Parallel Computing for Computational Mechanics

Summer Semester 2022

Prof. Marek Behr

Final Project

Parallelization of a FE code using OpenMP and MPI

Submitted by: Sajjala, Sreekar Reddy
sreekar.sajjala@rwth-aachen.de
Matr. Nu.: 415917

Table of Contents

1. ABSTRACT

The project's major goal is to comprehend and see the trends in speed and efficiency of a serial code when it is parallelized using OpenMP and MPI. A Finite Element solver is used to discretize a heat equation characterizing the temperature distribution across a domain shaped in disc. Initially, the problem is solved in serial and then in parallel using OpenMP and MPI. Various optimization flags for the serial solver have been implemented to help reduce the runtime. A combination of the -O2 and -qopt-prefetch was discovered to be the most effective for solving the problem in the shortest amount of time. It is noted that, using parallel techniques resulted in a significant increase in speed and efficiency. The static scheduling with 1024 chunk size was proven to be the fastest configuration after optimizing the scheduling approaches. Changing the mesh density and number of threads resulted in the best thread combination for each type of mesh. Similarly, using an MPI-based technique, the number of cores was changed. The number of cores that resulted in the fastest and most parallelly efficient technique for each type of mesh were recorded and investigated.

2. INTRODUCTION

This report documents the final project for the course "*Parallel Computing for Computational Mechanics*" which is based on Parallelization of a Finite Element (FE) code using OpenMP and MPI. The FE code developed in homework during the courseware was used to solve the temperature distribution on a 2d disk domain. The solver was compiled and executed serially as well as in its parallelized version using OpenMP and MPI. The performance of the individual implementations has been documented and analysed.

The following chapter (-3) guides us through the theoretical background of heat equation, Finite Element Method (FEM), Scalar optimization, OpenMP and MPI. The later chapter (-4) "Implementation and Validation" highlights the snippets of C++ implementation of the governing equations and explains the code used in the project. It also briefs about the case setups for several configurations employed in the project.

Following is chapter-5 "Results and Discussion" where the results from several runs with different compiler flags, thread count, and scheduling configurations are presented with plausible causes for obtained outcomes.

The conclusion contains the summarization of this report with achieved results with possible future steps and improvements in the chapter-6.

3. THEORY AND METHODS

3.1. The Heat Equation

The project is based on the fundamental heat equation defined for a 2D circular (Disc) domain Ω with boundary Γ as represented in Figure 1.

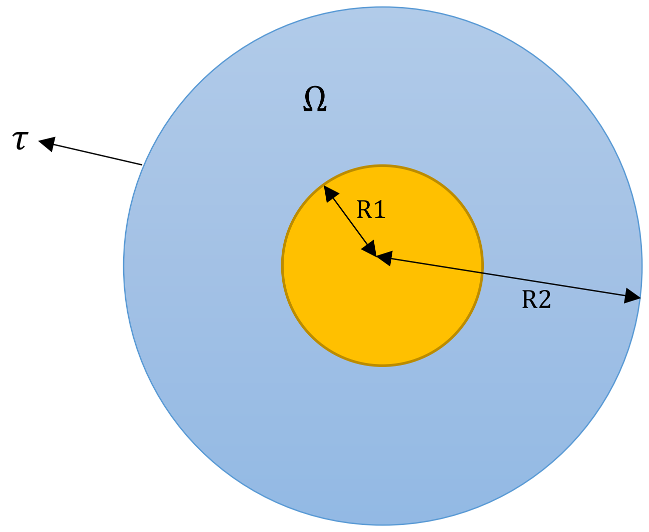


Figure 1: Domain of the heat problem

The heat equation along with its boundary conditions are given by the following set of equations:

$$\frac{\partial T}{\partial t} - \kappa \nabla T = f \qquad on\ \Omega \quad \forall f \in (0, t_f)$$

$$T(x,t) = T_D \qquad on\ \Gamma \quad \forall f \in (0, t_f)$$

$$T(x,0) = T_0 \qquad on\ \Omega$$

Where T is the Temperature, κ is the Thermal Diffusivity and f is the Thermal Heat Source. The boundary conditions are specified as Dirichlet boundary conditions, T_D. The equation is solved to find out the temperature at any time t ∈ $(0, t_f)$ given the initial temperature T_0 and the time at which the solution must be found.

The heat source changes based on the location of the node within the radius R_1:

$$f(r) = \begin{cases} \dfrac{Q}{\pi {R_1}^2} & if\ r < R_1 \\ 0 & if\ r \geq R_1 \end{cases}$$

Where Q = P/d_z, a volumetric heat source. d_z = 0.1m is the out-of-plane thickness of the domain.

The analytical solution for the equation as $t\ \rightarrow \infty$ is given by:

$$T(r) = \begin{cases} T_0 - \frac{Q}{2\pi\alpha}\left(\frac{1}{2}\left(\frac{r^2}{R_1^2} - 1\right) + \ln\left(\frac{R_1}{R_2}\right)\right) & if\ r < R_1 \\ T_0 - \frac{Q}{2\pi\alpha}\left(\ln\left(\frac{r}{R_2}\right)\right) & if\ r \geq R_1 \end{cases}$$

where, α and β are user specified parameters. Table 1 shows the important parameters of the domain used in this project.

Table 1: Problem parameters for the heated disk problem

Parameter	*Variable*	*Value*	*Unit*
Inner circle radius	R_1	0.01	[m]
Outer circle radius	R_2	0.1	[m]
Heat source on the area	Q	100	[W]
Dirichlet BC temperature	$T_0(x,y)$	500	[K]
Initial temperature	$T_s(x,y,0)$	0	[K]
Thermal diffusivity	κ	1.0	$[m^2/s]$

3.2. Finite Element Method

The finite element method (FEM) is a numerical technique for solving a wide range of physical phenomena that are often encountered in the physical and engineering sciences. These problems can be structural in nature, thermal (or thermo-mechanical), electrical, magnetic, acoustic etc. plus any combination of it. It is used most frequently to tackle problems that aren’t readily amenable to analytical treatments.

In FEM, the solution is obtained by reformulating the differential equations as a variational formulation using suitable test (shape) functions, followed by discretizing the domain and applying the necessary initial conditions and boundary conditions. The procedure that was followed for this project as discussed in the lecture is given below in a stepwise manner.

a. Pre-processing
    i. Mesh generation
    ii. Initial conditions
b. Time Step Loop
    i. Nonlinear iteration loop
        i. Formulation of Linear System
        ii. Solution of Linear System
    ii. End Loop
c. End Loop
d. Post-processing
    i. Data output
    ii. Data Visualisation

In this project 3 meshes based on the density of the elements: coarse, medium, and fine (the element density increasing from coarse to fine) were used. The initial and boundary conditions are applied based on the equations given in the section 3.1. The system of equations will be solved until the residuals (error) is below a certain value (say 1e-8).

3.3. Scalar Optimization

Generally, the FEM codes require thousands of iterations to converge, thus the runtime is quite high. One of the methods to reduce the runtime for a single thread operation is called as Scalar Optimization which uses single or set of compiler flags which offer different optimization features (such as -O1, -O2, -unroll , -qopt-prefetch etc.).

3.4. OpenMP

OpenMP is a set of compiler directives, library routines, and environmental variables as well as an API that provides support for parallel programming in shared-memory environments. OpenMP identifies parallel regions as blocks of code that may run in multiple threads. The central entity in an OpenMP program is not a process but a thread, also called "lightweight processes", because several of them can share a common address space and mutually access data. It creates as many threads as possible which are processing elements in the system.

Parallel execution happens inside parallel regions, where a set of threads executes instruction streams concurrently. The number of threads in a set may vary among parallel regions. The effect of changing the number of threads in an OpenMP execution of the Finite Element solver is investigated in this study.

OpenMP provides several additional directives for running code regions in parallel, including parallelizing loop. The project shows how the loop runtime is affected by different scheduling arguments and chunk sizes.

3.5. MPI

Message Passing Interface (MPI) is a communication protocol for parallel programming. MPI is specifically used to allow applications to run in parallel across several separate computers connected by a network. We carry out the experiment of studying the influence of the number of cores on the performance of the code.

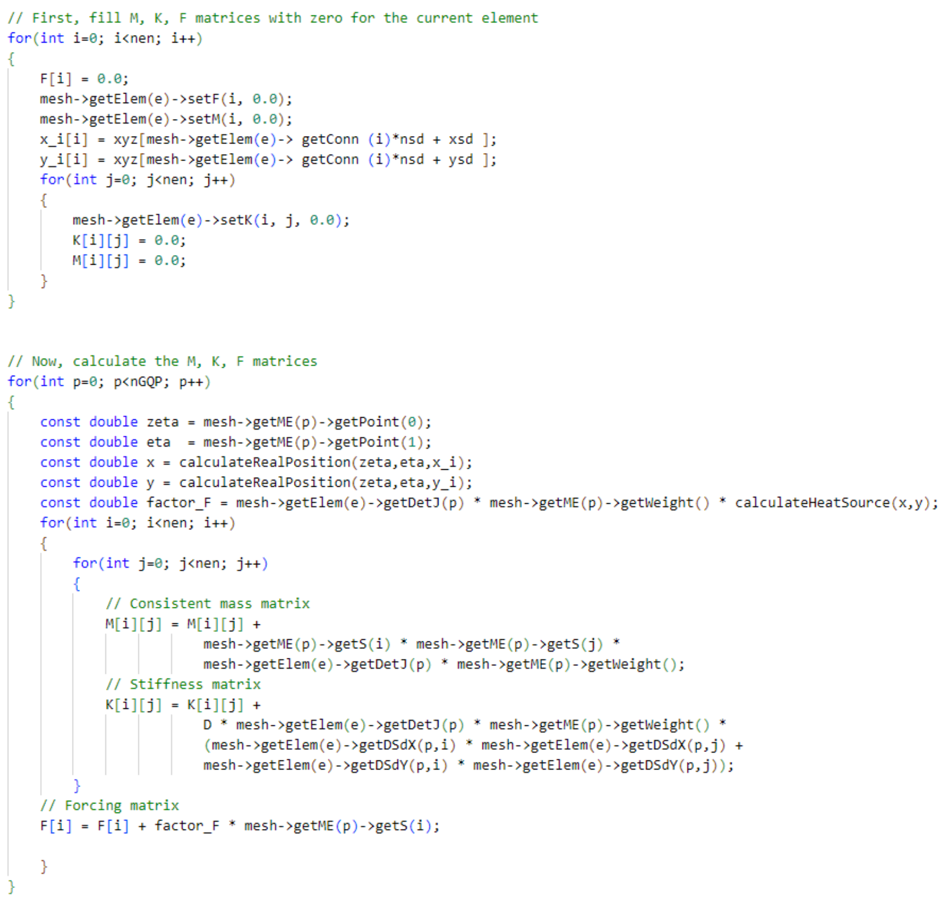
4. IMPLEMENTATION AND VALIDATION

4.1. Discretization

As stated earlier, 3 types of meshes based on the element density in each mesh type were used: coarse, medium, and fine. The meshes were discretized using Gmsh and exported in the MIXD format (minf, mxyz, mrng and mien files) which provide information about the number of elements, nodes, spatial coordinates, connectivity, and dimensions. This information is vital for solving the FEM problem and for post-processing.

4.2. Solving the Heat Equation

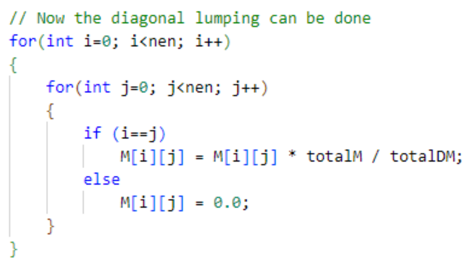
The equations are discretized in accordance to the mesh; the element level matrices are calculated by determining the weights, followed by initializing them and finally mapping the contribution back (analogous to assembly of global stiffness matrix) as shown in Figure 2.



```
// First, fill M, K, F matrices with zero for the current element
for(int i=0; i<nen; i++)
{
    F[i] = 0.0;
    mesh->getElem(e)->setF(i, 0.0);
    mesh->getElem(e)->setM(i, 0.0);
    x_i[i] = xyz[mesh->getElem(e)-> getConn (i)*nsd + xsd ];
    y_i[i] = xyz[mesh->getElem(e)-> getConn (i)*nsd + ysd ];
    for(int j=0; j<nen; j++)
    {
        mesh->getElem(e)->setK(i, j, 0.0);
        K[i][j] = 0.0;
        M[i][j] = 0.0;
    }
}


// Now, calculate the M, K, F matrices
for(int p=0; p<nGQP; p++)
{
    const double zeta = mesh->getME(p)->getPoint(0);
    const double eta  = mesh->getME(p)->getPoint(1);
    const double x = calculateRealPosition(zeta,eta,x_i);
    const double y = calculateRealPosition(zeta,eta,y_i);
    const double factor_F = mesh->getElem(e)->getDetJ(p) * mesh->getME(p)->getWeight() * calculateHeatSource(x,y);
    for(int i=0; i<nen; i++)
    {
        for(int j=0; j<nen; j++)
        {
            // Consistent mass matrix
            M[i][j] = M[i][j] +
                        mesh->getME(p)->getS(i) * mesh->getME(p)->getS(j) *
                        mesh->getElem(e)->getDetJ(p) * mesh->getME(p)->getWeight();
            // Stiffness matrix
            K[i][j] = K[i][j] +
                        D * mesh->getElem(e)->getDetJ(p) * mesh->getME(p)->getWeight() *
                        (mesh->getElem(e)->getDSdX(p,i) * mesh->getElem(e)->getDSdX(p,j) +
                        mesh->getElem(e)->getDSdY(p,i) * mesh->getElem(e)->getDSdY(p,j));
        }
    // Forcing matrix
    F[i] = F[i] + factor_F * mesh->getME(p)->getS(i);

    }
}
```

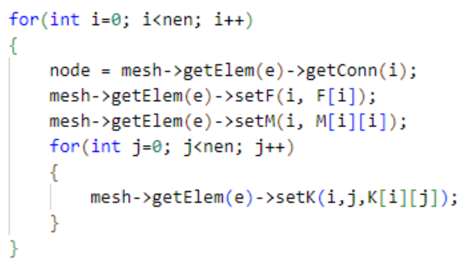
Figure 2: Initializing and calculating the element level matrices

This step is followed by mass lumping the mass matrix and then mapping them to the global elements as shown in Figure 3 and 4 respectively.



```
// Now the diagonal lumping can be done
for(int i=0; i<nen; i++)
{
    for(int j=0; j<nen; j++)
    {
        if (i==j)
            M[i][j] = M[i][j] * totalM / totalDM;
        else
            M[i][j] = 0.0;
    }
}
```

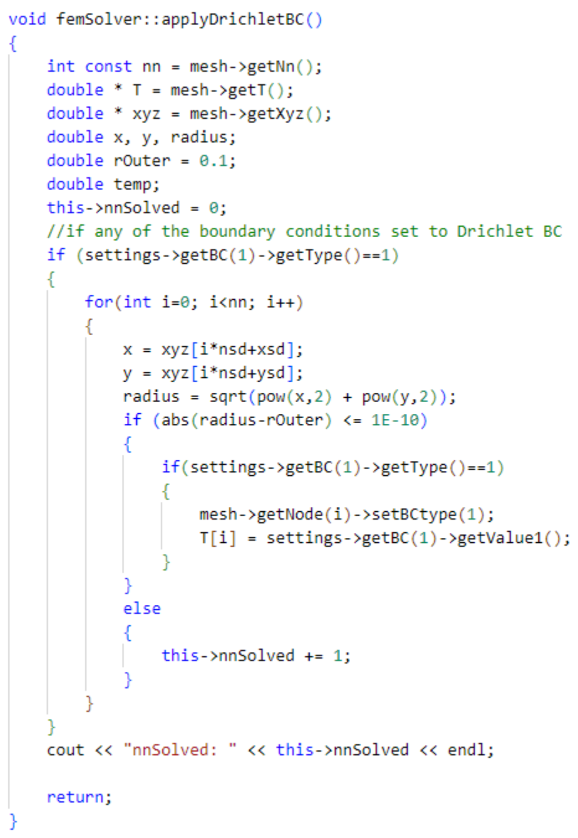
Figure 3: Mass lumping



```
for(int i=0; i<nen; i++)
{
    node = mesh->getElem(e)->getConn(i);
    mesh->getElem(e)->setF(i, F[i]);
    mesh->getElem(e)->setM(i, M[i][i]);
    for(int j=0; j<nen; j++)
    {
        mesh->getElem(e)->setK(i,j,K[i][j]);
    }
}
```

Figure 4: Global Matrices Mapping

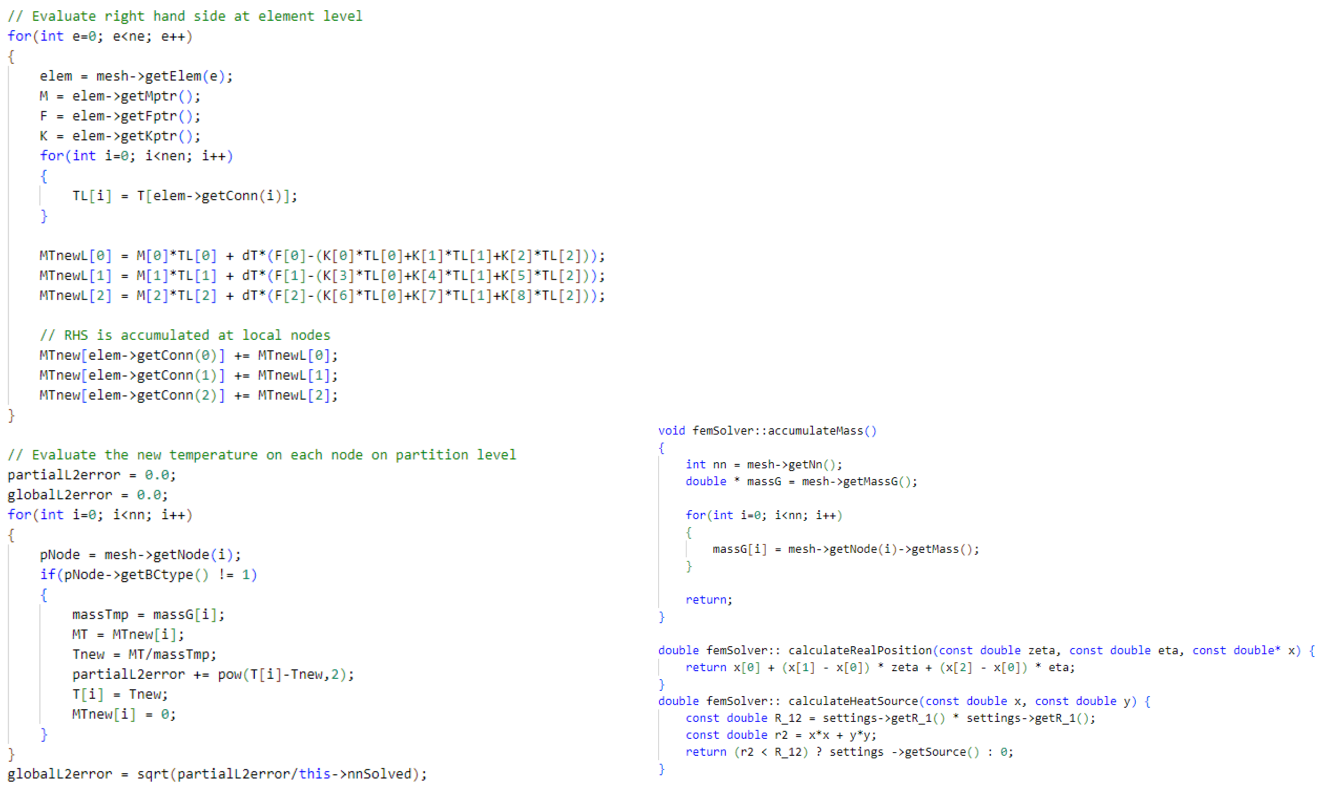
Now we move on to the application of the Dirichlet Boundary Conditions. If any points are found on the boundary, the code replaces them with the specified value according to the boundary conditions as depicted in Figure 5.



```cpp
void femSolver::applyDrichletBC()
{
    int const nn = mesh->getNn();
    double * T = mesh->getT();
    double * xyz = mesh->getXyz();
    double x, y, radius;
    double rOuter = 0.1;
    double temp;
    this->nnSolved = 0;
    //if any of the boundary conditions set to Drichlet BC
    if (settings->getBC(1)->getType()==1)
    {
        for(int i=0; i<nn; i++)
        {
            x = xyz[i*nsd+xsd];
            y = xyz[i*nsd+ysd];
            radius = sqrt(pow(x,2) + pow(y,2));
            if (abs(radius-rOuter) <= 1E-10)
            {
                if(settings->getBC(1)->getType()==1)
                {
                    mesh->getNode(i)->setBCtype(1);
                    T[i] = settings->getBC(1)->getValue1();
                }
            }
            else
            {
                this->nnSolved += 1;
            }
        }
    }
    cout << "nnSolved: " << this->nnSolved << endl;

    return;
}
```

Figure 5: Dirichlet Boundary Conditions' Application

Now we proceed to the last step (Figure 6) where we evaluate the matrix equation by solving the LHS of the matrix equation and then inverting the lumped mass matrix which was calculated before (Figure 3) at each nodal position. In addition, the difference between the

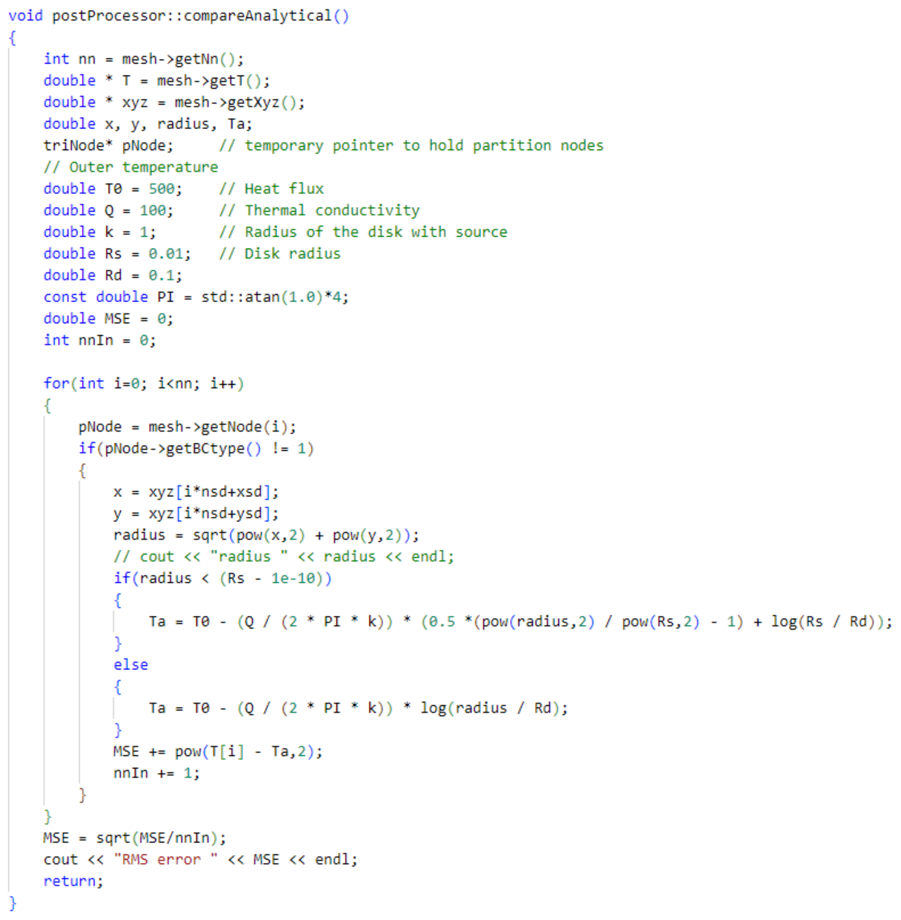


```cpp
// Evaluate right hand side at element level
for(int e=0; e<ne; e++)
{
    elem = mesh->getElem(e);
    M = elem->getMptr();
    F = elem->getFptr();
    K = elem->getKptr();
    for(int i=0; i<nen; i++)
    {
        TL[i] = T[elem->getConn(i)];
    }

    MTnewL[0] = M[0]*TL[0] + dT*(F[0]-(K[0]*TL[0]+K[1]*TL[1]+K[2]*TL[2]));
    MTnewL[1] = M[1]*TL[1] + dT*(F[1]-(K[3]*TL[0]+K[4]*TL[1]+K[5]*TL[2]));
    MTnewL[2] = M[2]*TL[2] + dT*(F[2]-(K[6]*TL[0]+K[7]*TL[1]+K[8]*TL[2]));

    // RHS is accumulated at local nodes
    MTnew[elem->getConn(0)] += MTnewL[0];
    MTnew[elem->getConn(1)] += MTnewL[1];
    MTnew[elem->getConn(2)] += MTnewL[2];
}

// Evaluate the new temperature on each node on partition level
partialL2error = 0.0;
globalL2error = 0.0;
for(int i=0; i<nn; i++)
{
    pNode = mesh->getNode(i);
    if(pNode->getBCtype() != 1)
    {
        massTmp = massG[i];
        MT = MTnew[i];
        Tnew = MT/massTmp;
        partialL2error += pow(T[i]-Tnew,2);
        T[i] = Tnew;
        MTnew[i] = 0;
    }
}
globalL2error = sqrt(partialL2error/this->nnSolved);
```

```cpp
void femSolver::accumulateMass()
{
    int nn = mesh->getNn();
    double * massG = mesh->getMassG();

    for(int i=0; i<nn; i++)
    {
        massG[i] = mesh->getNode(i)->getMass();
    }

    return;
}

double femSolver:: calculateRealPosition(const double zeta, const double eta, const double* x) {
    return x[0] + (x[1] - x[0]) * zeta + (x[2] - x[0]) * eta;
}
double femSolver:: calculateHeatSource(const double x, const double y) {
    const double R_12 = settings->getR_1() * settings->getR_1();
    const double r2 = x*x + y*y;
    return (r2 < R_12) ? settings ->getSource() : 0;
}
```

Figure 6: Calculating the RHS of matrix equation

global and local solutions is determined and utilized as a convergence criterion for the loop (Figure 7).



```
void postProcessor::compareAnalytical()
{
    int nn = mesh->getNn();
    double * T = mesh->getT();
    double * xyz = mesh->getXyz();
    double x, y, radius, Ta;
    triNode* pNode;     // temporary pointer to hold partition nodes
    // Outer temperature
    double T0 = 500;    // Heat flux
    double Q = 100;     // Thermal conductivity
    double k = 1;       // Radius of the disk with source
    double Rs = 0.01;   // Disk radius
    double Rd = 0.1;
    const double PI = std::atan(1.0)*4;
    double MSE = 0;
    int nnIn = 0;

    for(int i=0; i<nn; i++)
    {
        pNode = mesh->getNode(i);
        if(pNode->getBCtype() != 1)
        {
            x = xyz[i*nsd+xsd];
            y = xyz[i*nsd+ysd];
            radius = sqrt(pow(x,2) + pow(y,2));
            // cout << "radius " << radius << endl;
            if(radius < (Rs - 1e-10))
            {
                Ta = T0 - (Q / (2 * PI * k)) * (0.5 *(pow(radius,2) / pow(Rs,2) - 1) + log(Rs / Rd));
            }
            else
            {
                Ta = T0 - (Q / (2 * PI * k)) * log(radius / Rd);
            }
            MSE += pow(T[i] - Ta,2);
            nnIn += 1;
        }
    }
    MSE = sqrt(MSE/nnIn);
    cout << "RMS error " << MSE << endl;
    return;
}
```

Figure 7: Solution Comparison

4.3. Compiling the Solver

The solver described above has been updated to accommodate OpenMP and MPI directives. The two most time-consuming loops (evaluating RHS and new temperatures at each node) are parallelized by the *<#pragma omp>* directive and the reduction clause, as well as specific scheduling algorithms, in OpenMP. The mesh is subdivided into nodes-or-elements per processor in the MPI implementation. The commands in Figure 8 are used to compile these three solvers for the three mesh densities.

```
mkdir build                              # Create new directory to compile code
cd build                                 # navigate into build directory
cmake .. —DBUILD_SELECTOR=<build>        # Execute CMake to generate the make setup
make                                     # compile the actual code using make
```

Where < build > is replaced by the solver name as required i.e., Serial, OpenMP_Task_A, OpenMP_Task_B and MPI. The then formed executable "2D_Unsteady_<build> is used along with the run. <build>j files to run the simulation. The output files are saved in the directory and can be read directly or used for visualization on Paraview.

5. RESULTS AND DISCUSSIONS

5.1. Serial Solver

To optimize the code to run in serial, various compiler flags were employed running with the medium mesh quality. The timings obtained for the same and the compiler descriptions are mentioned in Table 2. All the compiler optimization flags are run more than once and averaged the time for better comparison. The combination of -O2 -qopt-prefetch was the most time efficient; clocking 54.59 seconds.

Table 2: Compiler Flags, timings, and descriptions for Serial Solver | Medium Mesh

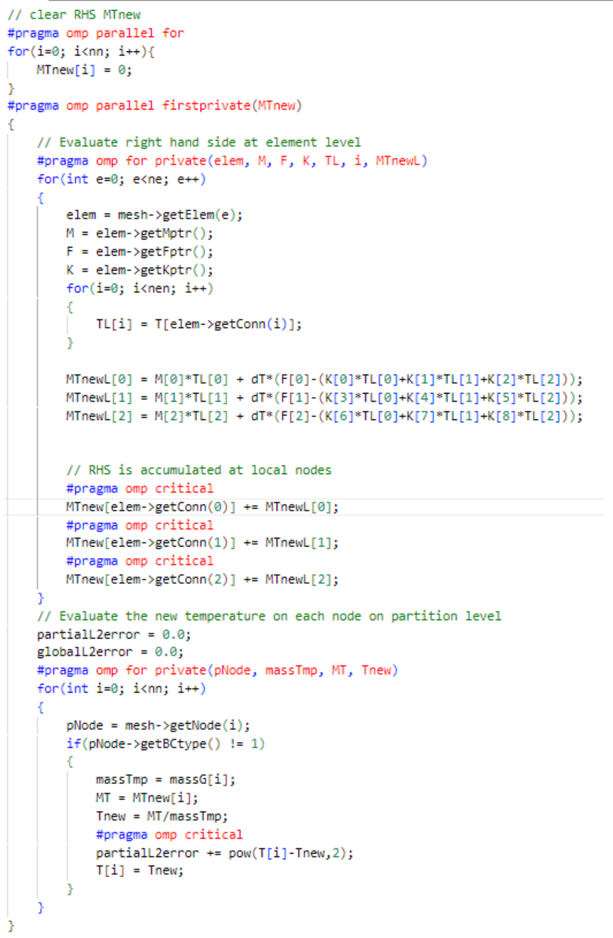
Compiler Flag	Runtime (seconds)	Flag Description
-O0	227.52	No optimization
-O1	62.78	Minimum optimization to reduce size of the code (no inlining)
-O1-fno-alias	64.69	Features of O1 and assume no aliasing in the program.
-O1-ip	62.66	Features of O1 with Inter-procedural optimizations, including selective inlining, within the current source file.
-O1-ipo	64.55	Features of O1 with Inter-procedural optimizations across multiple source files
-O1-qopt-prefetch	61.65	Features of O1 with prefetching
-O1-scalar-rep	63.235	Features of O1 with scalar replacement
-O1-unroll	62.24	Features of O1 with loop unrolling
-O2	55.27	Basic optimization including global common expression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, global register allocation, basic block merging, loop unrolling, software pipelining, tail recursion elimination, tail call elimination.
-O2-lmkl	57.08	Features of O2 plus link with optimized BLAS
-O2-qopt-prefetch	**54.59**	Features of O2 with data prefetching
-O2-scalar-rep	55.51	Features of O1 with scalar replacement
-O3	57.95	Performs -O2 optimisations and enables more aggressive loop transformations.
-O3-ipo	55.585	Features of O3 with Inter-procedural optimizations across source files
-O3-qopt-prefetch	56.225	Features of O3 with prefetching
-Ofast	56.05	Features certain aggressive options to improve the speed of application. It sets *-O3 -no-prec-div -fp-model fast=2*

When compiler flags are used, the code is considerably re-organized into assembly language. This would make it difficult to use a debugger properly. As a result, a non-optimized code is more advantageous to the user in the early stages of code development. The compiler has limited understanding of data dependencies in automatic optimization, notably across function borders. Use of unrestricted pointer arithmetic in C, makes it tough, and it can also slow down compilation. As the user has the most information about data dependencies, user-defined optimization can be beneficial. It can make code less readable, yet the same optimization can be helpful to one architecture while being harmful to another. Because optimization is usually costly, the code may take longer to build with optimizations enabled than it would without. In general, shorter compile times are advantageous. Another reason to avoid optimizations is because the compiler may have bugs that only exist when optimization is being performed. These bugs can be avoided by compiling without optimization.

5.2. Parallel OpenMP Solver

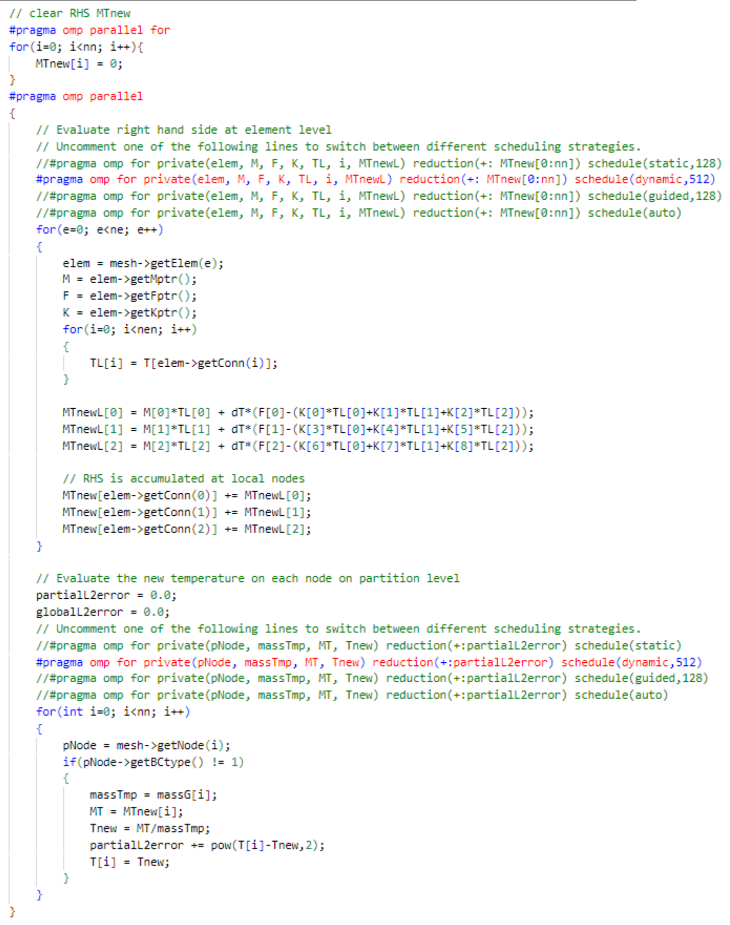
Table 3: Runtime Table for OpenMP Task A and B | 1, 2, and 4 thread(s) |coarse mesh

Threads	Runtime - OpenMP Task A (seconds)	Runtime - OpenMP Task B (seconds)
1	7.95747	1.24535
2	72.98202	1.48758
4	142.60266	1.25867



```
// clear RHS MTnew
#pragma omp parallel for
for(i=0; i<nn; i++){
    MTnew[i] = 0;
}
#pragma omp parallel firstprivate(MTnew)
{
    // Evaluate right hand side at element level
    #pragma omp for private(elem, M, F, K, TL, i, MTnewL)
    for(int e=0; e<ne; e++)
    {
        elem = mesh->getElem(e);
        M = elem->getMptr();
        F = elem->getFptr();
        K = elem->getKptr();
        for(i=0; i<nen; i++)
        {
            TL[i] = T[elem->getConn(i)];
        }

        MTnewL[0] = M[0]*TL[0] + dT*(F[0]-(K[0]*TL[0]+K[1]*TL[1]+K[2]*TL[2]));
        MTnewL[1] = M[1]*TL[1] + dT*(F[1]-(K[3]*TL[0]+K[4]*TL[1]+K[5]*TL[2]));
        MTnewL[2] = M[2]*TL[2] + dT*(F[2]-(K[6]*TL[0]+K[7]*TL[1]+K[8]*TL[2]));


        // RHS is accumulated at local nodes
        #pragma omp critical
        MTnew[elem->getConn(0)] += MTnewL[0];
        #pragma omp critical
        MTnew[elem->getConn(1)] += MTnewL[1];
        #pragma omp critical
        MTnew[elem->getConn(2)] += MTnewL[2];
    }
    // Evaluate the new temperature on each node on partition level
    partialL2error = 0.0;
    globalL2error = 0.0;
    #pragma omp for private(pNode, massTmp, MT, Tnew)
    for(int i=0; i<nn; i++)
    {
        pNode = mesh->getNode(i);
        if(pNode->getBCtype() != 1)
        {
            massTmp = massG[i];
            MT = MTnew[i];
            Tnew = MT/massTmp;
            #pragma omp critical
            partialL2error += pow(T[i]-Tnew,2);
            T[i] = Tnew;
        }
    }
}
```

Figure 9: Use of Critical Regions Open MP Task A



```
// clear RHS MTnew
#pragma omp parallel for
for(i=0; i<nn; i++){
    MTnew[i] = 0;
}
#pragma omp parallel
{
    // Evaluate right hand side at element level
    // Uncomment one of the following lines to switch between different scheduling strategies.
    //#pragma omp for private(elem, M, F, K, TL, i, MTnewL) reduction(+: MTnew[0:nn]) schedule(static,128)
    #pragma omp for private(elem, M, F, K, TL, i, MTnewL) reduction(+: MTnew[0:nn]) schedule(dynamic,512)
    //#pragma omp for private(elem, M, F, K, TL, i, MTnewL) reduction(+: MTnew[0:nn]) schedule(guided,128)
    //#pragma omp for private(elem, M, F, K, TL, i, MTnewL) reduction(+: MTnew[0:nn]) schedule(auto)
    for(e=0; e<ne; e++)
    {
        elem = mesh->getElem(e);
        M = elem->getMptr();
        F = elem->getFptr();
        K = elem->getKptr();
        for(i=0; i<nen; i++)
        {
            TL[i] = T[elem->getConn(i)];
        }

        MTnewL[0] = M[0]*TL[0] + dT*(F[0]-(K[0]*TL[0]+K[1]*TL[1]+K[2]*TL[2]));
        MTnewL[1] = M[1]*TL[1] + dT*(F[1]-(K[3]*TL[0]+K[4]*TL[1]+K[5]*TL[2]));
        MTnewL[2] = M[2]*TL[2] + dT*(F[2]-(K[6]*TL[0]+K[7]*TL[1]+K[8]*TL[2]));

        // RHS is accumulated at local nodes
        MTnew[elem->getConn(0)] += MTnewL[0];
        MTnew[elem->getConn(1)] += MTnewL[1];
        MTnew[elem->getConn(2)] += MTnewL[2];
    }

    // Evaluate the new temperature on each node on partition level
    partialL2error = 0.0;
    globalL2error = 0.0;
    // Uncomment one of the following lines to switch between different scheduling strategies.
    //#pragma omp for private(pNode, massTmp, MT, Tnew) reduction(+:partialL2error) schedule(static)
    #pragma omp for private(pNode, massTmp, MT, Tnew) reduction(+:partialL2error) schedule(dynamic,512)
    //#pragma omp for private(pNode, massTmp, MT, Tnew) reduction(+:partialL2error) schedule(guided,128)
    //#pragma omp for private(pNode, massTmp, MT, Tnew) reduction(+:partialL2error) schedule(auto)
    for(int i=0; i<nn; i++)
    {
        pNode = mesh->getNode(i);
        if(pNode->getBCtype() != 1)
        {
            massTmp = massG[i];
            MT = MTnew[i];
            Tnew = MT/massTmp;
            partialL2error += pow(T[i]-Tnew,2);
            T[i] = Tnew;
        }
    }
}
```

Figure 10: Use Reduction Clause in Open MP Task B

As indicated in the Figures 9 and 10, to avoid race condition OpenMP Task-A employs the critical directive, while OpenMP Task-B uses the reduction clause. Concurrent write access to shared variables (in our case in point, MTnew and PartialL2error) must be avoided to not have a race circumstance. This problem is solved with critical areas, which ensure that only one thread at a time performs a specific snippet of code. If a thread is running code within the critical zone, the others must wait until the first exit.
The reduction clause, on the other hand, privatizes the supplied variable and initializes the private instances with a logical starting value to collect numerous contributions into a single variable. To obtain the result, all partial outcomes are accumulated into a shared instance of the variable at the end of the construct using the provided operator. This permits threads to conduct loop operations in true parallel, with no need for one thread to wait for the other to finish. OpenMP Task-B is substantially faster than OpenMP Task-A, as expected. Because the threads operate in the critical region one at a time, the runtime for OpenMP Task-A increases with the increase in the number of threads. Threads can perform the parallel operation with OpenMP Task-B, and so the runtime decreases as the number of threads increases. The same can be observed from the Table-3 where the OpenMP Task-B performs significantly faster than Task-A.

Table 4: Runtime for different Scheduling options and chunk sizes for OpenMP Task B |4 threads | fine mesh

Chunk size	Runtime for static (seconds)	Runtime for dynamic (seconds)	Runtime for guided (seconds)	Runtime for Auto (seconds)
32	211.53719	339.33645	6.85618*	31.62000*
64	196.96788	278.15170	85.69063*	
128	199.46252	201.01179	95.75506*	
256	184.42301	62.95483*	10.28273*	
512	188.85252	19.72466*	10.46302*	
1024	183.96627	32.61024*	120.25798*	

In OpenMP, the default chunk size for static, dynamic, and guided is 1. The dynamic scheduling method is used by default. Static Scheduling is clearly the fastest of all the alternatives. Static divides the loop into equal-sized chunks, allowing each thread to execute on exactly one of them. Dynamic Scheduling assigns a portion of work to the next thread that has completed its current chunk, whose size is determined by the chunk size. Threads assigned to "easier" chunks will complete more of them using this strategy, and load imbalance will be considerably decreased. The threads request additional chunks dynamically in the guided schedule, but the chunk size is always proportional to the remaining number of iterations divided by the number of threads.

Table 5 shows the OpenMP Task B runtime on various meshes for various thread counts. The scheduling option is kept at static with chunk size 1024 as the same was observed to be the most efficient from the Table-4.[1]

* - Encountered issues while running the case. The solver stopping early; couldn't converge to 1e-8.

Table 5: Runtime for OpenMP Task B on various meshes and threads

Number of Threads	Runtime on coarse mesh (seconds)	Runtime on medium mesh (seconds)	Runtime on fine mesh (seconds)
1	1.24535	62.60195	666.6673
2	1.48758	39.70895	401.62145
4	1.25867	32.37038	188.59100
6	1.24845	30.99017	152.60274
8	1.18140	26.34636	129.28447
10	1.14660	25.29513	127.72116
12	1.08441	22.89489	112.02573

Serial runs were performed at all the mesh sized to compute the speedup. The fastest time clocked by the serial solver is 1.02, 54.59, and 643.69 seconds for coarse, medium, and fine meshes respectively. Figure 11 displays the speedup attained for the three meshes used for OpenMP task B. The speedup is given by the formula:

$$S = \frac{Time\ for\ fastest\ serial\ execution\ (T_s)}{Time\ for\ parallel\ execution\ (T_p)}$$

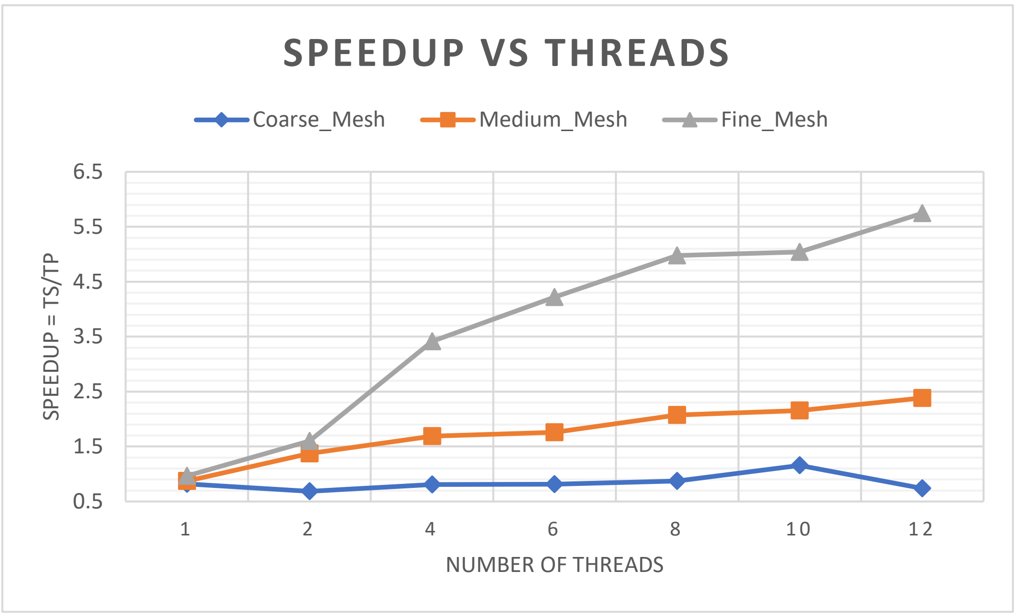


Figure 11: Speedup Attained for OpenMP Task B

Similarly Figure 12 represents the parallel efficiency achieved. The parallel efficiency is given by:

$$E = \frac{S(speedup)}{Number\ of\ threads\ used\ (p)}$$

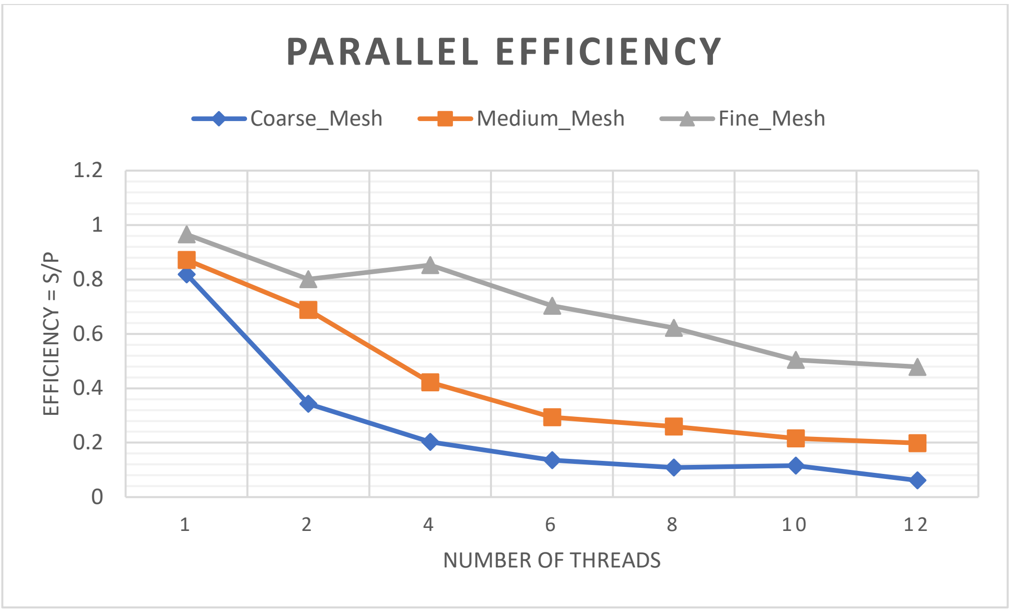


Figure 12: Parallel Efficiency Attained for OpenMP Task B

The speedup increases for the fine mesh as the number of threads used grows, whereas it peaks at 8 and 12 threads for the medium and fine meshes, respectively. With increasing threads, parallel efficiency declines for all the three meshes. For the coarse mesh there is a drastic drop in efficiency from 1 to 2 thread execution. As shown by this study the greatest speedup does not always mean the greatest efficiency. As they can have a high level of subjectivity, one must first grasp the code's requirements before employing the appropriate techniques. The finer the mesh, the greater the speedup and efficiency that may be attained, is a commonly seen factor. About 1 (100%) parallel efficiency was observed for fine mesh with 1 thread. This might be due to super-linear speedup caused due to improved cache usage when running multiple times.

5.3. Parallel MPI Solver

Table 6 depicts the runtime for MPI solver which includes runtimes for varying number of cores and mesh types.

Table 6: Runtime for MPI solver for various meshes and number of cores

Number of cores	Runtime on coarse mesh (seconds)	Runtime on medium mesh (seconds)	Runtime on fine mesh (seconds)
1	1.671	80.945	791.017
2	1.773	66.867	411.603
4	2.699	81.551	358.849
6	3.873	95.921	403.331
8	5.446	115.467	438.449
10	7.460	134.757	490.879
12	9.779	158.770	539.582
16	12.445	193.615	630.974

The graphs for MPI with varied meshes and number of cores were plotted using the equations for speedup and parallel efficiency presented in the previous section. Figures 13 and 14 demonstrate these, respectively.

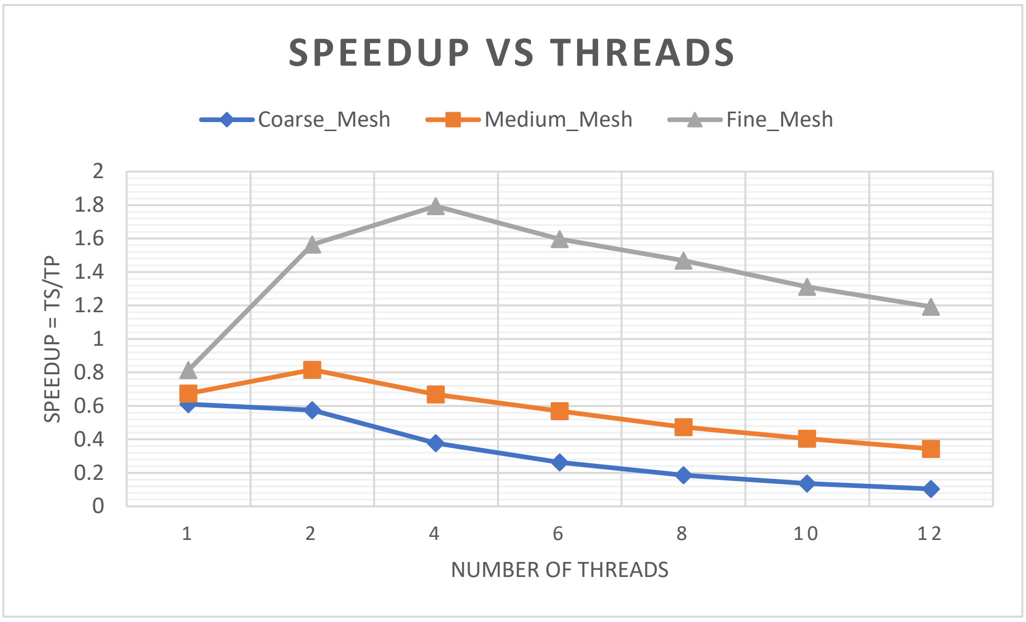


Figure 13: Parallel Efficiency Attained for MPI task

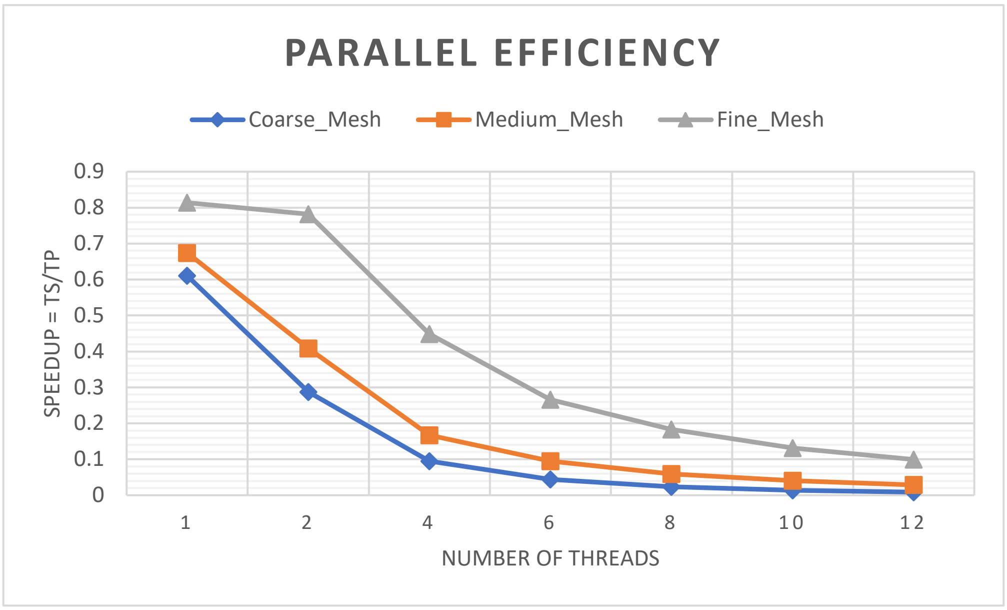


Figure 14: Parallel Efficiency Attained for MPI Task

The speedup for fine meshes is greatest when utilizing 4 cores, while the speedup for coarse meshes is greatest when using 2 cores to execute the solver. The speedup curve displays an upward, peaking, and downward tendency, which indicates the relative performance improvement. However, as the number of cores used to run the code grows, the parallel efficiency of all meshes falls. The finer the mesh, the better the speedup and parallel efficiency that may be attained, is a common factor noticed.

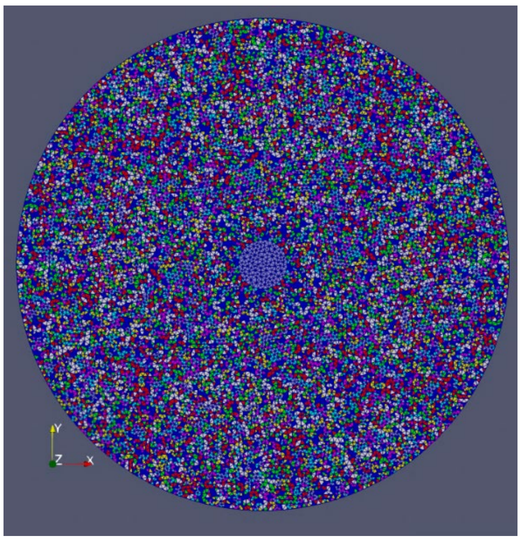
The meshes' maximum and lowest speedup and efficiency values are tabulated in Table 7

Table 7: Speedup and Efficiency values for MPI Task

	Coarse Mesh	Medium Mesh	Fine Mesh
Maximum Speedup	0.610413	0.816397	1.793763
Minimum Speedup	0.081961	0.281951	0.81375
Maximum Efficiency	0.610413	0.674409	0.81375
Minimum Efficiency	0.005123	0.017622	0.06376

Based on this data, it can be stated that in the case of the provided MPI solver, 2,2 and 4 cores should be used for the coarse, medium, and fine mesh respectively, if the goal is to get the highest speedup. However, for utmost parallel efficiency, the operation should be executed on a single core. About 0.8 (80%) parallel efficiency was observed for fine mesh with 1 core(s).

The MPI solver partitioning on a medium mesh with 8 cores can be seen in Figure 15. During gather and scatter, nodes that are touched by components from several processing elements always trigger communication. By splitting the mesh, the number of such border nodes can be minimized. Partitioning is the process of reassigning or permuting element numbers. On each processing element, it seeks to build a compact, continuous subset of elements. It's especially important for unstructured meshes like ours. As shown in the diagram, all items in the model are associated with a part and do not overlap into numerous parts, indicating that the partitioning is appropriate.



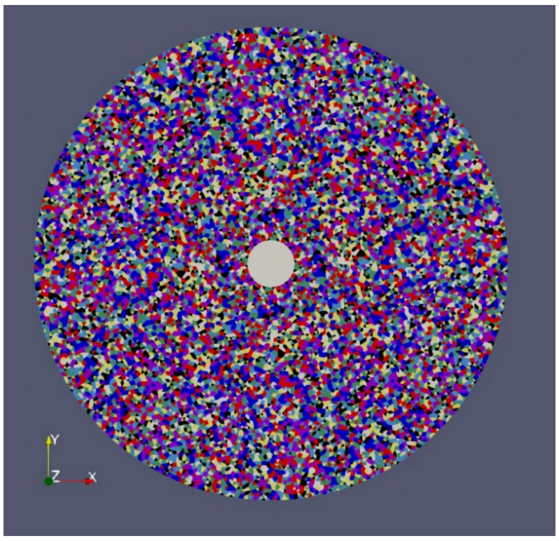


Figure 15: Partitioning for MPI solver with 8 cores and medium mesh 1) Surfaces with mesh edges 2) Surface only coloured for different PEs

MPI_Accumulate allows the caller to combine the data moved to the target process with data already present, such as accumulation of a sum at a target process. The same functionality could be achieved by using *MPI_Get* to retrieve data (followed by synchronization); performing the sum operation at the caller; then using *MPI_Put* to send the updated data back to the target process. Accumulate simplifies this messiness and allows for more flexibility for allowing concurrent operations. Multiple target processes are allowed to perform

MPI_Accumulate calls on the same target location, simplifying operations where the order of the operands (such as in a sum) does not matter.

One-sided communication in MPI enables users to take advantage of *Direct Memory Access (DMA)* to access the data of remote processes; thus, benefiting applications in which synchronization can be relaxed by reducing data movement. The advantages that MPI offers over OpenMP is that it runs on either shared or distributed memory architectures, it can be used on a wider range of problems than OpenMP, each process has its own local variables and last being the fact that distributed memory computers are less expensive than large-shared memory computers. On the contrary the main disadvantages of MPI are that MPI requires more programming changes to go from serial to parallel version and the resulting code can be harder to debug. In addition to that, performance is limited by the communication network between the nodes. (Above paragraphs answers the questions - R3d)

6. CONCLUSION

The FE problem for the Heat equation was compiled and executed serially as well as parallelly using OpenMP and MPI. In addition, the performance of the individual implementations was documented and analysed. In the serial implementation using compiler flags for optimization -O2 -qopt-prefetch combination was found to be the most time efficient and was chosen for further implementation in MPI and OpenMP. Coming to the OpenMP parallelization the speedup showed an increasing – peak – stagnation and further increase for medium and fine meshes, while for the coarser mesh the speedup was reached peak at 10 threads and started dropping further. In addition, the parallel efficiency was found to be decreasing as the number of threads increased, although closer to ~1 (100%) parallel efficiency was observed for fine mesh with 1 thread. This might be due to super-linear speedup caused due to improved cache usage. For execution with MPI, the speedup trend was increase – peak – (followed by) gradual decrease as the number of cores increased. Comparing to OpenMP the corresponding speedup values obtained in MPI were significantly lower. Coming to parallel efficiency, it decreased as the number of cores in the execution increased, the exception being fine mesh with 1 core which had about 0.8 (80%) efficiency.

Coming to additional steps that can be taken is that better compiler flag combinations can be identified which would further improve the performance of the code in its MPI and OpenMP implementation.

7. REFERENCES

[1] https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/compiler-options/optimization-options.html

[2] Lecture and Exercise notes from Parallel Computing for Computational Mechanics