T \cdot d\Gamma = \int_\Gamma WQ_{cpu} \cdot d\Gamma - \int_\Gamma WQ_{air} \cdot d\Gamma$$

$$\int_\Gamma kW\nabla T \cdot d\Gamma = \int_\Gamma WQ_{cpu} \cdot d\Gamma - \int_\Gamma hW(T - T_{air}) \cdot d\Gamma$$

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

$$\int_{\Omega} W\rho c\frac{dT}{dt}d\Omega + \int_{\Omega} k\nabla W \cdot \nabla T\, d\Omega = \int_{\Gamma} WQ_{cpu} \cdot d\Gamma - \int_{\Gamma} hW(T - T_{air}) \cdot d\Gamma$$

And then

$$\int_{\Omega_e} \eta_p\eta_q\rho c\frac{dT_q}{dt}d\Omega + \int_{\Omega_e} k\nabla\eta_p \cdot \nabla\eta_q T_q\, d\Omega = \int_{\Gamma_e} \eta_p Q_{cpu} d\Gamma - \int_{\Gamma_e} h\eta_p(\eta_q T - T_{air})d\Gamma$$

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

$$\sum_{e=1}^{Ne}\int_{\Omega_e} \eta_p\eta_q\rho c\frac{dT_q}{dt}d\Omega + \sum_{e=1}^{Ne}\int_{\Omega_e} k\nabla\eta_p \cdot \nabla\eta_q T_q\, d\Omega = \sum_{e=1}^{Ne}\int_{\Gamma_e} \eta_p Q_{cpu} d\Gamma - \sum_{e=1}^{Ne}\int_{\Gamma_e} h\eta_p(\eta_q T - T_{air})d\Gamma$$

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:

$$M\frac{dT_q}{dt} = KT_q + s$$

which will give us:

$$M = \sum_{e=1}^{Ne}\int_{\Omega_e} \eta_p\eta_q\rho c d\Omega$$

$$K = -\sum_{e=1}^{Ne}\int_{\Omega_e} k\nabla\eta_p \cdot \nabla\eta_q\, d\Omega - \sum_{e=1}^{Ne}\int_{\Gamma_e} h\eta_p\eta_q d\Gamma$$

$$s = \sum_{e=1}^{Ne}\int_{\Gamma_e} \eta_p Q_{cpu} d\Gamma + \sum_{e=1}^{Ne}\int_{\Gamma_e} h\eta_p T_{air} d\Gamma$$

The derivations shown above are the discretization part of our partial differential equations into tetrahedron elemtns; 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:

$$\int_{\Omega_e} \eta_p^a \eta_q^b \eta_r^c \eta_s^d \, d\Omega = \frac{a!\, b!\, c!\, d!\, 6\Omega_e}{(a+b+c+d+3)!}$$

And

$$\int_{\Gamma_e} \eta_p^a \eta_q^b \eta_r^c \eta_s^d \, d\Gamma = \frac{a!\, b!\, c!\, 2\Gamma_e}{(a+b+c+2)!}$$

where Ω_e is the volume of the tetrahedron volume and Γ_e 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:

$$\int_{\Omega_e} \eta_p^2 \, \eta_q^0 = \frac{2!\, 6\Omega_e}{(2+3)!} = \frac{1}{10}\Omega_e$$

and p and q are not equal:

$$\int_{\Omega_e} \eta_p^1 \, \eta_q^1 = \frac{1!\, 1!\, 6\Omega_e}{(2+3)!} = \frac{1}{20}\Omega_e$$

So the M matrix of each element will be:

$$M = \frac{\rho c \Omega_e}{20} \begin{bmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix}$$

And Ω_e 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:

$$\int_{\Gamma_e} \eta_p^2 \eta_q^0 d\Gamma = \frac{2!\, 2\Gamma_e}{(2+2)!} = \frac{1}{6}\Gamma_e$$

when p and q are not equal:

$$\int_{\Gamma_e} \eta_p^1 \eta_q^1 d\Gamma = \frac{1!\,1!\,2\Gamma_e}{(1+1+2)!} = \frac{1}{12}\Gamma_e$$

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

$$\frac{\Gamma_e}{12}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

for the $-\sum_{e=1}^{Ne} \int_{\Omega_e} k\nabla\eta_p \cdot \nabla\eta_q \, d\Omega$ part of the K matrix. Taken from lecture notes on page 349, to calculate$\nabla\eta_p \cdot \nabla\eta_q$, the above will follow the following derivations:

$$\frac{\partial \eta_1(x,y,z)}{\partial x} = \frac{1}{6V_e}\left((y_4-y_2)(z_3-z_2)-(y_3-y_2)(z_4-z_2)\right)$$
$$\frac{\partial \eta_1(x,y,z)}{\partial y} = \frac{1}{6V_e}\left((x_3-x_2)(z_4-z_2)-(x_4-x_2)(z_3-z_2)\right)$$
$$\frac{\partial \eta_1(x,y,z)}{\partial z} = \frac{1}{6V_e}\left((x_4-x_2)(y_3-y_2)-(x_3-x_2)(y_4-y_2)\right)$$
$$\frac{\partial \eta_2(x,y,z)}{\partial x} = \frac{1}{6V_e}\left((y_3-y_1)(z_4-z_3)-(y_3-y_4)(z_1-z_3)\right)$$
$$\frac{\partial \eta_2(x,y,z)}{\partial y} = \frac{1}{6V_e}\left((x_4-x_3)(z_3-z_1)-(x_1-x_3)(z_3-z_4)\right)$$
$$\frac{\partial \eta_2(x,y,z)}{\partial z} = \frac{1}{6V_e}\left((x_3-x_1)(y_4-y_3)-(x_3-x_4)(y_1-y_3)\right)$$
$$\frac{\partial \eta_3(x,y,z)}{\partial x} = \frac{1}{6V_e}\left((y_2-y_4)(z_1-z_4)-(y_1-y_4)(z_2-z_4)\right)$$
$$\frac{\partial \eta_3(x,y,z)}{\partial y} = \frac{1}{6V_e}\left((x_1-x_4)(z_2-z_4)-(x_2-x_4)(z_1-z_4)\right)$$
$$\frac{\partial \eta_3(x,y,z)}{\partial z} = \frac{1}{6V_e}\left((x_2-x_4)(y_1-y_4)-(x_1-x_4)(y_2-y_4)\right)$$
$$\frac{\partial \eta_4(x,y,z)}{\partial x} = \frac{1}{6V_e}\left((y_1-y_3)(z_2-z_1)-(y_1-y_2)(z_3-z_1)\right)$$
$$\frac{\partial \eta_4(x,y,z)}{\partial y} = \frac{1}{6V_e}\left((x_2-x_1)(z_1-z_3)-(x_3-x_1)(z_1-z_2)\right)$$
$$\frac{\partial \eta_4(x,y,z)}{\partial z} = \frac{1}{6V_e}\left((x_1-x_3)(y_2-y_1)-(x_1-x_2)(y_3-y_1)\right)$$

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 $\sum_{e=1}^{Ne} \int_{\Omega_e} \nabla\eta_p \cdot \nabla\eta_q \, d\Omega$ 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)];
```

$$\int_{\Omega_e} \nabla\eta_p \cdot \nabla\eta_q d\Omega = (\mathrm{Gp} \cdot \mathrm{Gq})/(36\Omega)$$

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

$$\int_{\Gamma_e} \eta_p d\Gamma = \frac{1!\,2\Gamma_e}{(1+2)!} = \frac{1}{3}\Gamma_e$$

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 into tetrahedron elements, 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:

$$M\frac{dT}{dt} = KT + s$$

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 $\sum_{e=1}^{Ne}\int_{\Omega_e} k\nabla\eta_p \cdot \nabla\eta_q \, d\Omega$ 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, $\sum_{e=1}^{Ne}\int_{\Gamma_e} h\eta_p\eta_q d\Gamma$ 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 $\sum_{e=1}^{Ne}\int_{\Gamma_e} h\eta_p\eta_q d\Gamma$ 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

$$\phi^{l+1} = \phi^{l} + \Delta t f(\phi^{l+1}, t^{l+1})$$

which can be written in the form:

$$AT^{l+1} = b$$

And

$$A = M - \Delta t K$$

$$b = MT^{l} + \Delta t \cdot s$$

In the time marching loop T could be caudated from T values from previous time step by calculating T=b/A with initial temperate assumption to be 300K (T(:)=300). The MATLAB analysis is calculated on the BOX grid and the final result are plotted in figure 1,2 and 3.

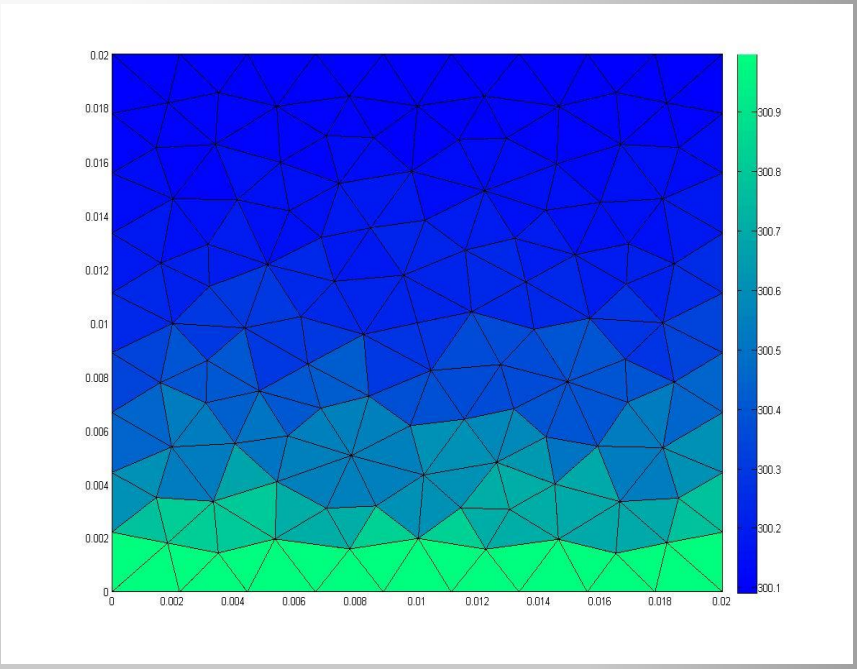

FIGURE 1 NEUMANN BOUNDARY PLOT AT T~0S

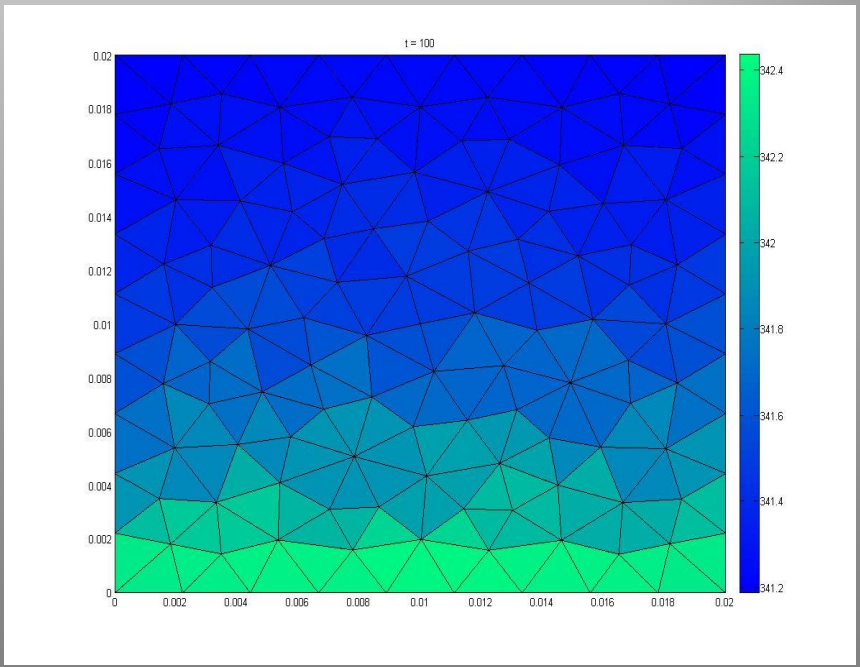

FIGURE 2 NEUMANN BOUNDARY PLOT AT T=100S

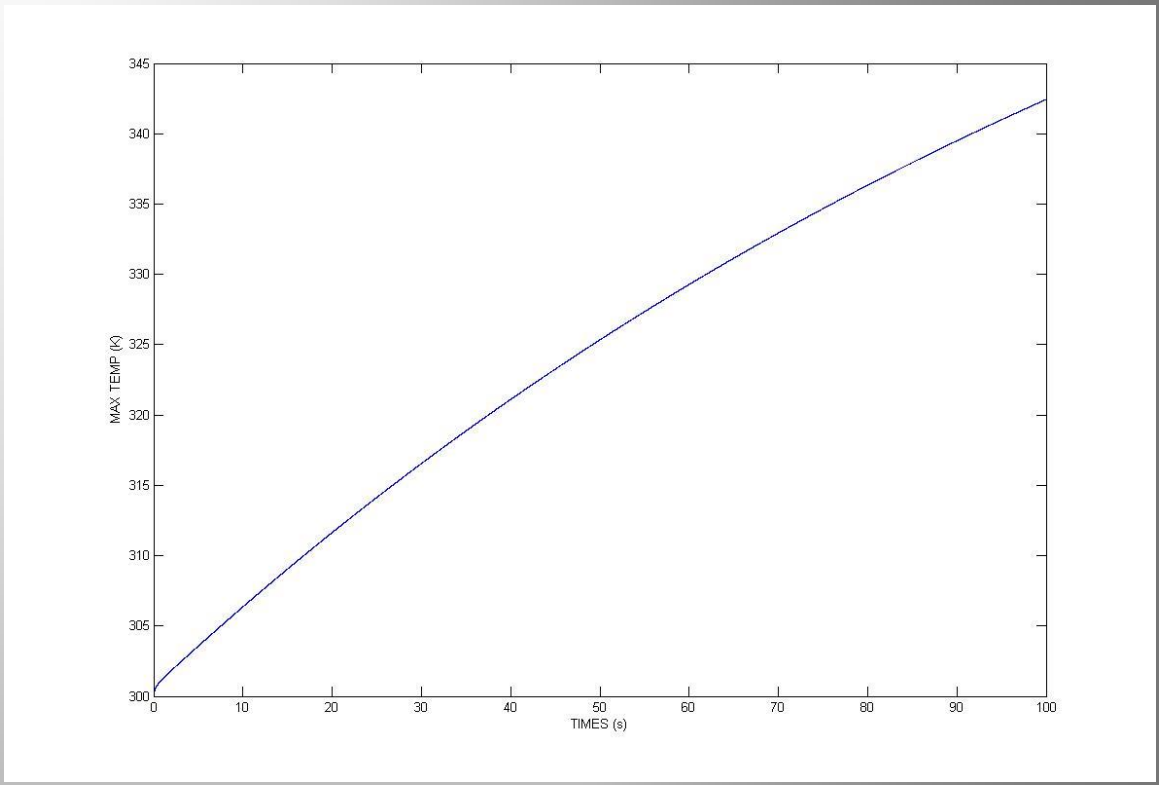

FIGURE 3 MAXIMUM TEMPERATURE VS. TIME

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 degrees as 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;
   }
```

Code 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;

    }
```

Code 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]);
        }
}
```

CODE 3 shows how matrices 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);
```

CODE 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.

Code Parallelization 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 is an API that accomplishes this by using shared memory architecture. Multiple process threads can access the same memory pool of data and process code to quickly come to an output. MPI on the other hand is an API that uses distributed memory architecture, designed specifically for distributed computer environments. Basically data is stored over a variety of memory locations and was accessed by individual threads through a series of calls (sends and receives).

In this section of the assignment we had to take our serial code and implement an OpenMP MPI hybrid 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 which related to Derichelet boundary conditions for points with values we knew unlike our current unstructured grid where heat flux values at each point made them unknowns. So we removed them from both the functions and then tweaked them a bit. 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];
				}
			}
		}
```

CODE 5 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];

}
```

CODE 6 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 section of the code.

Our VTK files successfully visualized on Paraview and showed how temperature gradients varied over a square surface. The temperature levels reached a threshold of around the 342 degree mark, which was well within the range of accuracy of our Matlab code's analysis.

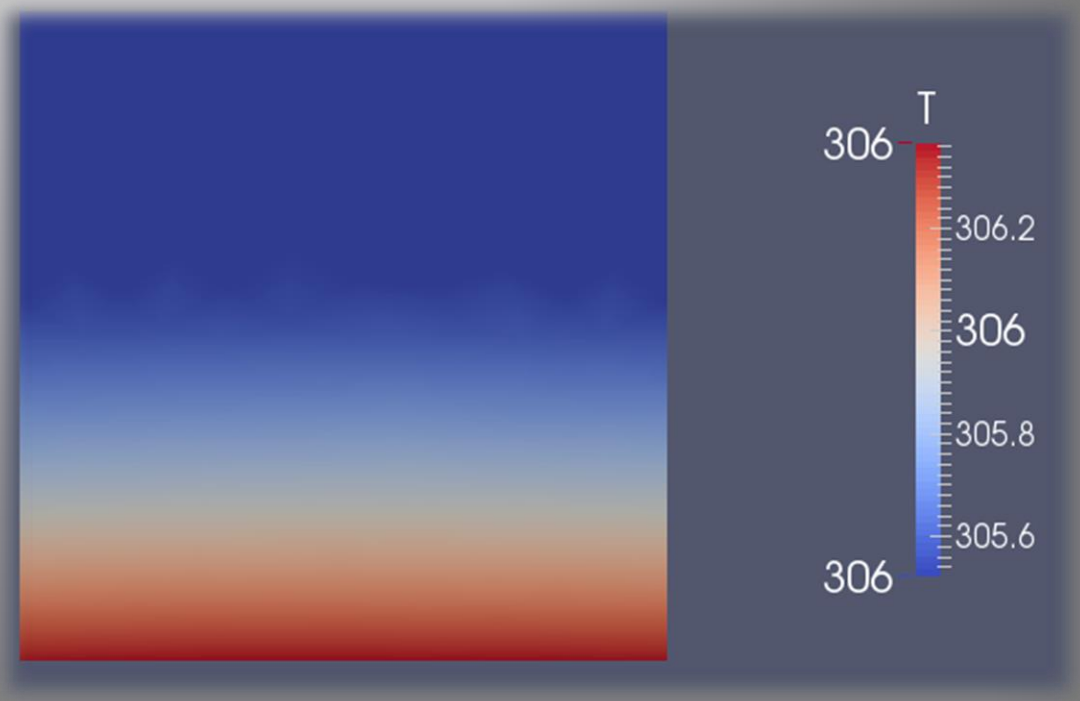

FIGURE 4 SHOWS THE FACE OF OUR BOX GRID AND THE TEMPERATURE FLUX FROM BOTTOM TO TOP.

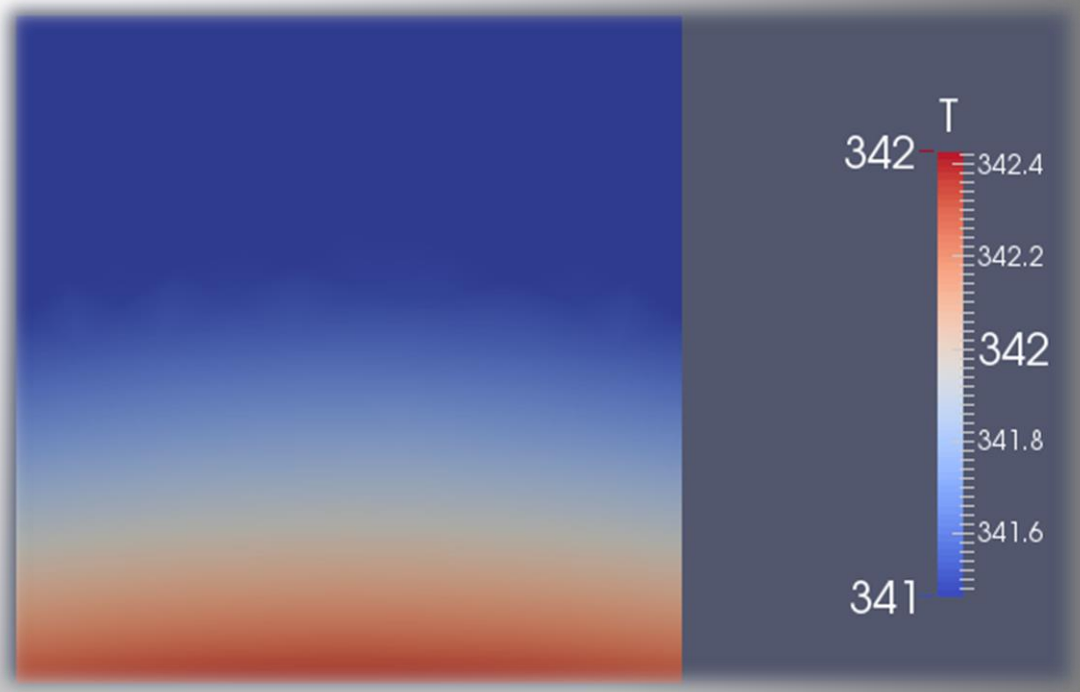

FIGURE 5 SHOWS THE MAXIMUM TEMPERATURE GRADIENT REACHED BY OUR BOX GRID AT 342.4 DEGREES. THIS IS IN LINE WITH OUR MATLAB CODE'S CALCULATIONS AND VTK'S FINAL TEMPERATURE OUTPUT.

Using the CPUHeatSink grid output in Paraview, we were able to visualize the temperature gradients over the heat sink the same way we saw with the Box grid. It was interesting to note that the maximum temperature reached in this system was less then what we calculated in the Box grid or Matlab, by a whole 20 degrees! That's probably because of the shape difference of the CPU heat sink and the fins allowing for more dynamic cooling in deeper regions all over the heat sink. The increased surface area due to the number of fins present, with each fin being a face (system of points) its obvious why this shape can dissipate heat more effectively.

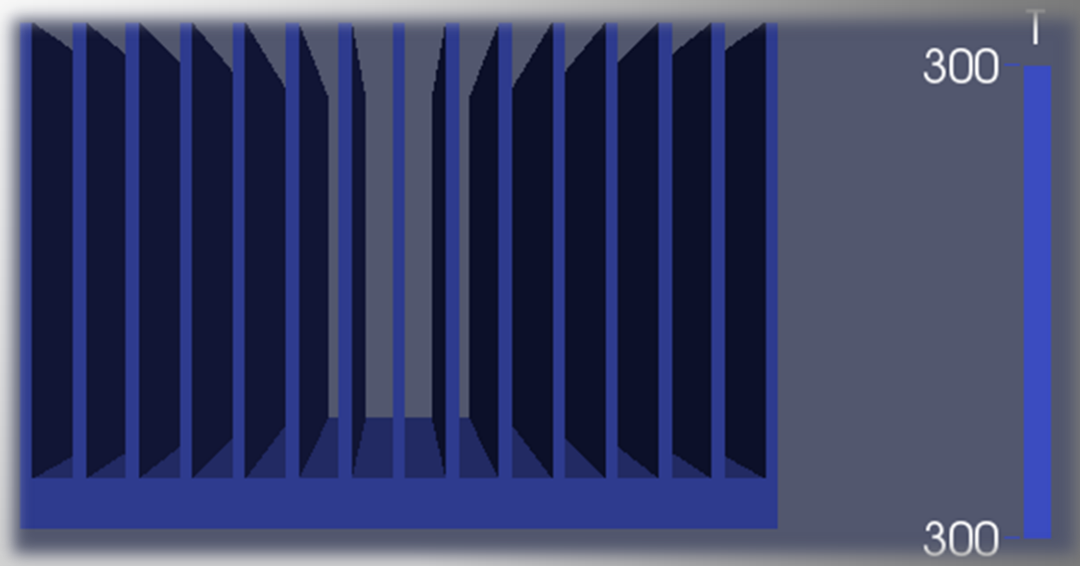

FIGURE 6 SHOWING PARAVIEW ON THE CPUHEATSINK GRID AT ITS INITIAL POINT.

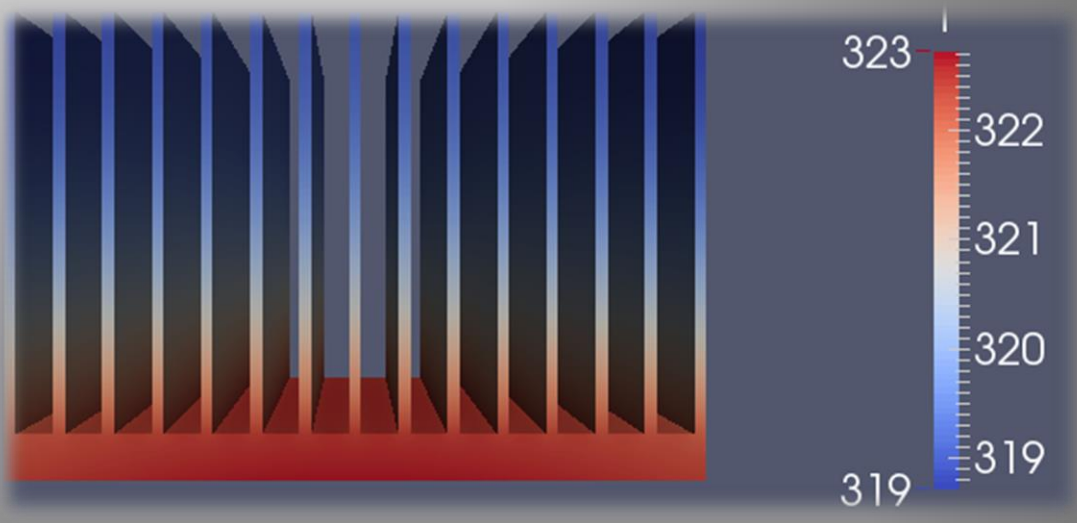

FIGURE 7 SHOWS THE MAXIMUM THRESHOLD TEMPERATURE REACHED BY PARAVIEW AT 323 DEGREES, A WHOLE 20 LESS THAN OUR BOX GRID/MATLAB ESTIMATE. INCREASED SURFACE AREA = BETTER COOLING.

Scaling run analysis

Scaling runs for our serial code were in the ranges of 250<x<300 seconds depending on the target computers machine performance which in my case was an Alienware equipped with Sandy bridge I7 processor and dual GTX50 GPUs in SLI. Basically, I wasn't really impressed…

However, the moment we introduced MPI over the whole program and OpenMP (mostly around the Solve function) we were able to see interesting results.

Box grid - with 8 processes, 8 tasks per node and 1 thread per process - 76.85 seconds

Box grid - with 8 processes, 4 tasks per node and 1 thread per process - 51.48 seconds

Box grid - with 8 processes, 2 tasks per node and 1 thread per process - 38.53 seconds

Box grid - with 8 processes, 1 task per node and 1 thread per process - 30.85 seconds

Which showed that the lesser the tasks each node has, the more efficiently the system was able to run. That's because each node handles tasks in a serial fashion, much like our CPUs today. So by overloading each node doubles the time it takes to complete the same task.

We also analyzed the biggest area of issue or lag was during IOs. This was especially true during writing data to file which was obvious.