Finite Element Method for The Heat equation

Andres Chaves(706801), Diego Montufar(661608)

October 27, 2014

Abstract

Throughout this document we are going to make use of the Finite Element Method to solve the Heat Equation, that models heat flux distribution, applied to a CPU Sink. Given the big amount of computation required to solve the numerical system constructed by using this method, it's a perfect study case to apply High Performance computing tools like OpenMP and MPI. In this particular problem, we have implemented a Matlab program and a C++ version. The domain was decomposed using tetrahedral linear elements and the time discretization was done by using the implicit Euler method. For visualization tools we used the ones provided by Matlab and Paraview. Our first approach shows how a constant heat flux generated by a CPU dissipates through a Box grid and then we extend that into an unstructured grid of the CPU Sink. We tested the speedup of the implemented programs with a weak scaling methodology.

1 Introduction

The Finite Element method is a powerful numerical technique to solve a variety of 'real-world' engineering problems having complex domains subjected to general boundary conditions. Those domains are decomposed into simpler shaped regions, or "elements". An approximate solution for the PDE can be developed for each of these elements. The total solution is then generated by linking together, or "assembling", the individual solutions taking care to ensure continuity at the interelement boundaries. Thus, the PDE is satisfied in a piecewise fashion.

Although the particulars will vary, the implementation of the finite-element approach usually follows a standard step-by-step procedure:

1. Discretization of the domain into finite elements.
2. Selection of shape functions
3. Derive element matrices for the system
4. Assembly the element matrices into a global matrix.
5. Impose boundary conditions
6. Compute the problem solution.

In order to solve the Heat Equation for this particular problem we are going to derive all the equations by following the mentioned procedure and then we are going to compute and visualize the solution through the implementation of a Matlab program. Our first approach involves the simulation of a heat flux constantly applied to the bottom of a 3D Box which represents what we'll later convert into an ustructured grid of a CPU heat sink. The discretization of the domain will be done by using tetrahedral linear elements.

Then we are going to translate our Matlab code into a C++ sequential version which involves the implementation of linear algebra operations and as we are going to store big amount of data for the global system of equations (which are represented by sparse matrices), we are going to use some implementations of code that allow us to use memory efficiently by using a "Compress Row Storage" technique. The results then will be visualized with Paraview throughout Vtk files generated by the C++ program.

Once we have our sequential version of the program, the next step will be to parallelize it with an Hybrid version of the code, implementing OpenMP for multithreading and MPI for multiprocessor communication. This version will be run on the BlueGeneQ (Avoca) and there will be a weak scaling test in order to show the speedup results of the parallelization process.

Finally, we must mention that we have used as reference the code implementations from Example 15.3 and Example 21.4 taken from the Lecture's Book[1] and other resources such as unstructured grid files as well. The learning outcomes of this Assignment are the ability to apply The Finite element method to the 3D Heat equation and compute its solution using High performance computing techniques with OpenMP and MPI for this particular problem.

2 Problem Statement

The heat equation can be used to describe the temperature variation through a solid body. The equation derives from the principle of conservation of energy and gives rise to a second order linear PDE of the form:

$$\rho C \frac{\partial T}{\partial t} = k \nabla^2 T \tag{1}$$

Where T is the temperature $[K]$,ρ is the mass density $[kgm^3]$, C is the specific heat capacity $[Jkg^{-1}K^{-1}]$, and k is the thermal conductivity $[Wm^{-1}K^{-1}]$. For this particular implementation the computational domain will describe cooling fins on top of a central processing unit and will be defined by an unstructured tetrahedral grid. We assumed that the cooling fins are made from pure copper, having the properties $\rho = 8954kgm^3$, $C = 380Jkg^{-1}K^{-1}$, and $k = 386Wm^{-1}K^{-1}$.

In this particular case the CPU creates a constant heat flux Q_{cpu} , given by:

$$Q_{cpu} = k \nabla T \tag{2}$$

which defines a Neumann boundary condition on the base of the domain and $Q_{cpu} = 40,000Wm^{-2}$. For the remaining boundary of the cooling fins we assumed a convection boundary condition, given by:

$$Q_{air} = k\nabla T = h(T - T_{air}) \tag{3}$$

which defines a Robin boundary condition. Here we assumed that the convection coefficient $h = 100W/m^2K^{-1}$ and the ambient temperature $T_{air} = 300K$. Defining the initial temperature of the cooling fins as ambient, the main goal is to simulate the temperature rise in the domain $t \in [0, 100]s$ and determine the maximum steady state temperature of the cooling fins.

3 Governing Equations

3.1 Discretization of the domain into finite elements

Weighted residual methods assume that a solution can be approximated analytically or piecewise analytically. The idea of this method is that we can multiply the residual by a weighting function and force the integral of the weighted expression over the domain to vanish:

$$\int_{\Omega} W(x)r(x)d\Omega = 0 \tag{4}$$

We understand the residual as a continuous function of space. In this case we are going to use the Galerking method of weighted residuals. In this method the weight functions are chosen to be identical to the trial functions where the trial function is our assumed approximate solution for the PDE which takes the form:

$$\phi(x) = \sum_{n=0}^{N} a_n(x)p_n(x), \quad \text{Where} \quad W_n(x) = p_n(x) \tag{5}$$

Applying equation (4) to the Heat Equation (1) we have:

$$r = \rho C\frac{\partial T}{\partial t} - k\nabla^2 T$$

$$\int_{\Omega} W(\rho C\frac{\partial T}{\partial t} - k\nabla^2 T)d\Omega = 0$$

This is an appropriate time to include the convection term into our equation and expanding terms we get:

$$\int_{\Omega} W\rho C\frac{\partial T}{\partial t}d\Omega - \int_{\Omega} Wk\nabla^2 T d\Omega + \int_{\Omega} WQ_{air}d\Omega = 0 \tag{6}$$

We can modify the diffusive second order term by applying the product rule for derivatives which is defined by:

$$\begin{aligned}\nabla \cdot (W\nabla T) &= \nabla W \cdot \nabla T + W\nabla \cdot \nabla T \\ &= \nabla W \cdot \nabla T + W\nabla^2 T\end{aligned}$$

In our second term of Equation (6): Applying the product rule, rearranging, and integrating over the domain we get:

$$\int_{\Omega} Wk\nabla^2 T d\Omega = \int_{\Omega} \nabla \cdot (Wk\nabla T) d\Omega - \int_{\Omega} \nabla W \cdot k\nabla T d\Omega \tag{7}$$

in order to eliminate second order terms of the Equation (7), another important concept we are going to use is the theorem of divergence which can be expressed as follows:

$$\int_{\Omega} \nabla^2 T d\Omega = \int_{\Gamma} \nabla T \cdot d\Gamma$$

Now, we can make use of the Divergence Theorem and apply it to the second order term of Equation (7) as follows:

$$\int_{\Omega} kW\nabla^2 T d\Omega = \int_{\Gamma} kW\nabla T \cdot d\mathbf{\Gamma} - \int_{\Omega} k\nabla W \cdot \nabla T d\Omega \tag{8}$$

Substitute Equation (8) in (6) we get:

$$\int_{\Omega} \rho CW \frac{\partial T}{\partial t} d\Omega - \int_{\Gamma_{Base}} kW\nabla T \cdot d\mathbf{\Gamma} - \int_{\Omega} k\nabla W \cdot \nabla T d\Omega + \int_{\Gamma_{Fins}} WQ_{air} d\Gamma = 0 \tag{9}$$

Then we can see the second term in the equation corresponds to the value of Q_{cpu} as stated in the model problem. In contrast to other methods, the Neumann boundary conditions are automatically incorporated into the integral form of the PDE. Rearranging terms we get the Weak form of the Heat Equation:

$$\int_{\Omega} \rho CW \frac{\partial T}{\partial t} d\Omega + \int_{\Omega} k\nabla W \cdot \nabla T d\Omega = \int_{\Gamma_{Base}} kW\nabla T \cdot d\mathbf{\Gamma} - \int_{\Gamma_{Fins}} Wh(T - T_{air}) d\Gamma \tag{10}$$

3.2 Shape functions

One of the fundamental steps in a finite element analysis is the discretization of a continuous body containing infinite number of points in the surface into a discrete model with a limited number of points (or nodes) in the surface. The shape of the body between these nodes its approximated by functions. These functions are known as shape functions, and allow us to relate the coordinates of every point of a finite element with the positions of its nodes.
Thinking of T most generally as being a continuous function of space and time, we will write the assumed solution in the form:

$$T(\mathbf{x}, t) = \sum_{n=1}^{N_n} \eta_n(\mathbf{x}) T_n(t) \tag{11}$$

Where η_n are known as the *shape functions*, which are functions of spatial location only, while the nodal values of T are functions of time only.
If we now substitute in our assumed form of the solution from Equation (10), and consider the domain of integration to be the domain of an element itself, the weighted residual expression can be rewritten using Einstein summation notation as:

$$\int_{\Omega_e} \eta_p \eta_q \rho C \frac{\partial T}{\partial t} d\Omega + \int_{\Omega_e} k\nabla \eta_p \cdot \nabla \eta_q T_q d\Omega = \int_{\Gamma_e} k\eta_p \nabla T \cdot d\mathbf{\Gamma} - \int_{\Gamma_e} h\eta_p(\eta_q T_q - T_e) d\Gamma \tag{12}$$

Then the overall System of Equations to solve the Heat Equation is the given by:

$$\sum_{e=1}^{N_e} \int_{\Omega_e} \left(\eta_p \eta_q \rho C \frac{\partial T_q}{\partial t} d\Omega + k \nabla \eta_p \cdot \nabla \eta_q T_q \right) d\Omega = \sum_{e=1}^{N_e} \int_{\Gamma_e} k \eta_p \nabla T \cdot d\mathbf{\Gamma} - \sum_{e=1}^{N_e} \int_{\Gamma_e} \eta_p h (\eta_q T_q - T_{air}) d\Gamma \quad (13)$$

We can then rewrite the system of equations in the form:

$$M\dot{T} = KT + s \quad (14)$$

Where M represents the Mass matrix, K the Stiffness Matrix, both have size of N_pxN_p and finally the s term that is the load vector with size N_px1. For the Heat Equation we can recognize each term of Equation(14) in the Equation (12) as follows:

$$M_{pq}^e = \int_{\Omega_e} \rho C \eta_p \eta_q d\Omega \quad (15)$$

$$K_{pq}^e = -k \int_{\Omega_e} \nabla \eta_p \cdot \nabla \eta_q d\Omega - h \int_{\Gamma_e} \eta_p \eta_q d\Gamma \quad (16)$$

$$S_p^e = h T_{air} \int_{\Gamma_e} \eta_p d\Gamma + k \int_{\Gamma_e} \eta_p \nabla T \cdot d\mathbf{\Gamma} \quad (17)$$

For this case we are going to use tetrahedral elements, which means we are going to have 4 nodes in a 3D element as shown in Figure 1. In order to derive the corresponding shape functions we begin by assuming a trial solution which takes the form:

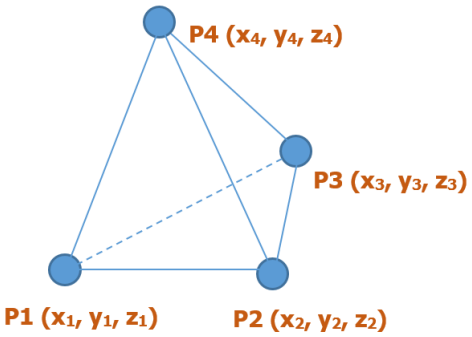

Figure 1: *3D Linear tetrahedron element*

$$T(x, y, z) = a_0 + a_1 x + a_2 y + a_3 z \quad (18)$$

Where a_0,a_1,a_2 and a_3 are scalar coefficients. This is essentially the 3D equivalent of the trial solution to a triangular element. If we apply this trial solution in every node of the element we'll get:

$$\begin{Bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{Bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{Bmatrix} = \begin{bmatrix} \mathbf{p_1} \\ \mathbf{p_2} \\ \mathbf{p_3} \\ \mathbf{p_4} \end{bmatrix} \mathbf{a} = C\mathbf{a} \quad (19)$$

Where $p_n = \{1\, x_n\, y_n\, z_n\}$, C will be a square 4x4 matrix and we can solve for the unknown parameters by computing $\mathbf{a} = C^{-1}\mathbf{T}$. Doing so we can find our shape functions by substituting these coefficients back into our trial solution, So we can factor out the nodal values and rewrite the solution in the form:

$$T(x,y,z) = \sum_{n=1}^{N_n} \eta_n(x,y,z)T_n = \eta_1 T_1 + \eta_2 T_2 + \eta_3 T_3 + \eta_4 T_4 \tag{20}$$

Where the shape functions for the linear tetrahedron are:

$$\begin{aligned}\eta_1(x,y,z) =& \frac{1}{6V_e}(x_2(y_3z_4 - y_4z_3) + x_3(y_4z_2 - y_2z_4) + x4(y_2z_3 - y_3z_2) \\ &+ ((y_4 - y_2)(z_3 - z_2) - (y_3 - y_2)(z_4 - z_2))x \\ &+ ((x_3 - x_2)(z_4 - z_2) - (x_4 - x_2)(z_3 - z_2))y \\ &+ ((x_4 - x_2)(y_3 - y_2) - (x_3 - x_2)(y_4 - y_2))z)\end{aligned}$$

$$\begin{aligned}\eta_2(x,y,z) =& \frac{1}{6V_e}(x_1(y_4z_3 - y_3z_4) + x_3(y_1z_4 - y_4z_1) + x4(y_3z_1 - y_1z_3) \\ &+ ((y_3 - y_1)(z_4 - z_3) - (y_3 - y_4)(z_1 - z_3))x \\ &+ ((x_4 - x_3)(z_3 - z_1) - (x_1 - x_3)(z_3 - z_4))y \\ &+ ((x_3 - x_1)(y_4 - y_3) - (x_3 - x_4)(y_1 - y_3))z)\end{aligned}$$

$$\begin{aligned}\eta_3(x,y,z) =& \frac{1}{6V_e}(x_1(y_2z_4 - y_4z_2) + x_2(y_4z_1 - y_1z_4) + x4(y_1z_2 - y_2z_1) \\ &+ ((y_2 - y_4)(z_1 - z_4) - (y_1 - y_4)(z_2 - z_4))x \\ &+ ((x_1 - x_4)(z_2 - z_4) - (x_2 - x_4)(z_1 - z_4))y \\ &+ ((x_1 - x_4)(y_1 - y_4) - (x_1 - x_4)(y_2 - y_4))z)\end{aligned}$$

$$\begin{aligned}\eta_4(x,y,z) =& \frac{1}{6V_e}(x_1(y_3z_2 - y_2z_3) + x_2(y_1z_3 - y_3z_1) + x3(y_2z_1 - y_1z_2) \\ &+ ((y_1 - y_3)(z_2 - z_1) - (y_1 - y_2)(z_3 - z_1))x \\ &+ ((x_2 - x_1)(z_1 - z_3) - (x_3 - x_1)(z_1 - z_2))y \\ &+ ((x_1 - x_3)(y_2 - y_1) - (x_1 - x_2)(y_3 - y_1))z)\end{aligned}$$

And V_e is the volume of the element and is defined in terms of its nodal coordinates as:

$$6V_e = \begin{bmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{bmatrix} \tag{21}$$

Deriving the the linear tetrahedron shape functions we get:

$$\frac{\partial \eta_1(x,y,z)}{\partial x} = \frac{1}{6V_e}((y_4 - y_2)(z_3 - z_2) - (y_3 - y_2)(z_4 - z_2))$$
$$\frac{\partial \eta_1(x,y,z)}{\partial x} = \frac{1}{6V_e}((x_3 - x_2)(z_4 - z_2) - (x_4 - x_2)(z_3 - z_2))$$
$$\frac{\partial \eta_1(x,y,z)}{\partial x} = \frac{1}{6V_e}((x_4 - x_2)(y_3 - y_2) - (x_3 - x_2)(y_4 - y_2))$$

$$\frac{\partial \eta_2(x,y,z)}{\partial x} = \frac{1}{6V_e}((y_3 - y_1)(z_4 - z_3) - (y_3 - y_4)(z_1 - z_3))$$
$$\frac{\partial \eta_2(x,y,z)}{\partial x} = \frac{1}{6V_e}((x_4 - x_3)(z_3 - z_1) - (x_1 - x_3)(z_3 - z_4))$$
$$\frac{\partial \eta_2(x,y,z)}{\partial x} = \frac{1}{6V_e}((x_3 - x_1)(y_4 - y_3) - (x_3 - x_4)(y_1 - y_3))$$

$$\frac{\partial \eta_3(x,y,z)}{\partial x} = \frac{1}{6V_e}((y_2 - y_4)(z_1 - z_4) - (y_1 - y_4)(z_2 - z_4))$$
$$\frac{\partial \eta_3(x,y,z)}{\partial x} = \frac{1}{6V_e}((x_1 - x_4)(z_2 - z_4) - (x_2 - x_4)(z_1 - z_4))$$
$$\frac{\partial \eta_3(x,y,z)}{\partial x} = \frac{1}{6V_e}((x_2 - x_4)(y_1 - y_4) - (x_1 - x_4)(y_2 - y_4))$$

$$\frac{\partial \eta_4(x,y,z)}{\partial x} = \frac{1}{6V_e}((y_2 - y_3)(z_2 - z_1) - (y_1 - y_2)(z_3 - z_1))$$
$$\frac{\partial \eta_4(x,y,z)}{\partial x} = \frac{1}{6V_e}((x_2 - x_1)(z_1 - z_3) - (x_3 - x_1)(z_1 - z_2))$$
$$\frac{\partial \eta_4(x,y,z)}{\partial x} = \frac{1}{6V_e}((x_1 - x_3)(y_2 - y_1) - (x_1 - x_2)(y_3 - y_1))$$

Furthermore, we have the integration formula defined for the linear tetrahedron:

$$\int_{\Omega_e} \eta_p^a \eta_q^b \eta_r^c \eta_s^d d\Omega = \frac{a!b!c!d!6\Omega_e}{(a+b+c+d+3)!} \tag{22}$$

and:

$$\int_{\Gamma_e} \eta_p^a \eta_q^b \eta_r^c d\Omega = \frac{a!b!c!2\Gamma_e}{(a+b+c+2)!} \tag{23}$$

Where for the 3D case Ω_e and Γ_e represent the volume and face area of a tetrahedron respectively.

3.3 Derivation of element matrices for the System

Let's first consider the integration of the shape functions as required in the mass matrix from Equation (15):

$$M^e_{p,q} = \rho C \int_{\Omega_e} \eta_p \eta_q d\Omega$$

Remembering that p and q are in the range of 1 to 4 for the linear tetrahedral element, what we end up with is a local or 'sub' matrix M_e, which for our linear tetrahedral element will be a 4 x 4 matrix. In order to evaluate each term in the matrix we simply input the values of p and q to the integration formula in Equation (22). For the case where p and q are equal (i.e. for elements on the main diagonal) we get:

$$\begin{aligned} M^e_{p,p} &= \rho C \int_{\Omega_e} \eta_p \eta_p d\Omega \\ &= \rho C \int_{\Omega_e} \eta_p^2 \eta_q^0 \eta_r^0 \eta_s^0 d\Omega = \frac{\rho C 2!0!0!0!6\Omega_e}{(2+0+0+0+3)!} \\ &= \frac{2\rho C \Omega_e}{20} \end{aligned}$$

For the case where p and q are not equal (i.e. for elements off the main diagonal) we get:

$$\begin{aligned} M^e_{p,q} &= \rho C \int_{\Omega_e} \eta_p \eta_q d\Omega \\ &= \rho C \int_{\Omega_e} \eta_p^1 \eta_q^1 \eta_r^0 \eta_s^0 d\Omega = \frac{\rho C 1!1!0!0!6\Omega_e}{(1+1+0+0+3)!} \\ &= \frac{\rho C \Omega_e}{20} \end{aligned}$$

So we can write a single expression for the any element in our local mass matrix as:

$$M^e_{p,q} = \frac{\rho C (1 + \delta_{pq})}{20}$$

which can be finally written as:

$$M^e = \frac{\rho C \Omega_e}{20} \begin{bmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \tag{24}$$

Considering now the contribution of the diffusive term to the stiffness matrix, and writing shape function derivative terms in the form of a column vector with modulo 3 notation for compactness we have:

$$-k\int_{\Omega_e}\nabla\eta_p\cdot\nabla\eta_q d\Omega =$$

$$= -k\int_{\Omega_e}\frac{1}{6\Omega_e}\begin{Bmatrix}(y_{p+3}-y_{p+1})(z_{p+2}-z_{p+1})-(y_{p+2}-y_{p+1})(z_{p+3}-z_{p+1})\\(x_{p+2}-x_{p+1})(z_{p+3}-z_{p+1})-(x_{p+3}-x_{p+1})(z_{p+2}-z_{p+1})\\(x_{p+3}-x_{p+1})(y_{p+2}-y_{p+1})-(x_{p+2}-x_{p+1})(y_{p+3}-y_{p+1})\\\cdots\end{Bmatrix}$$

$$.\frac{1}{6\Omega_e}\begin{Bmatrix}(y_{q+3}-y_{q+1})(z_{q+2}-z_{q+1})-(y_{q+2}-y_{q+1})(z_{q+3}-z_{q+1})\\(x_{q+2}-x_{q+1})(z_{q+3}-z_{q+1})-(x_{q+3}-x_{q+1})(z_{q+2}-z_{q+1})\\(x_{q+3}-x_{q+1})(y_{q+2}-y_{q+1})-(x_{q+2}-x_{q+1})(y_{q+3}-y_{q+1})\\\cdots\end{Bmatrix}$$

Performing the Integration we get:

$$-k\int_{\Omega_e}\nabla\eta_p\cdot\nabla\eta_q d\Omega =$$

$$-\frac{k}{36\Omega_e^2}\begin{Bmatrix}(y_{p+3}-y_{p+1})(z_{p+2}-z_{p+1})-(y_{p+2}-y_{p+1})(z_{p+3}-z_{p+1})\\(x_{p+2}-x_{p+1})(z_{p+3}-z_{p+1})-(x_{p+3}-x_{p+1})(z_{p+2}-z_{p+1})\\(x_{p+3}-x_{p+1})(y_{p+2}-y_{p+1})-(x_{p+2}-x_{p+1})(y_{p+3}-y_{p+1})\\\cdots\end{Bmatrix} \quad (25)$$

$$.\begin{Bmatrix}(y_{q+3}-y_{q+1})(z_{q+2}-z_{q+1})-(y_{q+2}-y_{q+1})(z_{q+3}-z_{q+1})\\(x_{q+2}-x_{q+1})(z_{q+3}-z_{q+1})-(x_{q+3}-x_{q+1})(z_{q+2}-z_{q+1})\\(x_{q+3}-x_{q+1})(y_{q+2}-y_{q+1})-(x_{q+2}-x_{q+1})(y_{q+3}-y_{q+1})\\\cdots\end{Bmatrix}\Omega_e$$

The important point to note is that the dot product between these two vectors produce a scalar value. Now we obtain the second part of the contribution to the stiffness matrix again first evaluating the integral for the main diagonal where p=q:

$$-h\int_{\Gamma_e}\eta_p\eta_p d\Gamma = -h\int_{\Gamma_e}\eta_p^2\eta_q^0\eta_r^0 d\Gamma$$

$$= -\frac{h2!0!0!2\Gamma_e}{(2+0+0+2)!}$$

$$= -\frac{2h\Gamma_e}{12}$$

Then we do the same for the off diagonal terms where p $\neq$ q:

$$-h\int_{\Gamma_e}\eta_p\eta_q d\Gamma = -h\int_{\Gamma_e}\eta_p^1\eta_q^1\eta_r^0 d\Gamma$$

$$= -\frac{h1!1!0!2\Gamma_e}{(1+1+0+2)!}$$

$$= -\frac{h\Gamma_e}{12}$$

So we can write a single expression for the any element in the contribution term for our local stiffness matrix as:

$$K^e_{p,q} = -\frac{h(1+\delta_{pq})}{12}$$

which is simple enough that we could write out the whole thing as:

$$K^e = -\frac{h\Gamma_e}{12}\begin{bmatrix}2 & 1 & 1\\ 1 & 2 & 1\\ 1 & 1 & 2\end{bmatrix} \tag{26}$$

Considering the contribution of the Neumann boundary condition term to the load vector, we have to perform the integral:

$$\mathbf{s}^f_p = k\int_{\Gamma_e} \eta_p \nabla T \cdot d\mathbf{\Gamma}$$

An important point to note is that we will only perform this integral over the boundary of the element (which in our case translates into a face of a triangle) if that element happens to be on a Neumann boundary of our computational domain. In this case p will be in the range 1 to 3. Because our Neumann boundary conditions are applied normal to the boundary, the gradient can be specified as a scalar (the implied direction being normal to the boundary face) and so the dot product will drop out of the integral. We will furthermore assume that T does not vary over the boundary of the element. Making use of the second integration formula (23) we then have:

$$\begin{aligned}\mathbf{s}^f_p &= k\int_{\Gamma_e} \eta_p \nabla T \cdot d\mathbf{\Gamma}\\ &= k\nabla T\int_{\Gamma_e} \eta_p \cdot d\mathbf{\Gamma}\\ &= k\nabla T\frac{h1!0!0!2\Gamma_e}{(1+0+0+2)!}\\ &= \frac{k\nabla T\Gamma_e}{3}\end{aligned}$$

So we can write the contribution to our load vector as:

$$\mathbf{s}^f_p = \frac{k\nabla T\Gamma_e}{3}\begin{bmatrix}1\\1\\1\\1\end{bmatrix} \tag{27}$$

And finally we obtain the first part of our load vector as follows:

$$
\begin{aligned}
\mathbf{s}_p^e &= hT_{air} \int_{\Gamma_e} \eta_p \cdot d\mathbf{\Gamma} \\
&= hT_{air} \int_{\Gamma_e} \eta_p \cdot d\mathbf{\Gamma} \\
&= hT_{air} \frac{h1!0!0!2\Gamma_e}{(1+0+0+2)!} \\
&= \frac{hT_{air}\Gamma_e}{3}
\end{aligned}
$$

So we can write the contribution to our load vector as:

$$
\mathbf{s}_p^e = \frac{hT_{air}\Gamma_e}{3} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{28}
$$

3.4 Assemble

There is however the trick question on how to compute all the integrals that appear in the matrix and right hand side of the system. This is done by a process called assembly of the system, another 'ingredient' of the many good deeds of the Finite element method [3]. When we think about how we assemble M , K, and s with the Finite Element method, we loop over all of the elements and for each element loop over all of the nodes that a given element uses, adding in terms to M , K, and s in the rows and columns corresponding to those nodes.

3.5 Apply Boundary conditions

The final step after assembling our global system is to compute the Robin and Neumann boundary conditions which in fact, have contribution terms to the stiffness matrix and load vector as well. Finally, we can proceed with the solution of the resulting system equations using some ODE numerical method lets say with an implicit Euler method for temporal discretization and a conjugate gradient method for solving matrices that could be easily done in Matlab with the backslash operator.

4 Problem Solution and Implementation

In order to compute the solution of the problem, firstly we implemented a Matlab program solving a Box grid as our first approach to test the results. Then we translated it into a C++ sequential version. At this point the output results of an unstructured Box grid will be shown in Paraview as well as a more complex figure representing a CPU heat sink. After getting the desired results we are going to parallelize the sequential version of the C++ code by using MPI and OpenMP.

4.1 Matlab and C++ implementation

As we are going to reuse the code provided in the Lectures and tutorials, we decided to describe only the part of the code that was modified/added and some relevant comments related to it. For this part of the implementation we used the *Example21-4.m* for the Matlab code and *Example21-4.cpp* for the C++ sequential version. For the spatial discretization we will use the Finite Element method with linear tetrahedral elements, for the temporal discretization we will use the implicit Euler method, and we will solve the resulting linear system with the Conjugate Gradient method. The first step is to set up all the constant terms from the Heat equation. This could be done by adding the next lines:

```
rho             = 8954.00;
C               = 380.00;
k               = 386.00;
h               = 100.00;
Tair            = 300.00;
Qcpu            = 40000.00;
```

In c++ we could do the similar:

```
const double rho            = 8954.00;
const double C              = 380.00;
const double k              = 386.00;
const double h              = 100.00;
const double Tair           = 300.00;
const double Qcpu           = 40000.00;
```

As mentioned in the problem statement, we assumed the initial temperature of the cooling fins as ambient (300K). Thus we set the initial condition as follows:

```
% Set initial condition
phi(:,1)        = 300;
```

And in the C++ version:

```
for(int m=0; m<N_p; m++){
    phi[m]  = 300;
}
```

In Matlab the read function is defined to return the value and number of the Points, Faces, Elements and Boundaries defined in the grid file. This file contains pretty exactly the same for both Matlab and C++ programs, the only difference is that Matlab will 'automatically' set the parameter values in structs and vector variables and the C++ program has to go through the file line by line, so that it can dynamically allocate the required memory and will also 'know' how many of each entity to read in. An example of a grid file is shown as follows:

```
N_p = 812;
N_f = 1244;
N_e = 2949;
N_b = 2;
Points = [
0.00000 0.00000 0.00000
0.00000 0.02000 0.00000
```

```
0.02000 0.02000 0.00000
...
];
Faces = [
     9      10     185
     9     155       1
     9     185     155
...
];
Elements = [
   574     544     556     757
   686     133     169     740
   237     725     240     754
...
];
Boundaries = struct('name',{},'type',{},'N',{},'indices',{},'value',{});
Boundaries(1).name    = 'Fins';
Boundaries(1).type    = 'robin';
Boundaries(1).N       = 1050;
Boundaries(1).indices = [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ... ];
Boundaries(1).value   = 0.00000;
...
```

The *read* function for the C++ version was modified in order to get the data for the linear tetrahedron elements defined in the grid file. This file could be a simple Box grid or a more complex form like a CPU sink. We will be storing three arrays for this problem; an array called Points which is an N_px3 array storing the x, y and z coordinates of the points defining the grid, an array called Faces which is an N_fx3 array storing the three indices of the points defining a face in the grid, and an array called Elements which is an N_ex4 array storing the three indices of the points defining an element in the grid, we can see this definitions in the following code:

```
file >> temp >> N_p;
file >> temp >> N_f;
file >> temp >> N_e;
file >> temp >> N_b;

Points          = new double*   [N_p];
Faces           = new int*      [N_f];
Elements        = new int*      [N_e];
Boundaries      = new Boundary  [N_b];
Points[0]       = new double    [N_p*3];
Faces[0]        = new int       [N_f*3];
Elements[0]     = new int       [N_e*4];
```

Where "file" is the fstream object which handles the input file, the "temp" variable will be used to read and store some 'useless' data from the file (i.e. the description labels for each value), All the arrays to store Points, Faces and Elements, and finally a "Boundaries" variable which is an instance of the *Boundary* class.

At this point we should mention that Matlab allow us to perform basic linear algebra operations very easily with matrices and structs, however, we cannot do exactly the same with C++. As we are going to store and compute a quite big amount of data, specially when we are dealing with the global system of equations after the assemble process, we are going to make use of the provided *SparseMatrix* and *Boundary* class definitions in our C++ program. The first one allows to store a sparse matrix in an efficient compressed format and handles

basic operations between matrices like sums, subtracts, multiplications, etc., the second one is a basic data structure to store all the information related to Boundaries defined on each grid file.

Now we have all the variables ready to load all the data from the input files, and this can be done simply by looping over each parameter and structure defined in the file as follows:

```
file >> temp;
for(int p=0; p<N_p; p++){
    file >> Points[p][0] >> Points[p][1] >> Points[p][2];
}

file >> temp;
for(int f=0; f<N_f; f++){
    file >> Faces[f][0] >> Faces[f][1] >> Faces[f][2];
}

file >> temp;
for(int e=0; e<N_e; e++){
    file >> Elements[e][0] >> Elements[e][1] >> Elements[e][2] >> Elements[e][3];
}
```

At this point we can apply spatial discretization, which is the Finite Element method, to our PDE. We know that we will have a discrete system which takes the form $M\dot{T} = KT + s$ with each element matrix and load vectors defined in Equation 15,16 and 17 respectively, with that in mind now we are going to compute all those local element matrices by using the derivations performed in part 3.3.

To assemble these matrices we are going to need to evaluate Ω_e and Γ_f for the elements and faces respectively. Now in this case for the 3D Heat Equation Ω_e is the volume of each tetrahedron, which we can evaluate in Matlab as:

```
% Calculate element volumes
for e=1:N_e
    x           = Points(Elements(e, :), 1);
    y           = Points(Elements(e, :), 2);
    z           = Points(Elements(e, :), 3);
    Omega(e)    = abs(    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) ...
                        - x(1)*y(2)*z(4) + x(1)*y(4)*z(2) + x(2)*y(1)*z(4) ...
                        - x(2)*y(4)*z(1) - x(4)*y(1)*z(2) + x(4)*y(2)*z(1) ...
                        + x(1)*y(3)*z(4) - x(1)*y(4)*z(3) - x(3)*y(1)*z(4) ...
                        + x(3)*y(4)*z(1) + x(4)*y(1)*z(3) - x(4)*y(3)*z(1) ...
                        - x(2)*y(3)*z(4) + x(2)*y(4)*z(3) + x(3)*y(2)*z(4) ...
                        - x(3)*y(4)*z(2) - x(4)*y(2)*z(3) + x(4)*y(3)*z(2) ) /6;
end
```

And the C++ version which looks similar for that part would be:

```
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]     = abs(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;
}
```

In 3D Γ_f is a face area of a triangular element and we can compute this in matlab quite simply via:

```
% Calculate face areas
for f=1:N_f
    x           = Points(Faces(f, :), 1);
    y           = Points(Faces(f, :), 2);
    z           = Points(Faces(f, :), 3);
    Gamma(f)    = sqrt(((y(2)-y(1))*(z(3)-z(1)) - (z(2)-z(1))*(y(3)-y(1)))^2 ...
       + ((z(2)-z(1))*(x(3)-x(1)) - (x(2)-x(1))*(z(3)-z(1)))^2 ...
       + ((x(2)-x(1))*(y(3)-y(1)) - (y(2)-y(1))*(x(3)-x(1)))^2)/2;
end
```

And in C++ we have:

```
for(int f=0; f<N_f; f++){
    for(int p=0; p<3; p++){
        x[p]    = Points[Faces[f][p]][0];
        y[p]    = Points[Faces[f][p]][1];
        z[p]    = Points[Faces[f][p]][2];
    }
    Gamma[f]    = sqrt(pow(((y[1]-y[0])*(z[2]-z[0]) - (z[1]-z[0])*(y[2]-y[0])),2)
                + pow(((z[1]-z[0])*(x[2]-x[0]) - (x[1]-x[0])*(z[2]-z[0])),2)
                + pow(((x[1]-x[0])*(y[2]-y[0]) - (y[1]-y[0])*(x[2]-x[0])),2))/2;
}
```

The next step is to loop over the nodes of an element and evaluate the terms in the local mass matrix, stiffness matrix and the load vector. An important point to note here is that all the contributions to the load vector correspond to boundary values as well as the second term of the stiffness matrix and they will be considered later when we in fact will be computing the Boundaries. [1]

Now in order to actually implement this in our Matlab code, our assemble function is going to involve 'looping' over all of the elements in the grid, then looping over all of the nodes of each element. The code will look something like:

```
...
M_e             = [2, 1, 1, 1;
                   1, 2, 1, 1;
                   1, 1, 2, 1;
                   1, 1, 2, 1];
...
% Assemble M, K, and s
    for e=1:N_e
```

```
    Nodes       = Elements(e,:);
    x           = Points(Nodes,1);
    y           = Points(Nodes,2);
    z           = Points(Nodes,3);
    gradEta     = [((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)))]/(6*Omega(e));
    for p=1:4
        m               = Nodes(p);
        for q=1:4
            n           = Nodes(q);
            gradEta_q   = [gradEta(1,q), gradEta(2,q), gradEta(3,q)];

            M(m,n)      = M(m,n) + rho*C*M_e(p,q)*Omega(e)/20;
            K(m,n)       = K(m,n) - k*dot(gradEta_p,gradEta_q)*(Omega(e));
        end
    end
end
```

As we can see is quite easy to compute the mass matrix and the corresponding contribution to the stiffness matrix as well. The analogous C++ version of that code looks like this:

```
...
//The Mass element Matrix
double  M_e[4][4]   = {{2.0, 1.0, 1.0, 1.0},
                       {1.0, 2.0, 1.0, 1.0},
                       {1.0, 1.0, 2.0, 1.0},
                       {1.0, 1.0, 1.0, 2.0}};
...
// Assemble M, K, and s
M.initialize(N_p, 10);
K.initialize(N_p, 10);

for(int e=0; e<N_e; e++){
    for(int p=0; p<4; p++){
        Nodes[p]= Elements[e][p];
        x[p]    = Points[Nodes[p]][0];
        y[p]    = Points[Nodes[p]][1];
        z[p]    = Points[Nodes[p]][2];
    }

    gradEta[0][0] = ((y[3]-y[1])*(z[2]-z[1])-(y[2]-y[1])*(z[3]-z[1]))/(6*Omega[e]);
    gradEta[0][1] = ((y[2]-y[0])*(z[3]-z[2])-(y[2]-y[3])*(z[0]-z[2]))/(6*Omega[e]);
    gradEta[0][2] = ((y[1]-y[3])*(z[0]-z[3])-(y[0]-y[3])*(z[1]-z[3]))/(6*Omega[e]);
    gradEta[0][3] = ((y[0]-y[2])*(z[1]-z[0])-(y[0]-y[1])*(z[2]-z[0]))/(6*Omega[e]);
    gradEta[1][0] = ((x[2]-x[1])*(z[3]-z[1])-(x[3]-x[1])*(z[2]-z[1]))/(6*Omega[e]);
    gradEta[1][1] = ((x[3]-x[2])*(z[2]-z[0])-(x[0]-x[2])*(z[2]-z[3]))/(6*Omega[e]);
    gradEta[1][2] = ((x[0]-x[3])*(z[1]-z[3])-(x[1]-x[3])*(z[0]-z[3]))/(6*Omega[e]);
    gradEta[1][3] = ((x[1]-x[0])*(z[0]-z[2])-(x[2]-x[0])*(z[0]-z[1]))/(6*Omega[e]);
```

```
    gradEta[2][0] = ((x[3]-x[1])*(y[2]-y[1])-(x[2]-x[1])*(y[3]-y[1]))/(6*Omega[e]);
    gradEta[2][1] = ((x[2]-x[0])*(y[3]-y[2])-(x[2]-x[3])*(y[0]-y[2]))/(6*Omega[e]);
    gradEta[2][2] = ((x[1]-x[3])*(y[0]-y[3])-(x[0]-x[3])*(y[1]-y[3]))/(6*Omega[e]);
    gradEta[2][3] = ((x[0]-x[2])*(y[1]-y[0])-(x[0]-x[1])*(y[2]-y[0]))/(6*Omega[e]);

    // Outer loop over each node
    for(int p=0; p<4; p++){
        m       = Nodes[p];
        gradEta_p[0]    = gradEta[0][p];
        gradEta_p[1]    = gradEta[1][p];
        gradEta_p[2]    = gradEta[2][p];

        // Inner loop over each node
        for(int q=0; q<4; q++){
            n           = Nodes[q];
            gradEta_q[0]        = gradEta[0][q];
            gradEta_q[1]        = gradEta[1][q];
            gradEta_q[2]        = gradEta[2][q];

            M(m,n)      += rho*C*M_e[p][q]*Omega[e]/20;
            K(m,n)      -= k*(gradEta_p[0]*gradEta_q[0]
            +gradEta_p[1]*gradEta_q[1]
            +gradEta_p[2]*gradEta_q[2])*Omega[e];
        }
    }
}
```

Here we are computing matrices M and K, both are instances of the *SparseMatrix* class, thus we can make use of all the defined operations (methods) of that class, in fact, we are initializing the matrices in order to start assigning values to them and also computing some matrix operations like the dot product of the gradients on the right hand side of the $K(m,n)$ matrix calculation. The only thing to keep in mind is that in our C++ version we should remember that we need to initialize the *SparseMatrix* objects, assemble the coefficients with the overloaded () operator, then finalize them.

Now it's time to compute the Boundary conditions. In this case as we mention earlier, for the Robin boundaries we are going to compute the contribution to the load vector and the contribution to the stiffness matrix with the integration performed in the first and second term of Equation (17) respectively. Then we'll do the same with the second term of Equation (16), which corresponds to the Neumann boundary condition. We can do this by adding in another for loop over the boundaries within our Matlab assemble function as:

```
...
k_e             =   [2, 1, 1;
                     1, 2, 1;
                     1, 1, 2];
s_e             = [1; 1; 1; 1];
...
% Apply boundary conditions
Fixed = [];
for b=1:N_b
    if  strcmp(Boundaries(b).type, 'neumann')
        for f=1:Boundaries(b).N;
            Nodes       = Faces(Boundaries(b).indices(f),:);
            for p=1:3
```

```
                m       = Nodes(p);
                %second term contribution to the load vector
                s(m)    = s(m)+ s_e(p)*Qcpu*Gamma(Boundaries(b).indices(f))/3;

            end
        end
    elseif strcmp(Boundaries(b).type, 'robin')
       for f=1:Boundaries(b).N;
            Nodes       = Faces(Boundaries(b).indices(f),:);
            for p=1:3
                m       = Nodes(p);
                %first term contribution to the load vector
                s(m)    = s(m)+ s_e(p)*h*Tair*Gamma(Boundaries(b).indices(f))/3;
                for q=1:3
                    n = Nodes(q);
                    %Contribution to the stiffness  matrix
                    K(m,n)=K(m,n)-h*k_e(p,q)*Gamma(Boundaries(b).indices(f))/12;
                end
            end
       end
    end
    Free = [1:N_p];
end
```

Where the Free vector is indicating all of the interior Neumann and Robin boundary points where we don't actually know the values of T and the Fixed vector is indicating all of the Dirichlet boundary points where we do. In this case we don't have Dirichlet boundary conditions so in fact all the points will be assigned to the Free vector. The C++ version of the algorithm will look something like:

```
...
//The contribution to the Stiffness Matrix
double  k_e[3][3]   = {{2.0, 1.0, 1.0},
                       {1.0, 2.0, 1.0},
                       {1.0, 1.0, 2.0}};
//The contribution to the Load vector
double  s_e[4]      = {1.0, 1.0, 1.0, 1.0};
...
//Apply Boundary Conditions
for(int b=0; b<N_b; b++){
    if      (Boundaries[b].type_=="neumann"){
        for(int f=0; f<Boundaries[b].N_; f++){
            for(int p=0; p<3; p++){
                Nodes[p]    = Faces[Boundaries[b].indices_[f]][p];
                m           = Nodes[p];
                //second term contribution to the load vector
                s[m]       += s_e[p]*Qcpu*Gamma[Boundaries[b].indices_[f]]/3;
            }
        }
    }
    else if (Boundaries[b].type_=="robin"){
        for(int f=0; f<Boundaries[b].N_; f++){
            for(int p=0; p<3; p++){
                Nodes[p]    = Faces[Boundaries[b].indices_[f]][p];
                m           = Nodes[p];
                //second term contribution to the load vector
                s[m]       += s_e[p]*h*Tair*Gamma[Boundaries[b].indices_[f]]/3;
```

```
                for (int q=0; q<3; q++){
                    Nodes[q]    = Faces[Boundaries[b].indices_[f]][q];
                    n = Nodes[q];
                    //Contribution to the stiffness matrix
                    K(m,n)     -= h*k_e[p][q]*Gamma[Boundaries[b].indices_[f]]/12;
                }
            }
            Free[m] = true;
        }
    }
}
K.finalize();
M.finalize();
```

Now the final step is the solution of our global assembled system, and we will be partitioning the matrix such that we have:

$$A_{Free,Free}T_{Free}^{l+1} = \mathbf{b}_{Free} - A_{Free,Fixed}T_{Free}^{l+1}$$

And so we solve a reduced system of equations and with Matlab this could be done as follows:

```
phi(Free,l+1) = A(Free,Free)\(b(Free) - A(Free,Fixed)*phi(Fixed,l+1));
```

Where *phi* is the general name of our field variable T. Here we will only compute the solution for the interior points and so as long as the Robin boundary points are initialized correctly in phi, then we will have imposed the boundary conditions correctly. Now that we have shown how we assemble our system of equations, we are now in a position to write out the core of the program as:

```
...
A = M - Delta_t*K;
...
% Time marching loop
for l=1:N_t-1

    b = M*phi(:,l) + Delta_t*s;
    phi(Free,l+1)    = A(Free,Free)\(b(Free) - A(Free,Fixed)*phi(Fixed,l+1));

    % Plot phi at every timestep
    set(Solution, 'CData', phi(:,l+1));
    title(['t = ' num2str(t(l+1))]);
    drawnow;
end
```

If we want to show the results in Matlab, we can do it by updating a Figure inside the time marching loop. At this point we can compute the maximum steady state temperature of the Box grid after 100s which is: 342.427K. The complete Matlab program is given in *Assignment2.m* and the final results obtained with the simulations of our Matlab program are shown on Figure 2. Further captures are shown in **APPENDIX A**.

For our C++ version the time marching loop will be a little bit more complex since we don't have the backslash operator to perform matrix solving easily. So we are going to explain the most relevant parts of the following code:

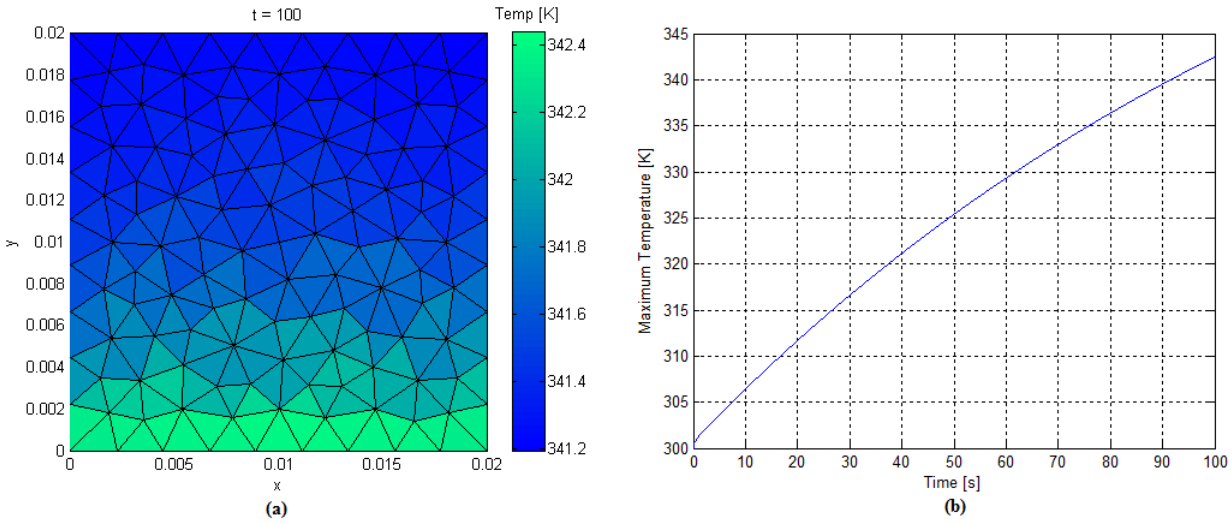

Figure 2: *(a) Heat flux trough an unstructured Box grid using linear tetrahedral elements. (b) Maximum temperature after 100s*

```
1. A = M;
2. A.subtract(Delta_t, K); // At this point we have A = M-Delta_t*K
3.
4. // Compute the column vector to subtract
5. //from the right hand side to take account of fixed nodes
6. A.multiply(AphiFixed, phi, Free, Fixed);
7.
8. writeData(phi, Points, Elements, N_p, N_e, 0);
9.
10. // Time marching loop
11. for(int l=0; l<N_t-1; l++){
12. // Assemble b
13. M.multiply(b, phi);
14. for(int m=0; m<N_p; m++){
15.     b[m]    += Delta_t*s[m] - AphiFixed[m];
16. }
17.
18. // Solve the linear system
19. solve(A, phi, b, Free, Fixed);
20.
21. // Write the solution
22. if (l%10==0){
23.    writeData(phi, Points, Elements, N_p, N_e, (l+1));
24. }
25.}
```

In order to make the necessary computation as simple as possible the first member function that we are going to add will overload the = operator so that we can have the line of code A=M; in our program and the resulting function call will copy all of the data stored in the `val_`, `col_`, `row_`, arrays, etc. A similar computation is performed in *Line 2*, where internally the *SparseMatrix* class will handle all the memory allocation, operations and method calls in order to perform the subtract operation. At this point Following the use of the = and the subtract functions we will have the correctly assembled the matrix A. The next step is to compute the column vector to subtract from the right hand side to take account of fixed nodes, for that we will perform a matrix vector multiplication, taking as an input a 1D array defining the vector

that the matrix should be multiplied by, and a vector that is the output of the matrix vector multiplication (*Line 6*).

What we can see in *Line 8* is the *writeData* method, which in this case will be printing the initial state of the system using the grid structure as well as the Points, Elements and the Temperature changes into Vtk files in order to visualize the results in Paraview. We are not going to discuss the structure of this method as we consider this is not so relevant because it is used only for visualization and as we will see later, it will be omitted for scaling and performance testing. However, the code inside looks fairly simple and the only thing to take account of is that we have to follow the corresponding Vtk file structure defined for our linear tetrahedron elements.

Now let's move on to the time marching loop, After the first multiply statement, the 1D array **b** will contain the matrix vector product $M\phi^l$, then after the for loop $M\phi^l + \Delta ts$ With this array evaluated for any given time step, we call the solve function, which will look like:

```
...
// Compute the initial residual
A.multiply(Aphi, phi, Free, Free);
for(m=0; m<N_row; m++){
    if(Free[m]){
        r_old[m]    = b[m] - Aphi[m];
        d[m]        = r_old[m];
        r_oldTr_old+= r_old[m]*r_old[m];
    }
}
r_norm = sqrt(r_oldTr_old);

// Conjugate Gradient iterative loop
while(r_norm>tolerance && k<N_k){
    dTAd    = 0.0;
    A.multiply(Ad, d, Free, Free);
    for(m=0; m<N_row; m++){
        if(Free[m]){
            dTAd   += d[m]*Ad[m];
        }
    }
    alpha   = r_oldTr_old/dTAd;
    for(m=0; m<N_row; m++){
        if(Free[m]){
            phi[m] += alpha*d[m];
        }
    }
    for(m=0; m<N_row; m++){
        if(Free[m]){
            r[m]    = r_old[m] - alpha*Ad[m];
        }
    }
    rTr = 0.0;
    for(m=0; m<N_row; m++){
        if(Free[m]){
            rTr  += r[m]*r[m];
        }
    }
```

```
        beta    = rTr/r_oldTr_old;
        for(m=0; m<N_row; m++){
            if(Free[m]){
                d[m] = r[m] + beta*d[m];
            }
        }
        for(m=0; m<N_row; m++){
            if(Free[m]){
                r_old[m] = r[m];
            }
        }
        r_oldTr_old = rTr;
        r_norm      = sqrt(rTr);
        k++;
    }
    ...
    return;
```

The complete program is given in Assignment2.cpp. **APPENDIX B** illustrates the solution at a number of different time steps. It can be observed that as time progresses constant flux of heat on the bottom of the Box moves through the domain (due to the convective term) and continues heating the rest of it until 100s.

4.2 Parallel MPI C++ implementation

Now we are going to go through the MPI version of our C++ code. Again we took as reference the implementation of the given code in *Example21-4.cpp* and *Tutorial6.cpp* which solve the generic 2D scalar transport equation with MPI. For this part we are not going to use the concept of ghost cells in our parallel computation like in the first assignment, instead of this, each process will be responsible for a unique set of points where each process will have a local numbering of its Points, Faces, and Elements arrays, but there will also be an implicit global numbering. To do this we will break up our unstructured grid to assign the elements to each process as shown in Figure 3. In this case, the solution at a point on the boundary between two grids will be computed by each process sharing that point. Thus for the points that are shared between processes we are going to define a new type of boundary condition called an interprocess boundary condition.

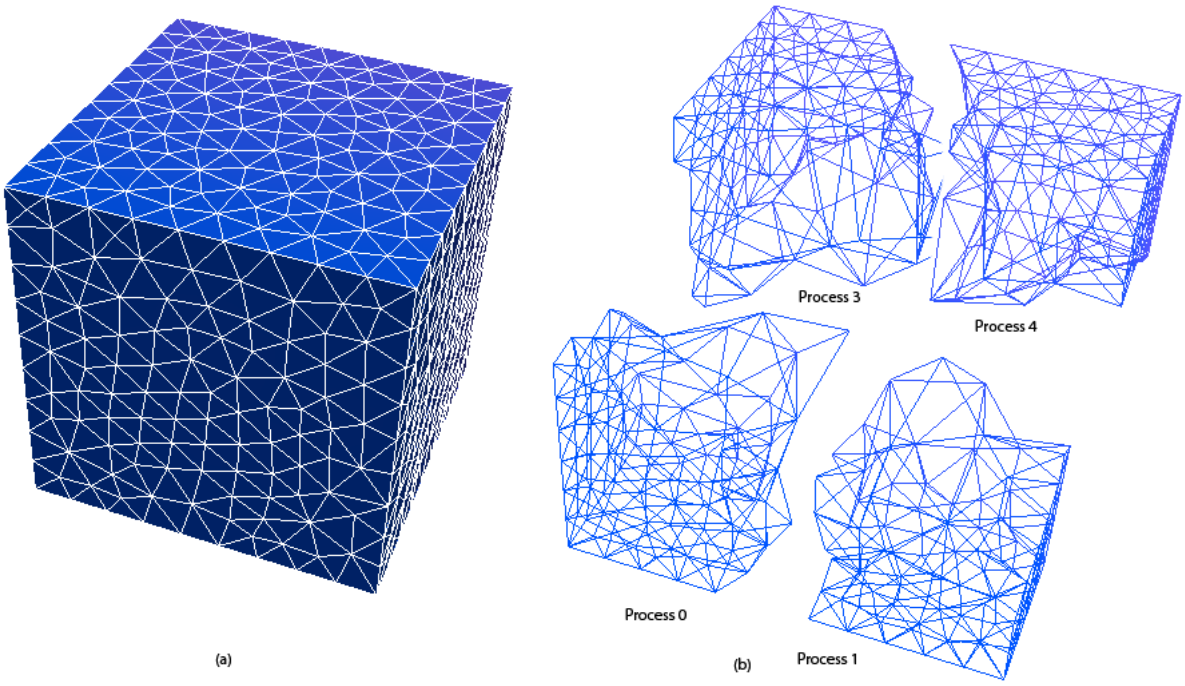

Figure 3: *(a). A schematic of the partitioned 3D problem domain for the Heat equation with a Box grid. (b). Portions of the grid that are mapped to each process, for this illustration we show 4 of the N processes that the Box grid could be divided*

For this part we are going to assume that we already have the grid files perfectly defined and ready to be processed. The only difference is that now we will have one grid file per process, and each process will 'work' with its own part of the data and will have its unique set of elements sharing with other processes only the configured interprocess data defined at the end of those files.

So, although the elements are uniquely assigned to a process, certain points on the boundaries between processes will be duplicated and we can avoid this by the concept of ownership. That is although a face is shared between two cells, we pick one of those cells to be the 'owner' of the face and one to be the 'neighbour' of the face. Here will be assigning one of the two processes sharing the interprocess points to be the owner of them.

From here we are going to modify some methods of the original C++ sequential version of the program, so in fact we firstly will do so with our *readData* method. The only thing that is going to change is the definition of the interprocess boundaries which can be done as follows:

```
...
for(int b=0; b<myN_b; b++){
    file >> Boundaries[b].name_ >> Boundaries[b].type_ >> Boundaries[b].N_;
    Boundaries[b].indices_  = new int [Boundaries[b].N_];
    for(int n=0; n<Boundaries[b].N_; n++){
        file >> Boundaries[b].indices_[n];
    }
    file >> Boundaries[b].value_;
    if(Boundaries[b].type_=="interprocess"){
        myMaxN_sb++;
        myMaxN_sp   = max(myMaxN_sp, Boundaries[b].N_);
```

```
        yourID      = static_cast<int> (Boundaries[b].value_);
        if(yourID>myID)
        {
            for(int p=0; p<Boundaries[b].N_; p++){
                yourPoints[Boundaries[b].indices_[p]]   = true;
            }
        }
    }
}

MPI_Allreduce(&myMaxN_sp, &maxN_sp, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
buffer      = new double [maxN_sp];
bufferSize  = (maxN_sp*sizeof(double)+MPI_BSEND_OVERHEAD)*myMaxN_sb;
MPI_Buffer_attach(new char[bufferSize] , bufferSize);
...
```

As we can see in the code we are looping over the interprocess boundary definitions in the file, we have declared a variables `MaxN_sp`, which is an integer variable storing the maximum number of shared points on any given interprocess boundary, and `myMaxN_sp` , which is an integer storing the number of interprocess boundaries that a given process has. This will be used in order to allocate a buffer array to use in the MPI send and receive routines. Then as mentioned before, we assign ownership to the process with the highest rank and therefore set the corresponding entries in the *yourPoints* array equal to "true". Finally, after having read in the file, we will make sure that every process has the same global value for `MaxN_sp` by using the *MPI_Allreduce* function, then we will allocate the buffer for MPI to store data until it can be exchanged.

At this point, the next step is the assembling of the mass and stiffness matrices. As it happens we don't need to modify our code for this at all and so the assemble function will be exactly the same as our sequential version of the program.

So, when each process comes to solving its system of equations at each time step, the equations for each shared point will be missing some terms and if we want our algorithm to work, we need to share this data in order to make sure that the shared point equations are the same on each process [1]. In order to illustrate how we make this work, let's imagine that we have just completed the execution of the assemble function and we have the incomplete M and K, and s.

We will define a function called exchange that we will call repeatedly throughout the program and we will pass this a 1D array, representing a column vector of size N_px1, which will look like:

```
void    exchangeData(double* v, Boundary* Boundaries, int myN_b)
{
    int         yourID      = 0;
    int         tag         = 0;
    MPI_Status  status;

    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);
        }
    }
    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];
        }
    }
    }
    return;
}
```

In this method we will be going to loop over the interprocess boundaries and find which points are shared and copy that information into the buffer array that was defined in the *readData* method. We will then send off this buffer to the process with whom these points are shared. We will do so with a "buffer Send" in order to avoid deadlock. So now we can use the exchange method after the calling to the assemble method, so that we ensure that M and K and s, doesn't have missing entries.

Now we can move on to the time marching loop where we actually compute the solution by calling the solve method.

```
...
A = M;
A.subtract(Delta_t, K); // At this point we have A = M-Delta_t*K

// Compute the column vector to subtract
//from the right hand side to take account of fixed nodes
A.multiply(AphiFixed, phi, Free, Fixed);
exchangeData(AphiFixed, Boundaries, myN_b);
exchangeData(s, Boundaries, myN_b);

writeData(phi, Points, Elements, myN_p, myN_e, 0, myID);

// Time marching loop
for(int l=0; l<N_t-1; l++){
    if(myID==0){
        cout << "t = " << l*Delta_t;
    }

    // Assemble b
    M.multiply(b, phi); // b = M*phi^l
    exchangeData(b, Boundaries, myN_b);
    #pragma omp parallel for default(none) shared(myN_p,b,s,AphiFixed)
    for(int m=0; m<myN_p; m++){
        b[m]    += Delta_t*s[m] - AphiFixed[m];
    } // b = M*phi^l + Delta_t*s - A_free,fixed*phi_fixed

    // Solve the linear system
    solve(A, phi, b, Free, Fixed, Boundaries, yourPoints, myN_b, myID);//Changed

    // Write the solution
    if(l%10==0){
```

```
        writeData(phi, Points, Elements, myN_p, myN_e, l, myID);
    }
}

if(myID==0){
    // Record the end time and calculate elapsed time
    wtime   = MPI_Wtime() - wtime;
    cout << "Simulation took " << wtime << " seconds with "
    << N_Procs << " processes and " << N_Threads << " Threads" << endl;
}

MPI_Buffer_detach(&buffer, &bufferSize); //Added

// Deallocate arrays
for(int boundary=0; boundary<myN_b; boundary++)
{
    delete [] Boundaries[boundary].indices_;
}
...
MPI_Finalize();
...
```

So, it can be observed that at this stage most of the code is the same as the serial version of the algorithm with the exception of the exchanging of the vectors s and b. We can now look at how we solve the global linear system of equations at each time step with the conjugate gradient method.

```
void    solve(SparseMatrix& A, double* phi, double* b, bool* Free, bool* Fixed,
Boundary* Boundaries, bool* yourPoints, int myN_b, int myID)
{
...
    // Compute the initial residual
    A.multiply(Aphi, phi, Free, Free);
    exchangeData(Aphi, Boundaries, myN_b); //Added
    for(m=0; m<N_row; m++){
        if(Free[m]){
            r_old[m]    = b[m] - Aphi[m];
            d[m]        = r_old[m];
        }
    }

    r_oldTr_old = computeInnerProduct(r_old, r_old, Free, yourPoints, N_row);
    r_norm      = sqrt(r_oldTr_old);

    // Conjugate Gradient iterative loop
    while(r_norm>tolerance && k<maxIterations)
    {
        A.multiply(Ad, d, Free, Free);
        exchangeData(Ad, Boundaries, myN_b);
        dTAd    = computeInnerProduct(d, Ad, Free, yourPoints, N_row);
        alpha   = r_oldTr_old/dTAd;
        for(m=0; m<N_row; m++){
            if(Free[m]){
                phi[m] += alpha*d[m];
            }
        }
        for(m=0; m<N_row; m++){
```

```
            if(Free[m]){
                r[m]    = r_old[m] - alpha*Ad[m];
            }
        }
        rTr = computeInnerProduct(r, r, Free, yourPoints, N_row);
        beta    = rTr/r_oldTr_old;
        for(m=0; m<N_row; m++){
            if(Free[m]){
                d[m] = r[m] + beta*d[m];
            }
        }

        for(m=0; m<N_row; m++){
            if(Free[m]){
                r_old[m] = r[m];
            }
        }
        r_oldTr_old = rTr;
        r_norm      = sqrt(rTr);
        k++;
    }
    ...
}
```

As we can see the exchange function is quite important here because it help us to correct possible missing entries in our global matrix system. Rather than trying to exchange the entries in the matrix itself what we do is exchange the vectors Aϕ and A**d**, which means less data to exchange, and much simpler code[1]. The next step is the computation of the residuals and the search directions. To do that we are going to use the *innerProduct* function which will be responsible for computing the correct global inner products that are used to define the residual norm, α and β. That is because each process has a portion of the grid, then it will only have a portion of the global residual vector, but we need each process to have the correct global residual in order to use compute the same α and β. Otherwise the solution may not be computed correctly.

```
double  computeInnerProduct(double* v1, double* v2, bool* Free,
bool* yourPoints, int N_row)
{
    double      myInnerProduct  = 0.0;
    double      innerProduct    = 0.0;

    for(int m=0; m<N_row; m++){
        if(Free[m] && !yourPoints[m]){
            myInnerProduct += v1[m]*v2[m];
        }
    }

    MPI_Allreduce(&myInnerProduct, &innerProduct, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

    return innerProduct;
}
```

Here we are using the *yourPoints* array in order to decide whether or not to include the values in the increment. If we didn't do this, then the shared points would be counted more than once in the computation of the residual vector (and all of the other vectors). So this is why it was important to determine the ownership of a shared point right back when reading in the grid on each process[1].

Finally, after each process computes its portion of the inner product we use an MPI Allreduce so that each process will have the correct global residual; and will then compute the same update for ϕ, will then compute the correct new search directions, etc.

In order to show the results of the MPI version of the code, it's the perfect time to test the CPU sink unstructured grid instead of the Box grid used so far only for explanation purposes. In this case we are going to run our MPI program with 8 processes and as input the grid file *"CPUHeatSink_08proc"*. The final state of the temperature is shown in Figure 4 and some captures of the simulation are listed on **APPENDIX C**.

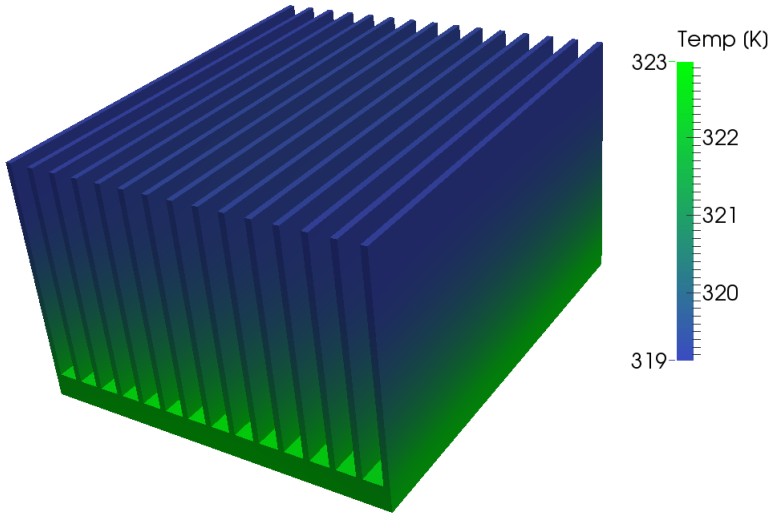

Figure 4: *Simulation results in Paraview with the output of the parallel MPI version of the code with 8 processes with an unstructured grid of a CPU sink*

4.3 Hybrid openMP and MPI implementation

Hybrid parallel programming consists of mixing several parallel programming paradigms in order to benefit from the advantages of the different approaches. In general, MPI is used for communication between processes, and another paradigm (OpenMP, pthreads, etc) is used inside each process. Our final approach includes the implementation of an Hybrid C++ version of our program with the use of MPI with OpenMP.

The main purpose is to exploit the strengths of both models: the efficiency, memory savings, and ease of programming of the shared-memory model and the scalability of the distributed-memory model. The advantages of this type of implementations allow us (in most of the cases) to improve speedup, but involves complexity and higher level of expertise. Also the necessity of having good MPI and OpenMP performances (Amdahl's law applies separately to the two approaches) and the total gains in performance are not guaranteed (extra additional costs)[2]. Figure 5 illustrates how an Hybrid implementation of MPI and openMP will work. As we can see this kind of parallel programs consists of cooperating processes, each with its own memory. A group of threads is executed within each process and there is interchange of data between one to another process as messages.

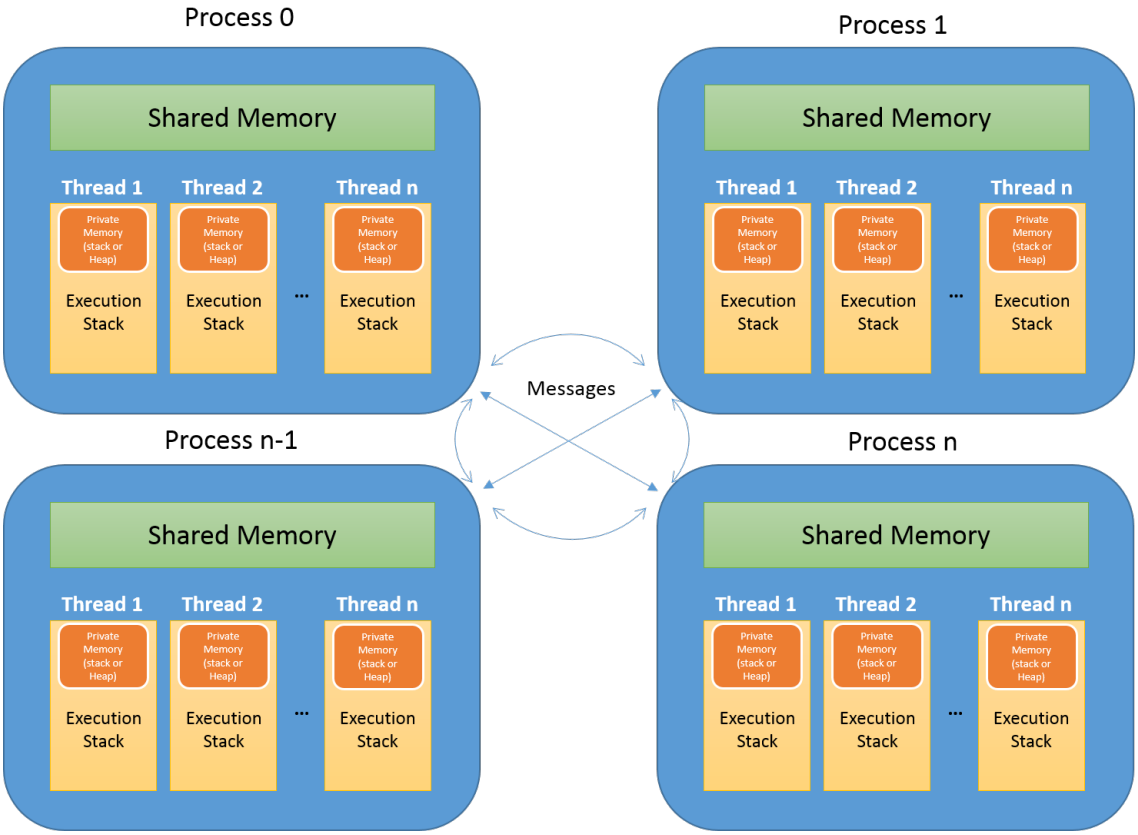

Figure 5: *Hybrid parallel implementation with MPI and openMP*

Now that we have an idea of what are the expected behavior of an Hybrid parallel program, it's time to go through our MPI C++ code and start adding multithreading to it. The most critical part of the code is the *solve* method, given that we have more than one Mpi calls when we use the exchange method, we have to make sure that the pragmas are not inside them, otherwise we may be sending incorrect data between processes and causing the program to 'crash'. With that in mind, the first for loop to be parallelized is the one where we assemble **b**, which is inside the time marching loop:

```
...
// Assemble b
M.multiply(b, phi); // b = M*phi^l
exchangeData(b, Boundaries, myN_b);
#pragma omp parallel for default(none) shared(myN_p,b,s,AphiFixed)
for(int m=0; m<myN_p; m++)
{
    b[m]    += Delta_t*s[m] - AphiFixed[m];
} // b = M*phi^l + Delta_t*s - A_free,fixed*phi_fixed
...
```

As we can see *after* exchanging the boundary values between processes we create a team of parallel threads with the *pragma omp parallel for* outside the for loop where the **b** array is assembled. The *myN_p,b,s,AphiFixed* variables will be shared by all the threads.

```
// Compute the initial residual
```

```
A.multiply(Aphi, phi, Free, Free);
exchangeData(Aphi, Boundaries, myN_b);
#pragma omp parallel for default(none) shared(N_row, Free, r_old, b, Aphi, d)
for(m=0; m<N_row; m++){
    if(Free[m]){
        r_old[m]    = b[m] - Aphi[m];
        d[m]        = r_old[m];
    }
}

// Solve the linear system
solve(A, phi, b, Free, Fixed, Boundaries, yourPoints, myN_b, myID);
...
```

Then we make the call to the *solve* method where we are going to solve our system of equations using the calculate the conjugate gradient method. Again here the pragmas are inserted in all the for loops after the *exchangeData* call.

```
...
// Conjugate Gradient iterative loop
    while(r_norm>tolerance && k<maxIterations)
    {
        A.multiply(Ad, d, Free, Free);
        exchangeData(Ad, Boundaries, myN_b);
        dTAd    = computeInnerProduct(d, Ad, Free, yourPoints, N_row);
        alpha   = r_oldTr_old/dTAd;

        #pragma omp parallel for default(none) shared(N_row, Free, phi, alpha, d)
        for(m=0; m<N_row; m++){
            if(Free[m]){
                phi[m] += alpha*d[m];
            }
        }

        #pragma omp parallel for default(none) shared(N_row, Free, r, r_old, alpha, Ad)
        for(m=0; m<N_row; m++){
            if(Free[m]){
                r[m]    = r_old[m] - alpha*Ad[m];
            }
        }
        rTr = computeInnerProduct(r, r, Free, yourPoints, N_row);
        beta    = rTr/r_oldTr_old;

        #pragma omp parallel for default(none) shared(N_row, Free, d, r, beta )
        for(m=0; m<N_row; m++){
            if(Free[m]){
                d[m] = r[m] + beta*d[m];
            }
        }

        #pragma omp parallel for default(none) shared (N_row, Free, r_old, r)
        for(m=0; m<N_row; m++){
            if(Free[m]){
                r_old[m] = r[m];
            }
        }
        r_oldTr_old = rTr;
```

```
        r_norm      = sqrt(rTr);
        k++;
    }
...
}
```

Finally, the last part where we can add multithreading is the *innerProduct* method. This method computes the inner product of the parameter arrays v1 and v2, so that a private copy of the *myInnerProduct* array variable is created for each thread using the reduction clause, the reduction variable is applied to all private copies of the shared variable, and at the end the final result is written to the global shared variable.

```
#pragma omp parallel for default(none)
        shared(N_row, Free, yourPoints, v1,v2)
        reduction (+:myInnerProduct)
for(int m=0; m<N_row; m++){
    if(Free[m] && !yourPoints[m]){
        myInnerProduct += v1[m]*v2[m];
    }
}
```

The Hybrid version of the C++ code was added to the *Assignment2_MPI.cpp* file and can be executed by passing as parameter the path to the grid files folder and the corresponding number of processes.

5 Scalability Results

Having our Hybrid MPI-OMP solution ready we started to test scalability across several different version of structured grids. First we made scalability tests on a Box Grid (1861 points). After that we ran scalability tests on a Heat Sink of 350 thousand points, 1.2 millon points and 3.6 millon points respectively. The tests were executed on the BlueGeneQ high performance computing cluster.

5.1 Box Grid Scaling

Our first approach was using a Box Grid of 1861 points. Figure 6 illustrates scaling results. As we can see there is no speedup in solving the Heat Equation on this grid with 8 processes due the relatively small problem size.

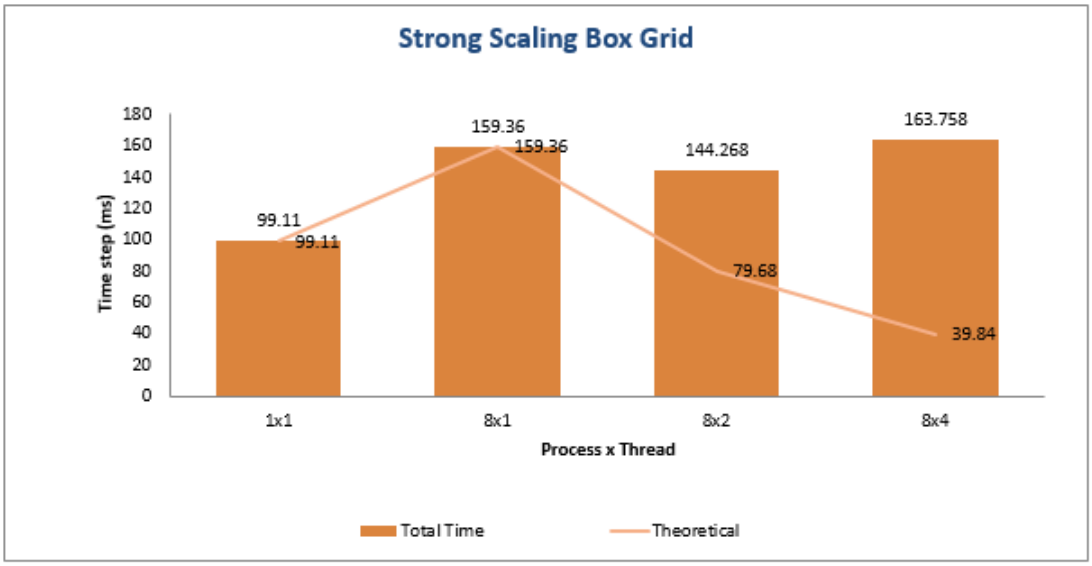

Figure 6: *Box Grid Scaling Results*

5.2 Heat Sink 350k Scaling

Because the 350k points Heat Sink grid represents a bigger computational problem, scalability results are more inline with the expected behaviour. Figure 7 illustrates the speed up improvement across the tests ran on 1, 8, 16, 32, 64 and 128 processors each one with only 1 thread. The line series displays the theoretical maximum time that may be achieved. The reason for the nonoptimal results lies in the overhead added by MPI and the delay added by the network.

The speedup when comparing different threads within the same number of process are also promising. In Figure 8 we can see that when more threads are added to the same number of process the solution speedup increases. For the 64 process x 16 threads and 128 x 16 threads combinations the time step increases instead of decrease, we think again this could be because of the 350k points problem size is too small for this computational power.

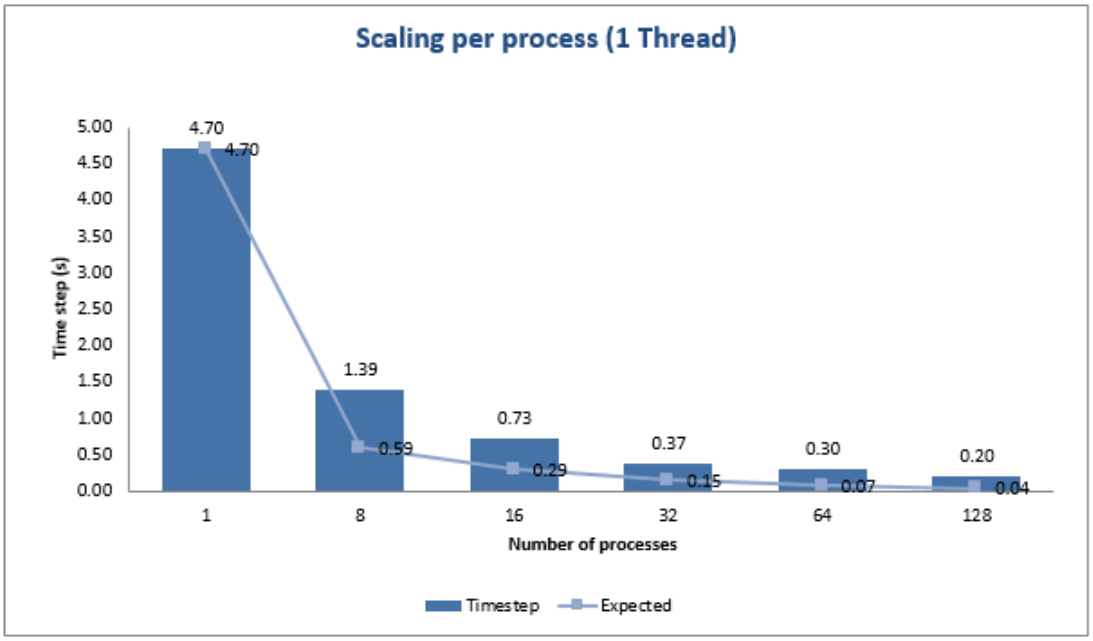

Figure 7: *350k points - 1 Thread Heat Sink Scaling Results*

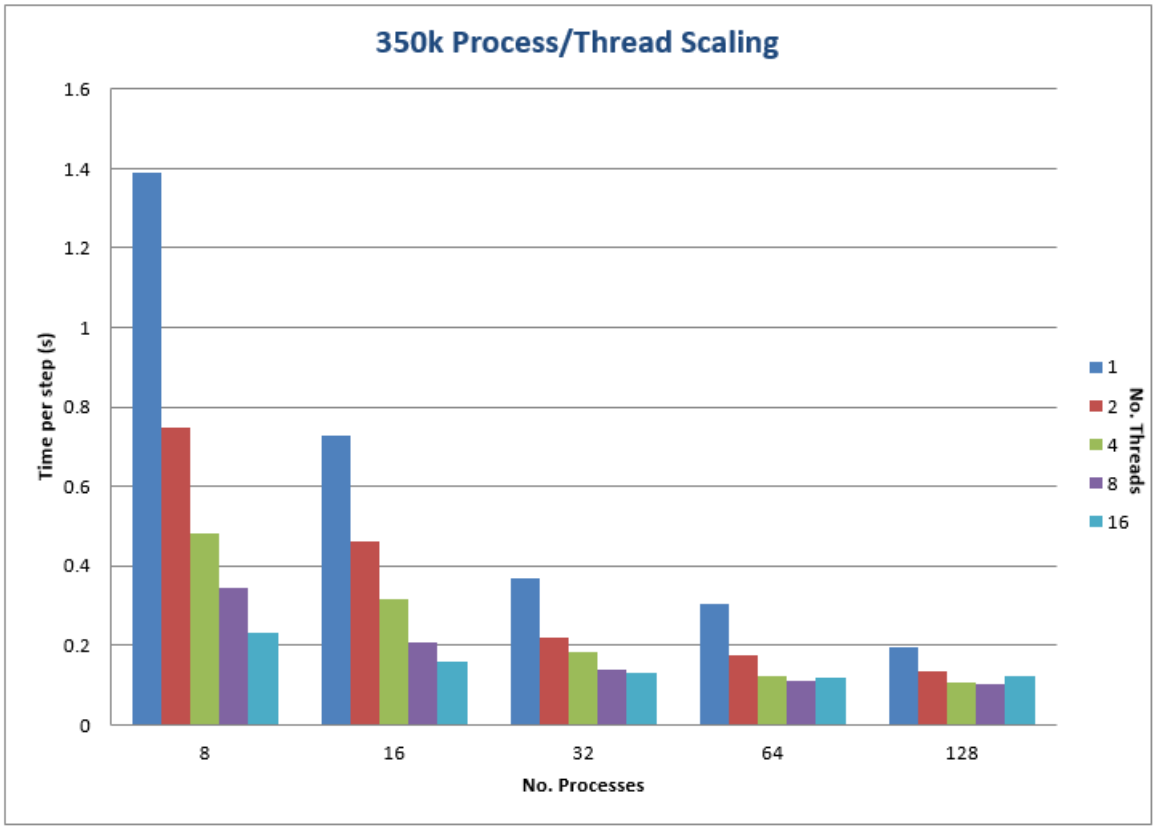

Figure 8: *350k points - ProcessxThread Heat Sink Scaling Results*

5.3 Heat Sink 1.2M Scaling

The next set of tests were with a 1.2 millon points Heat Sink grid, that is with a problem size increased by a factor approximately of 3.5 compared with the 350k grid. With this problem size it

is expected not to have speedup problems at a high number of process per thread combinations.

Figure 9 and Figure 10 shows the results of the scalability tests. There are positively speedup results when the number of process are increased with any combination of threads, although still at 64 process x 16 threads and 128 process x 16 threads the timestep improvement is not significant. Again, results are shown below with the theoretical expectations due to MPI and network overhead.

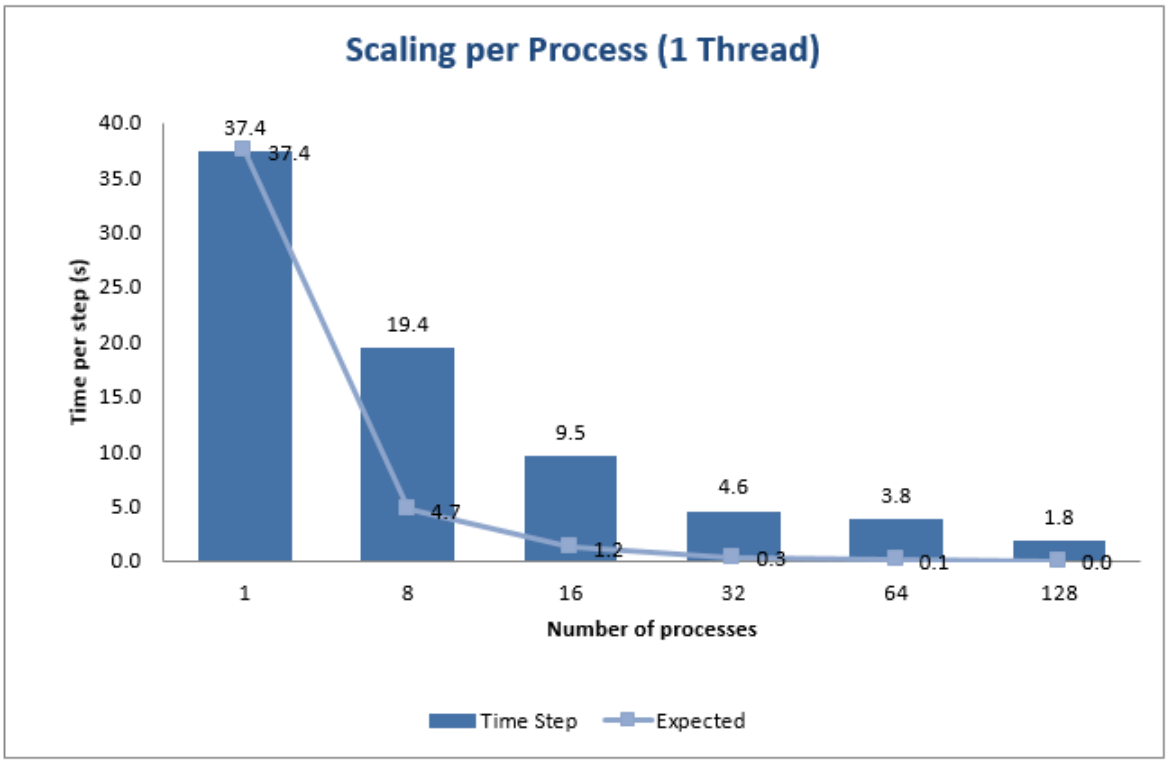

Figure 9: *1.2M points - 1 Thread Heat Sink Scaling Results*

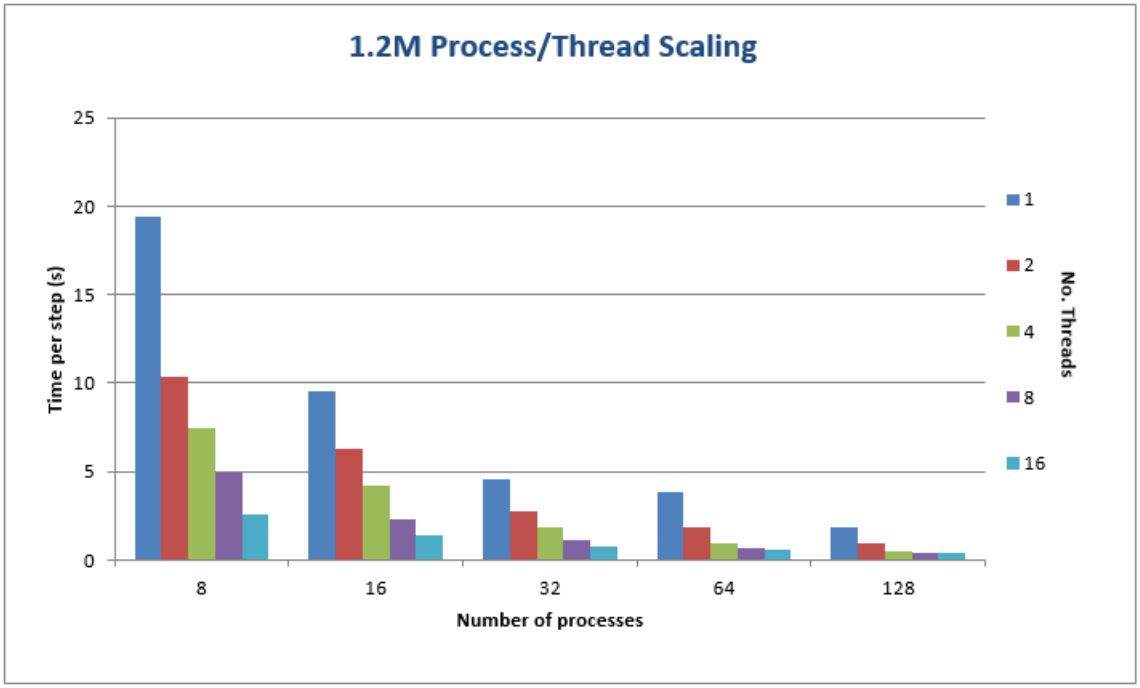

Figure 10: *1.2M points - ProcessxThread Heat Sink Scaling Results*

5.4 Heat Sink 3.6M Scaling

The last set of testings were done using a 3.6 millon points Heat Sink grid. This grid represents the biggest problem size we tested with this problem. The results of these test are inline with what we seen in the 1.2M tests. There is a significant reduction of the time in line with the expectations. Figures 11 illustrates the speedup results for different process combinations with 1 thread. Because this is the grid's size is bigger in Figure 12 it is clear that the computation is still scalable even with huge computational allocations. The speedup can be seen across all combinations of process and threads.

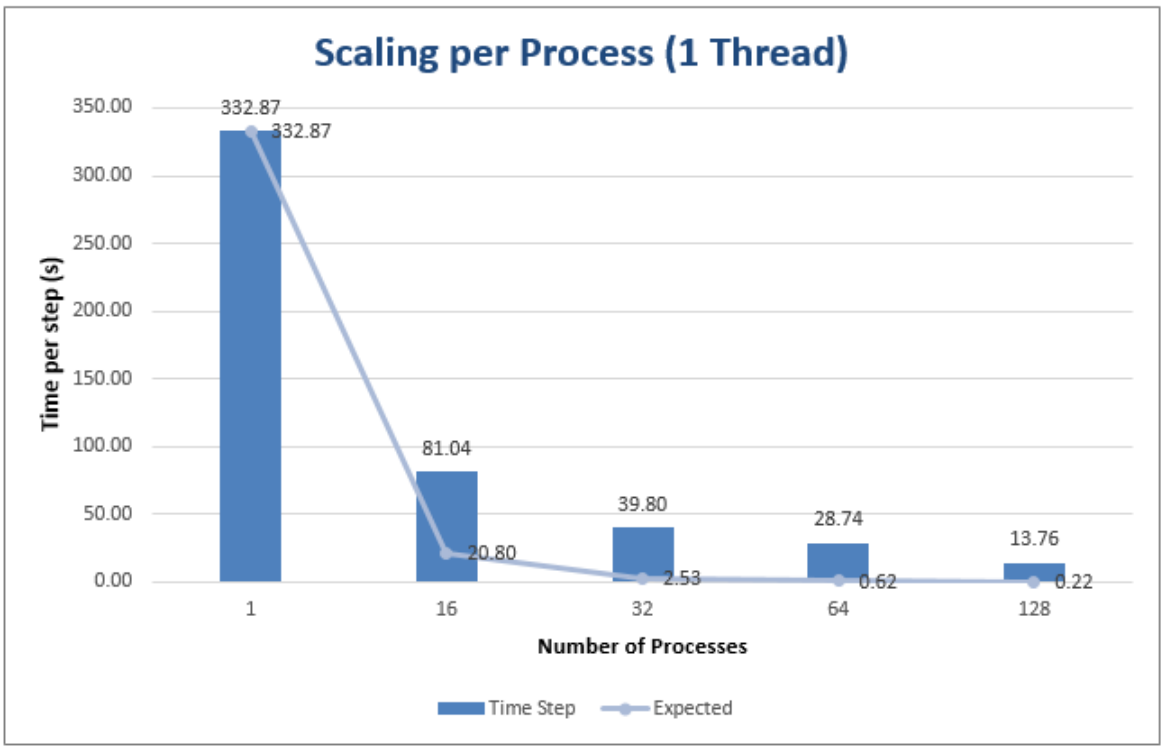

Figure 11: *3.6M points - 1 Thread Heat Sink Scaling Results*

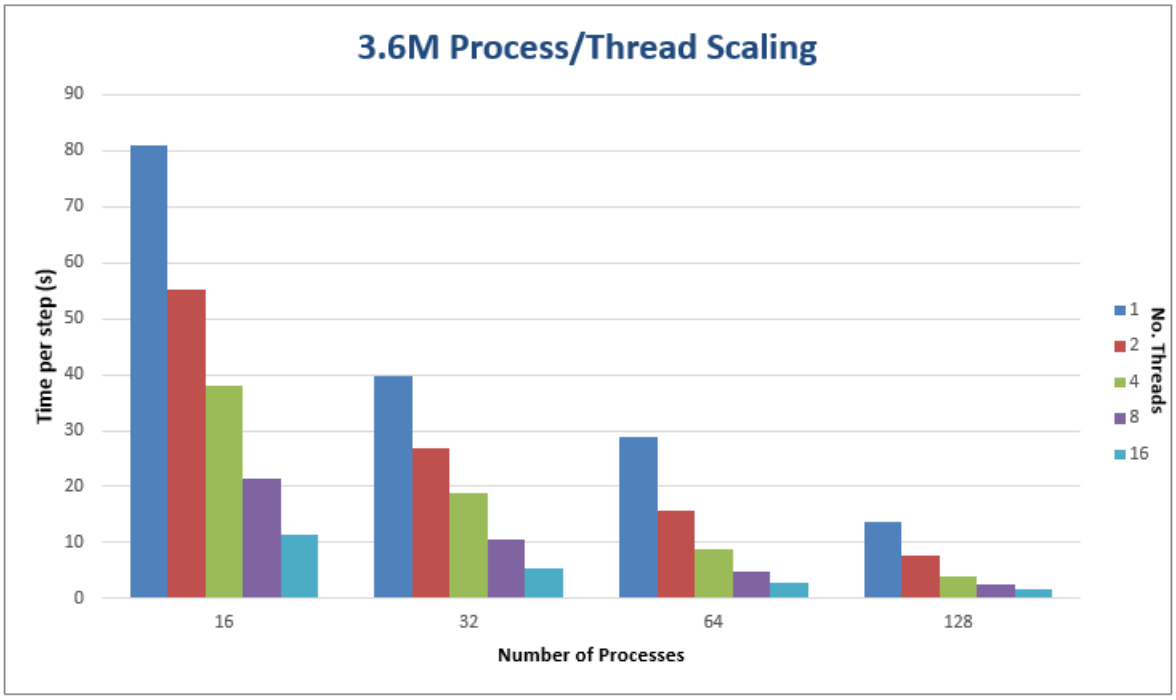

Figure 12: *3.6M points - ProcessxThread Heat Sink Scaling Results*

6 Discussion

Through this Assignment we have been following the process of solving a PDE (The heat equation) with the Finite element method. This process included a series of steps starting by the

discretization of the domain by deriving the weak form of the PDE with the Garleking method of weighted residuals and the shape functions using 3D linear tetrahedron elements. For the assemble process, computation of boundary conditions and the solution of the problem we implemented a Matlab program which results showed the heat flux on a Grid box (**APPENDIX A**). Then we developed the c++ sequential version of the program with Implicit Euler Method and the Conjugate Gradient Method for solving the resulting system of equations. After that we developed a parallel Hybrid version of our program using MPI and openMP and showed results of a heat flux in a 3D CPU sink grid (**APPENDIX C**). Finally we performed some execution and speedup tests on avoca.

With the obtained results we can conclude:

- The finite element method (FEM) is a powerful numerical technique for solving problems which are described by partial differential equations and can help us to model, discretize and solve complex domain problems.
- With this Assignment we studied some important concepts of the FEM and we applied them using Hybrid parallel programming considering High performance computing aspects.
- We consider the most difficult part was the understanding and derivation of all the governing equations correctly, However that was clarified when we started to develop the programs.
- Final speedup is limited by the purely sequential fraction of the code (Amdhals law).
- Scability is limited due to the additional costs related to the MPI library and load balancing management.
- The MPI version still gives the best results in almost every case. It appears that the problem always comes from communications which take more time in the hybrid version.

7 Appendices

APPENDIX A. Matlab simulation on a Box grid with Finite Element Method for linear 3D tetrahedral elements and Implicit Euler Method.

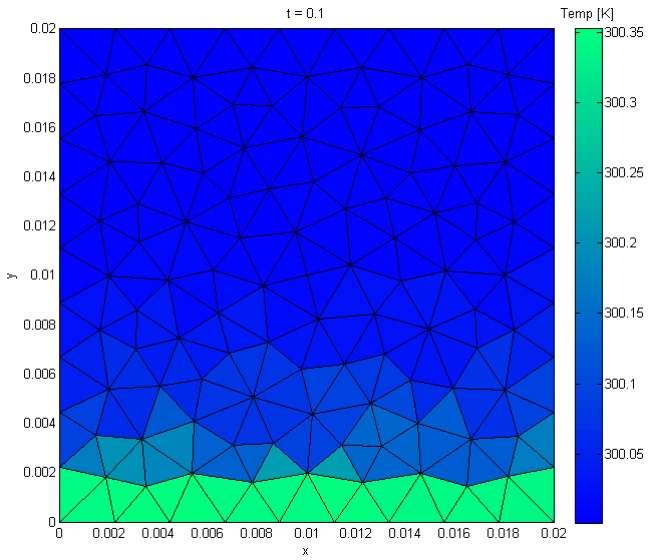

Figure 13: Initial state of Temperature

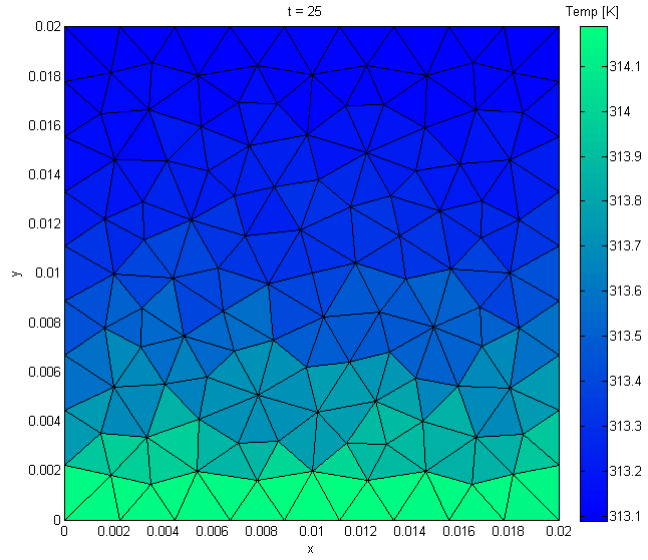

Figure 14: Temperature after 25 time step

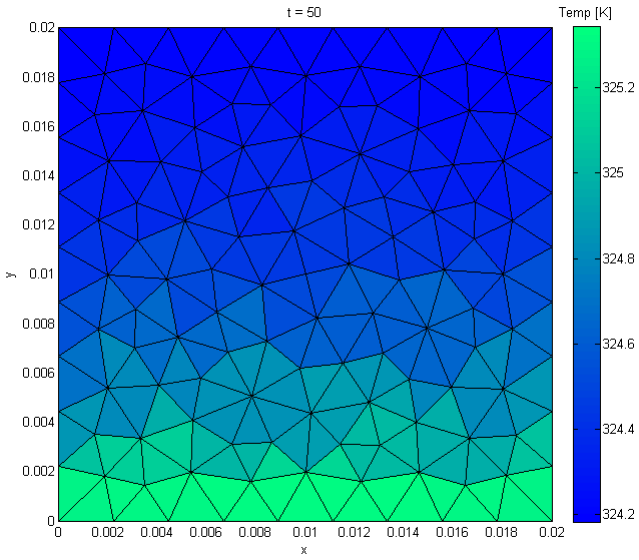

Figure 15: Temperature after 50 time steps

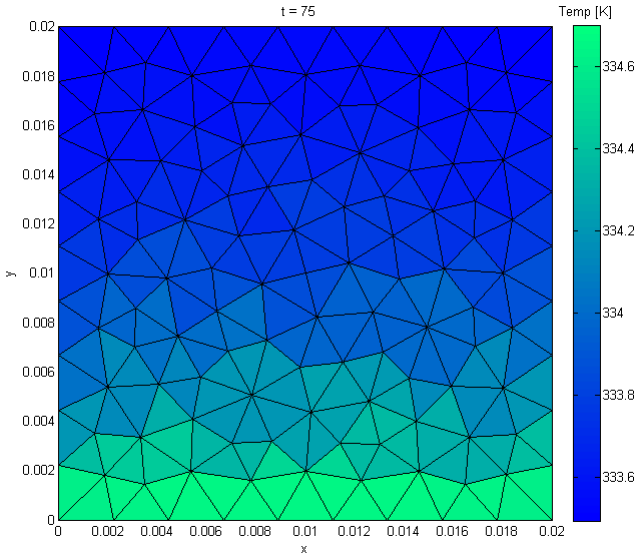

Figure 16: Temperature after 75 time steps

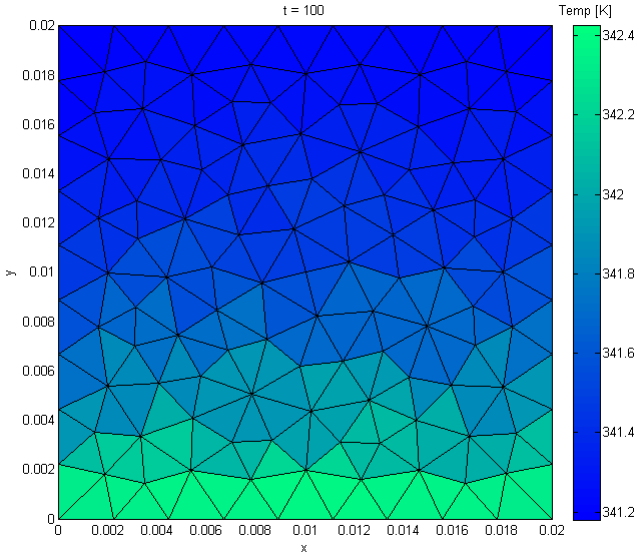

Figure 17: Temperature after 100 time steps

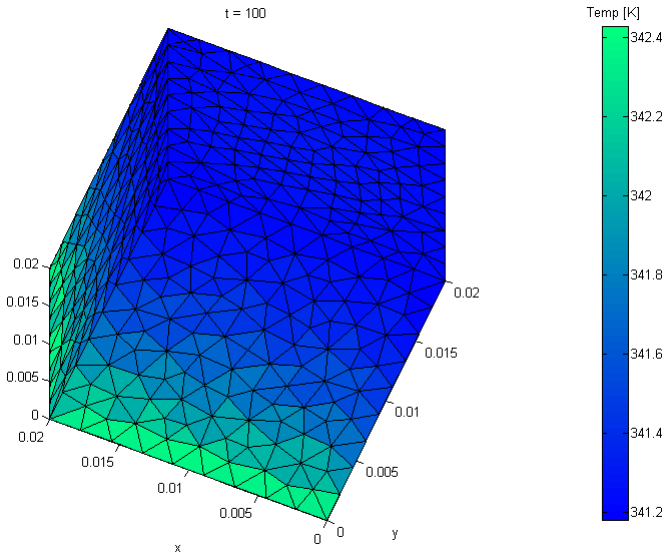

Figure 18: Temperature after 100 time steps, 3D Box perspective

APPENDIX B. C++ sequential simulation on a Box grid with Finite Element Method for linear 3D tetrahedral elements and Implicit Euler Method.

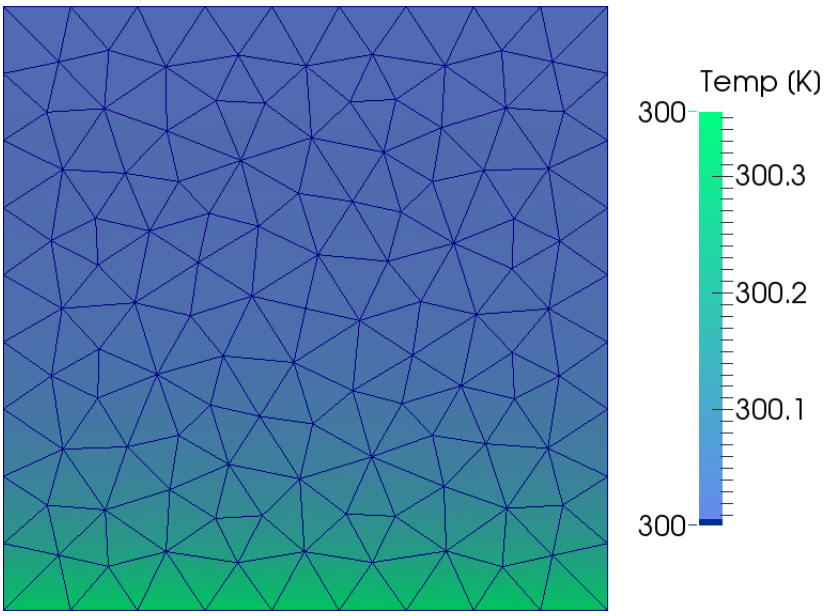

Figure 19: Initial state of Temperature

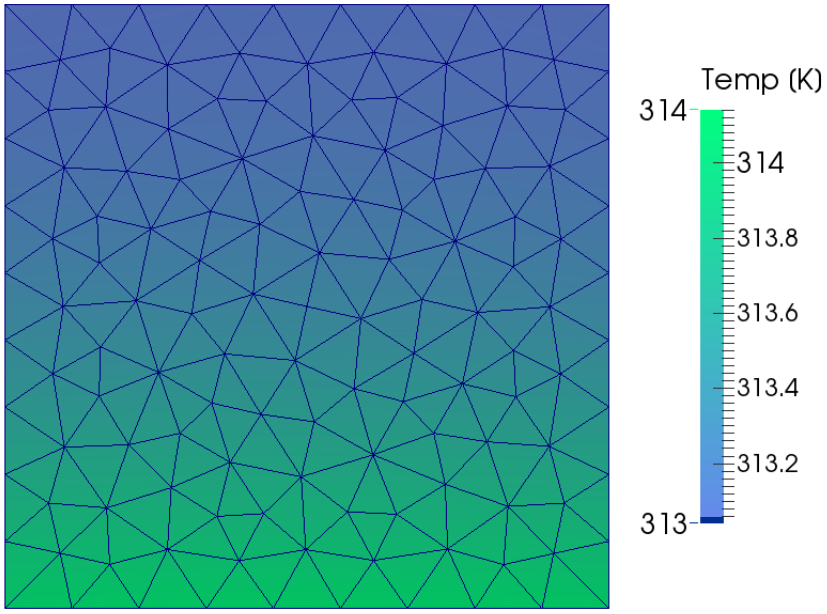

Figure 20: Temperature after 25 time step

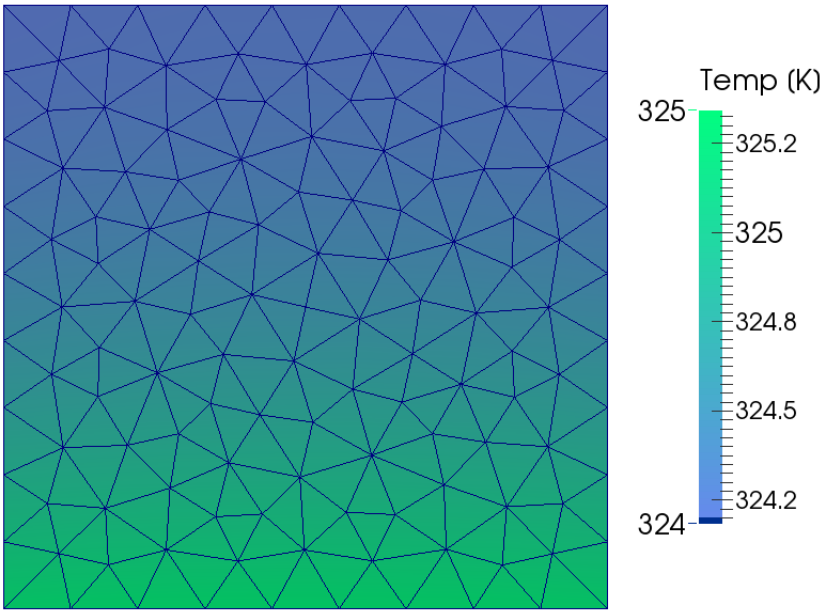

Figure 21: Temperature after 50 time steps

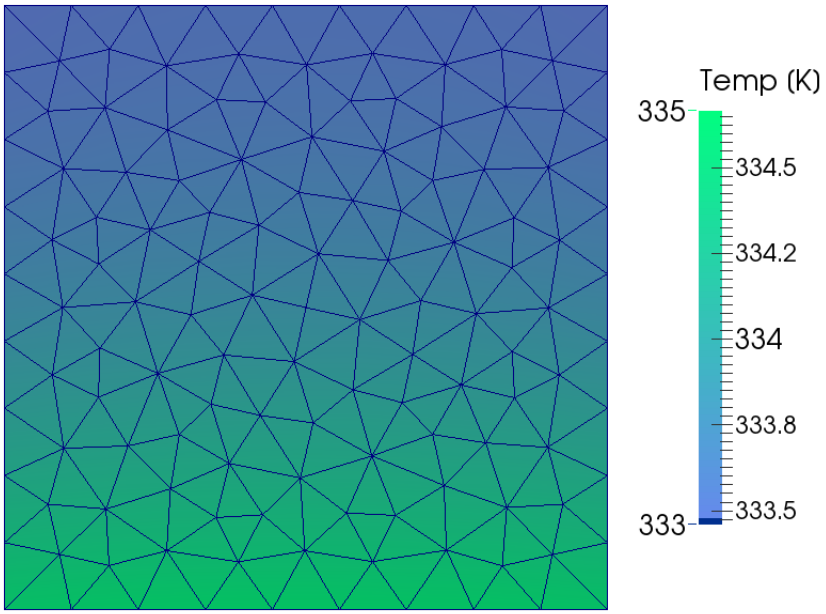

Figure 22: Temperature after 75 time steps

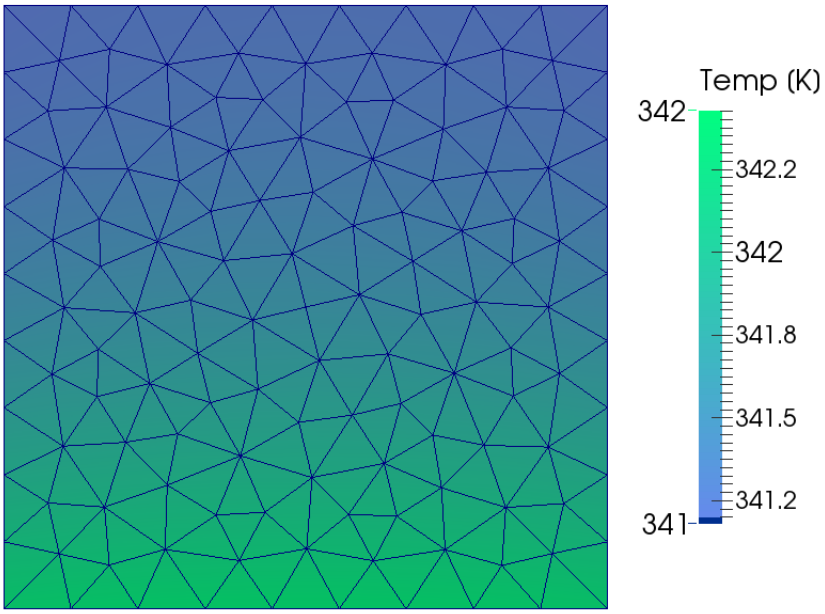

Figure 23: Temperature after 100 time steps

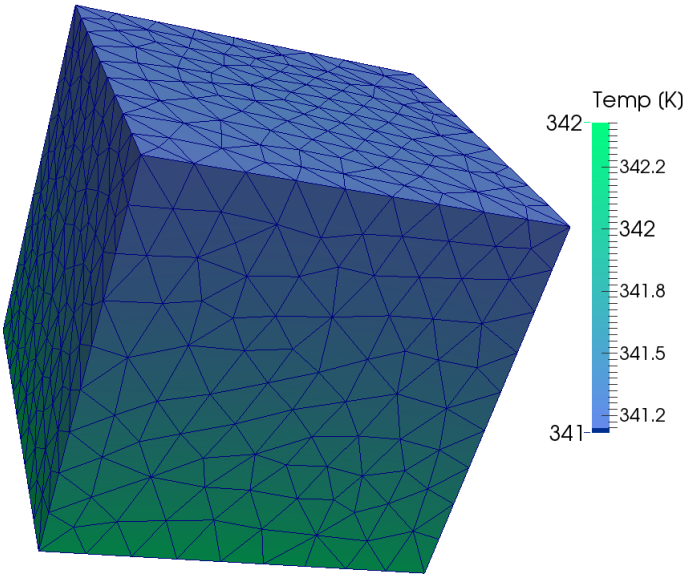

Figure 24: Temperature after 100 time steps, 3D Box perspective

APPENDIX C. c++ Hybrid simulation on a CPU sink grid with Finite Element Method for linear 3D tetrahedral elements and Implicit Euler Method.

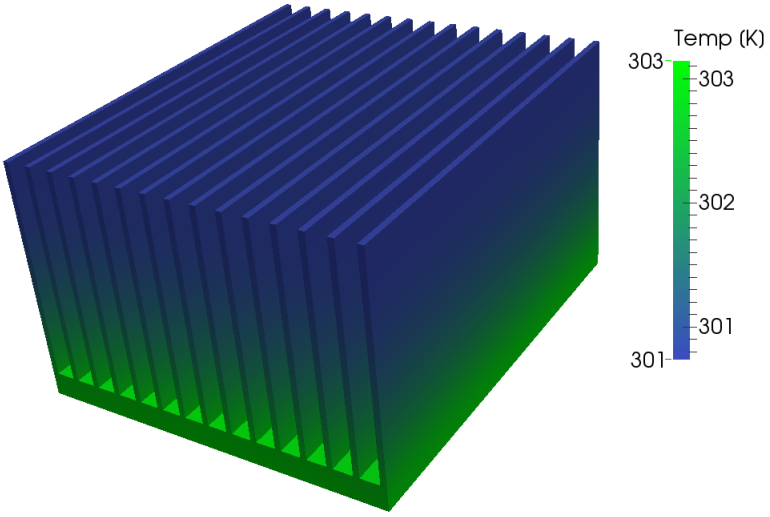

Figure 25: Initial state of Temperature

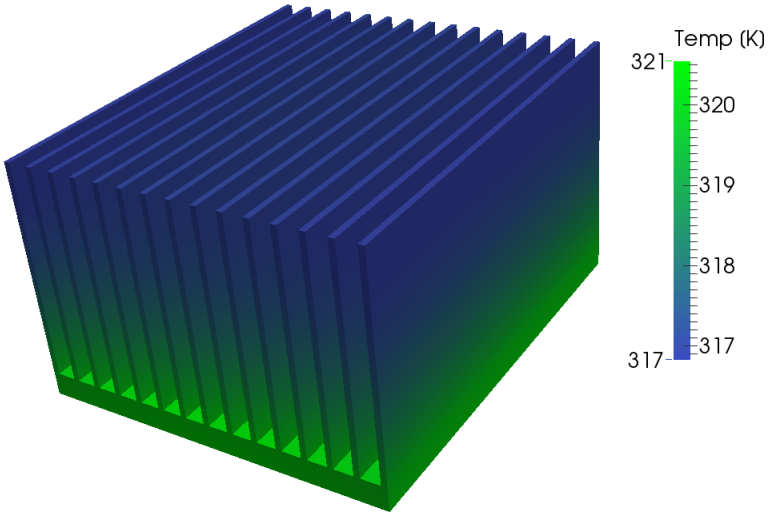

Figure 26: Temperature after 25 time step

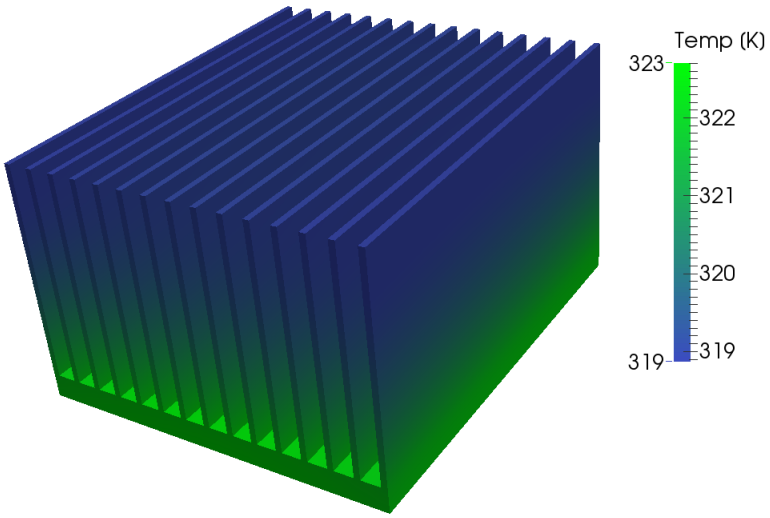

Figure 27: Temperature after 50 time steps

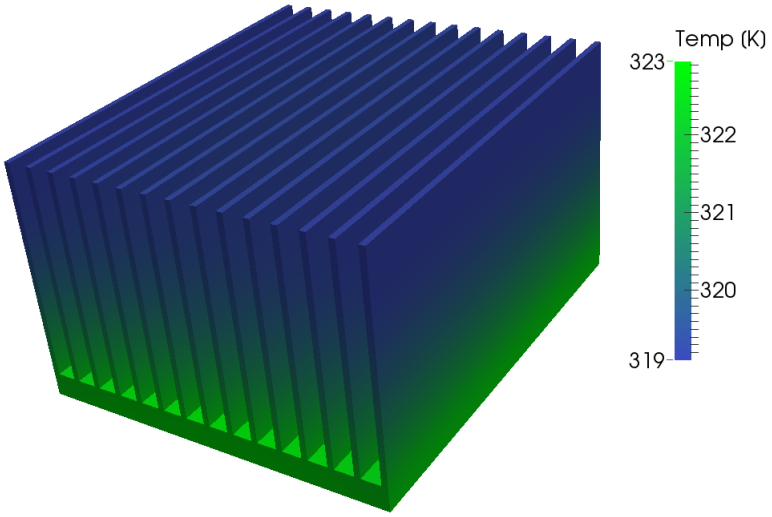

Figure 28: Temperature after 75 time steps

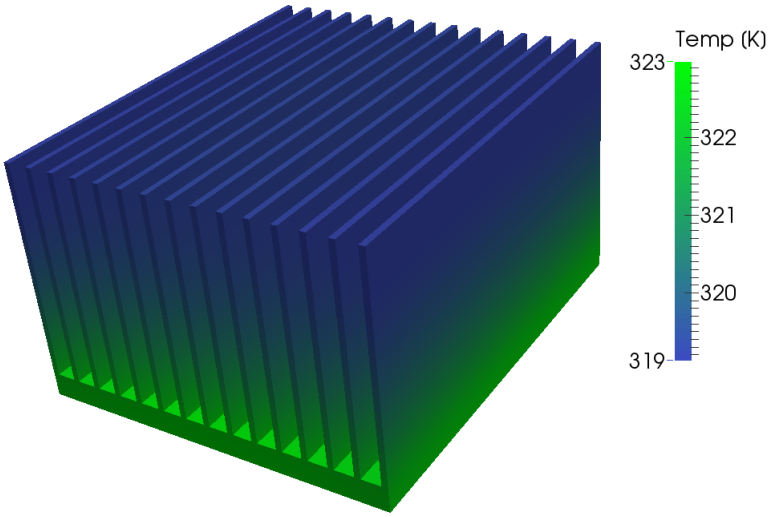

Figure 29: Temperature after 100 time steps

8 References

[1] S. Moore, *Applied numerical methods*, 2014.

[2] P. Wautelet P.Fr. Lavalle, *Hybrid mpi-openmp programming*, 2013, available at `http://www.idris.fr/data/cours/hybride/form_hybride_en_proj.pdf`.

[3] J. Sayas, *A gentle introduction to the finite element method*, preprint (2008), available at `http://www.imati.cnr.it/marini/didattica/Metodi-engl/Intro2FEM.pdf`.