Implementing compiler optimization and OpenMP to speed up unsteady 2D heat equation solver

Mridul Mani Tripathi

German Research School for Simulation Sciences GmbH
Laboratory for Parallel Programming
mridul.tripathi@rwth-aachen.de

Abstract: The aim of this project is to speed up the unsteady 2D heat equation solver by improving serial algorithm and cache utilization through adding desired compiler flags and parallelizing time-consuming loops using various OpenMP statements. The solver can further be sped up by increasing number of CPUs. Scaling and efficiency plots are plotted for analysis after noting down measured timings. A brief description of the solver is provided as follows: mesh file is read first and its data are stored. Element matrices M, K, F are calculated after calculating and storing jacobian for each element. Dirichlet boundary condition is applied. For solving the given heat equation, explicit first order time integration scheme is used. Further global error and discretization error are calculated, latter being calculated from analytical solution.

1 Introduction

Parallel computing is most widely used paradigm today to improve the system performance. System performance can be increased by executing application on multiple processors. OpenMP is API for running application parallelly to improve the speedup. The aim of the project is to speed up the 2D heat diffusion equation solver which is used to obtain the temperature distribution in a plane disk heated by a localized heat source acting only on a subset of the domain represented by small disk. The solver is sped up in three parts. Initially serial algorithm and cache utilization is improved by adding appropriate compiler flags. This serially optimized code is later required for plotting scaling and efficiency plots. Next step is to parallelize time consuming loops using OpenMP statements avoiding data races using two different algorithms and check out various types of scheduling. Third and last step is increasing the number of CPUs and decreasing memory per CPU simultaneously for further speed up of the execution.

2 Description of problem

The solver that is going to be sped up is for the problem described as follows: plane disk is heated by a localized heat source that is shown by gray area in figure 1. The disk has radius R2 = 0.1m. A fixed temperature To = 300K is set on the disk boundary, and a heat source of total power P = 1W is applied on a small circular area of radius R1 = 0.01m centered at the origin.

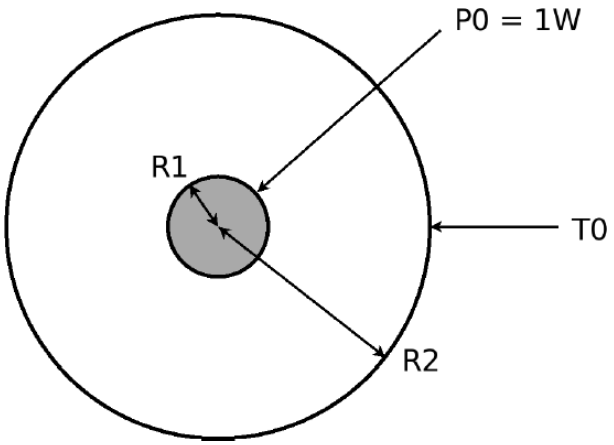

Figure 1: Geometry and boundary conditions

The formulation of the problem to solve is:

$$\begin{cases} \frac{\partial T}{\partial t} - \alpha \nabla^2 T = f & in\ the\ disk\ domain \\ T = To & on\ the\ disk\ voundary \end{cases}$$

Where T, α, and f are respectively the temperature, thermal diffusivity, and the heat source.

The heat source changes based on the location of the node:

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

$Q = P/d_z$ is the volumetric heat source. $d_z = 0.1m$ is the out of plane thickness. The analytical solution is also provided to compute the discretization error in the later stage. The analytical solution of the problem is:

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

Now, the problem at hand is to decrease the time taken by the solver to execute. Our current focus is on adding compiler flags, OpenMP optimizations, and increasing number of CPUs.

3 Methodology

To begin with the problem, we already have a working solver for 2D unsteady heat diffusion equation. Shifting our attending towards speeding up this solver, we can do the same in multiple ways.

Initially, appropriate compiler flags are added to improve cache utilization and serial algorithm. A table is provided highlighting the impact of the different compiler flags used in the CMakeLists.txt.

icc Flag	Description
-g	Tells the compiler to generate a level of symbolic debugging information in the object file.
-O3	Performs O2 optimizations with higher thresholds and enables more aggressive loop transformations
-axSSE4.2	Optimize for AVX (Sandy Bridge) CPU
-axSSE3	Optimize for SSE3 (Woodcrest) CPU
-axSSE2	Optimize for SSE2 (Pentium IV) CPU
-fp-model	Controls the semantics of the floating-point calculations.
-fast	Maximizes speed across the entire program
-mp1	Improves floating point precision and consistency. It ensures the out of range check of operands of transcendental functions.
-qopt-prefetch=3	Enables different levels of software prefetching to reduce cache misses. Its argument varies from 1 to 5, but higher amount of prefetching might increase memory consumption and time.
-no-prec-div	Improves the precision of the floating-point divides and has slight impact on speed.
-ip	Enables the additional interprocedural optimizations for a single file compilation.
-zp8	Specifies the alignment for double precision variables on the 8-byte boundaries.

Table 1: List of compiler flags used along with description

After introducing compiler flags, there is a slight speed up in the execution of the file. The time duration for the execution is decreased slightly for coarse and fine mesh, but for finest mesh the time is decreased drastically. Times before and after adding compiler flags are tabulated below.

Mesh type	**Runtime before adding compiler flags**	**Runtime after adding compiler flags**
	To (seconds)	Ts (seconds)
Coarse	3	0.44097
Fine	67	12.52825
Finest	1435	406.96421

Table 2: Speed up in runtime after compiler optimization

After taking care of compiler flags, we move towards identifying the most time-consuming loops in the solver. For this, omp_get_wtime() function is used. The solver is divided in to four major functions. Thus, to determine the most time-consuming loop, time for execution of all these four functions is noted. It is clearly seen that function explicitsolver() took majority percentage of the total elapsed time. Therefore, we should look for most time-consuming loop in this function.

There is one loop in the function explicitsolver() which has two nested loops. The outer loop cannot be parallelized since it has dependencies. Therefore, we check the time taken by the both inner loops when the function is run. Loop 1 is the most time-consuming loop as it takes maximum amount of time to run. Time taken by loop 2 is also higher but negligible when compared to loop 1. Below is the code excerpt showing the method used to identify the most time-consuming loop.

```
void femSolver::solverControl(inputSettings* argSettings, triMesh* argMesh)
{
   cout << endl << "==================== SOLUTION ======================" << endl;

   mesh = argMesh;
   settings = argSettings;
   int ne = mesh->getNe();

double start1,end1;                          // 1
start1=omp_get_wtime();


   for(int iec=0; iec<ne; iec++)
   {
      calculateJacobian(iec);
      calculateElementMatrices(iec);
   }

end1=omp_get_wtime();
double etime1=end1-start1;
cout<<endl<<"Time consumed by fucntions calculateJacobian() and calculateElementMatrices() together:
"<<etime1<<" seconds"<<endl;

double start2,end2;                          // 2
start2=omp_get_wtime();

   applyDrichletBC();

end2=omp_get_wtime();
double etime2=end2-start2;
cout<<endl<<"Time consumed by function applyDrichletBC(): "<<etime2<<" seconds"<<endl;
```

```
double start3,end3;                         // 3
start3=omp_get_wtime();

   accumulateMass();

end3=omp_get_wtime();
double etime3=end3-start3;
cout<<endl<<"Time consumed by function accumulateMass(): "<<etime3<<" seconds"<<endl<<endl;

double start4,end4;                         // 4
start4=omp_get_wtime();

   explicitSolver();

end4=omp_get_wtime();
double etime4=end4-start4;
cout<<endl<<"Time consumed by function explicitsolver(): "<<etime4<<" seconds"<<endl;


   return;
}
```

Code 1: Measuring runtime of four major functions in solver

To check most time consuming loop in the function explicitsolver(), we use omp_get_wtime() again.

```
void femSolver::explicitSolver()
{
   int const nn = mesh->getNn();
   int const ne = mesh->getNe();
   int const nIter = settings->getNIter();
   double const dT = settings->getDt();
   double TL[3], MTnewL[3];
   double * massG = mesh->getMassG();
   double * MTnew = mesh->getMTnew();
   double * T = mesh->getT();
   double massTmp, MTnewTmp;
   double MT;
   double Tnew;
   double partialL2error, globalL2error, initialL2error;
   double* M;         // pointer to element mass matrix
   double* F;         // pointer to element forcing vector
   double* K;         // pointer to element stiffness matrix
   triElement* elem;   // temporary pointer to hold current element
   triNode* pNode;     // temporary pointer to hold partition nodes
   double maxT, minT, Tcur;
   minT = std::numeric_limits<double>::max();
   maxT = std::numeric_limits<double>::min();



double etime1=0;
double etime2=0;
double start1,end1,start2,end2;

   for (int iter=0; iter<nIter; iter++)
   {
      // clear RHS MTnew
      for(int i=0; i<nn; i++)
      {
         MTnew[i] = 0;
      }

      // Evaluate right hand side at element level
```

```
start1=omp_get_wtime();            //looop 4.1


    for(int e=0; e<ne; e++)                                    // MOST TIME CONSUMING LOOP
    {
      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];
    }

end1=omp_get_wtime();
etime1=etime1+end1-start1;

    // Evaluate the new temperature on each node on partition level
    partialL2error = 0.0;
    globalL2error = 0.0;



start2=omp_get_wtime();            //4.2


    for(int i=0; i<nn; i++)                                    // SECOND MOST TIME CONSUMING LOOP
    {
      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;
      }
    }

end2=omp_get_wtime();
etime2=etime2+end2-start2;


    globalL2error = sqrt(partialL2error/this->nnSolved);

    if(iter==0)
    {
      initialL2error = globalL2error;
      cout << "The initial error is: " << initialL2error << endl;
      cout << "Iter" << '\t' << "Time" << '\t' << "L2 Error" << '\t' << endl;
    }

    globalL2error = globalL2error / initialL2error;
    if(iter%1000==0)
    {
```

```
        cout << iter << '\t';
        cout << fixed << setprecision(5) << iter*dT << '\t';
        cout << scientific << setprecision(5) << globalL2error << endl;
      }
      if(globalL2error <= 1.0E-7)
      {
        cout << iter << '\t';
        cout << fixed << setprecision(5) << iter*dT << '\t';
        cout << scientific << setprecision(5) << globalL2error << endl;
        break;
      }
    }

cout<<endl<<"Time consumed by loop 1 in function explicitsolver(): "<<etime1<<" seconds"<<endl;
cout<<endl<<"Time consumed by loop 2 in function explicitsolver(): "<<etime2<<" seconds"<<endl;

    return;
}
```

Code 2: Measuring runtime for loops inside function explicitsolver()

Now we have an idea about which loops to parallelize. We choose to parallelize both loops taking care of data races using two different ways: Reduction method and Atomic memory access method. Both parallelization methods are used for coarse and fine mesh to check their performance. The better one is used to perform parallelization for finest mesh. Code excerpts are given below for both the methods of parallelization.

For parallelization using reduction method, the following code is used:

```
    // Since function explicitsolver() took maximum amount of time to execute
    // Checking time consumed by each loop in this function

start1=omp_get_wtime();            //looop 4.1

// parallelizing using reduction

#pragma omp parallel for default(shared) firstprivate(elem,M,F,K,TL,MTnewL) reduction(+:MTnew[:nn])

      for(int e=0; e<ne; e++)                                    // MOST TIME CONSUMING LOOP
      {
        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];
      }


end1=omp_get_wtime();
etime1=etime1+end1-start1;
```

Code 3: Parallelizing loop 1 using reduction method

```
start2=omp_get_wtime();              //loop 4.2

//parallelizing using reduction

#pragma omp parallel for default(shared) firstprivate(pNode,massTmp,MT,Tnew,T) reduction(+:partialL2error)

    for(int i=0; i<nn; i++)                                          // SECOND MOST TIME CONSUMING LOOP
    {
      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;
      }
    }

end2=omp_get_wtime();
etime2=etime2+end2-start2;
```

Code 4: Parallelizing loop 2 using reduction method

For parallelization using atomic memory access method, following code is used:

```
// Since function explicitsolver() took maximum amount of time to execute
// Checking time consumed by each loop in this function

start1=omp_get_wtime();            //looop 4.1

// parallelizing using atomic memory access

#pragma omp parallel for default(shared) firstprivate(elem,M,F,K,TL,MTnewL)

    for(int e=0; e<ne; e++)                                          // MOST TIME CONSUMING LOOP
    {
      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

  #pragma omp atomic
      MTnew[elem->getConn(0)] += MTnewL[0];
  #pragma omp atomic
      MTnew[elem->getConn(1)] += MTnewL[1];
  #pragma omp atomic
      MTnew[elem->getConn(2)] += MTnewL[2];
    }


end1=omp_get_wtime();
etime1=etime1+end1-start1;
```

Code 5: Parallelizing loop 1 using atomic memory access method

```
start2=omp_get_wtime();              //loop 4.2

// parallelizing using atomic memory access

#pragma omp parallel for default(shared) firstprivate(pNode,massTmp,MT,Tnew,T)

    for(int i=0; i<nn; i++)                                    // SECOND MOST TIME CONSUMING LOOP
    {
      pNode = mesh->getNode(i);
      if(pNode->getBCtype() != 1)
      {
        massTmp = massG[i];
        MT = MTnew[i];
        Tnew = MT/massTmp;
  #pragma omp atomic
        partialL2error += pow(T[i]-Tnew,2);
        T[i] = Tnew;
      }
    }

end2=omp_get_wtime();
etime2=etime2+end2-start2;
```

Code 6: Parallelizing loop 2 using atomic memory access method

Time taken by both methods of parallelization is tabulated for coarse and fine mesh to get clear idea about each method.

Mesh type	**Time taken due to parallelization by reduction method**	**Time taken due to parallelization by atomic memory access method**
	(seconds)	(seconds)
Coarse	0.42645	1.41194
Fine	11.42899	31.66330

Table 3: Runtime for coarse and fine mesh using different methods of parallelization

A race condition is a special condition that may occur inside a critical section. A critical section is a section of code that is executed by multiple threads where at least one thread writes and at least one thread reads. To avoid data race condition, reduction method and atomic memory access method is used.

It is seen that reduction method does a lot better than atomic memory access method. Even though atomic access provides result immediately, while reduction only assumes its correct value at the end of the loop, but still reduction method is preferred due to its performance. This is because atomic variable comes with a price of synchronization. In order to ensure that there are no race conditions i.e. two threads modifying the same variable at the same moment, threads must synchronize which effectively means that we lose parallelism i.e. threads are serialized. Reduction on other hand is a general operation that can be carried out in parallel using parallel reduction algorithms.

Therefore, we use reduction algorithm to parallelize the finest mesh since it is the better than atomic memory access method. Time taken due to parallelization by reduction method is tabulated.

Mesh type	Time after serial optimization	Time after parallelization using reduction
	Ts (seconds)	Tp (seconds)
Coarse	0.44097	0.42645
Fine	12.52825	11.42899
Finest	406.96421	278.59246

Table 4: Runtime of all meshes after serial and parallel optimization

Now since the solver is parallelized, we now aim at speeding it further using scheduling. Scheduling in OpenMP is a specification of how iterations of associated loops are divided into contiguous nonempty subsets and how these subsets are distributed to threads. Each of these subsets is called chunk. There are three kinds of schedules: static, dynamic, and guided. The table is formed to view runtimes of different scheduling types.

Scheduling type	Time for Coarse mesh	Time for Fine mesh	Time for Finest mesh
	(seconds)	(seconds)	(seconds)
NO Scheduling	0.4265	11.42899	278.59246
Static	0.45625	11.48789	289.12275
Dynamic	0.46360	11.46192	282.84457
Guided	0.41555	11.71047	274.90225

Table 5: Effect of scheduling on runtime

Guided scheduling gives better results when compared with other two. Since the chunk size is not defined, it is taken as default value i.e. 1. In this case, since chunk size is 1, static and dynamic will have large overheads. But in case of guided scheduling, large chunk sizes are taken initially which later reduces to 1 to better handle load imbalance between iterations. Thus, overhead is smaller than previous cases. For dynamic scheduling, extra overhead is involved so we must be careful while using it since each dynamic chunk has some performance impact due to accessing shared state. The overhead of guided scheduling is slightly higher per chunk than dynamic scheduling as there is a bit more computation to do, however guided scheduling will have less total dynamic chunks than dynamic scheduling. If chunk size is defined cautiously, then static scheduling will give better result in scheduling.

Therefore, in this scenario for dynamic scheduling, runtime is always greater than the runtime when no scheduling was done (for all the meshes).

For static scheduling, if chunk size is not provided, it takes chunk size as 1 by default. This results in huge overhead if the number of iterations is high which results in increase in runtime. If chunk size is provided meticulously, then static overhead will be less, and it will result in least run time.

For guided scheduling, it is considered appropriate when iterations are poorly balanced between each other. The initial chunks are larger, because they reduce overhead. The smaller chunks fill the schedule towards the end of the computation and improve load balancing. Since chunk size is not given, by default it is taken as 1. Therefore, runtime using guided scheduling is comparatively lesser than the other two.

At last, we increase the number of CPUs to further speed up the solver runtime. Number of CPUs is chosen from 1, 2, 4, 6, 8, and 12. Memory per CPU is also decreased correspondingly.

Runtime for coarse, fine, finest meshes are noted while varying number of CPUs. The runtimes are tabulated in the table below.

	Runtime (Tp)		
No of CPUs	**Coarse mesh**	**Fine mesh**	**Finest mesh**
	(seconds)	(seconds)	(seconds)
1	0.42438	11.40282	281.25185
2	0.75393	10.78627	193.55402
4	0.57131	7.87759	115.69941
6	0.58505	6.27845	91.47000
8	0.48732	5.89364	83.78482
12	0.55824	5.64186	69.97574

Table 6: Effect of number of CPUs on runtime

Speedup and efficiency are calculated for each case and tabulated using the following formulas:

$$Speed\ up\ (S) = \frac{T_s}{T_p}$$

$$Efficiency\ (E) = \frac{S}{P}$$

Where, Ts = Time after serial optimization

Tp = Time after parallel optimization with P number of CPUs

P = Number of CPUs

	Speedup (S)		
No of CPUs	**Coarse mesh**	**Fine mesh**	**Finest mesh**
1	1.03909	1.09869	1.44697
2	0.58489	1.16149	2.10258
4	0.77185	1.59036	3.51742
6	0.75373	1.99543	4.44915
8	0.90488	2.12572	4.85725
12	0.78993	2.22058	5.81579

Table 7: Effect of number of CPUs on speedup

	Efficiency (E)		
No of CPUs	**Coarse mesh**	**Fine mesh**	**Finest mesh**
1	1.03909	1.09869	1.44697
2	0.29244	0.58075	1.05129
4	0.19296	0.39759	0.87935
6	0.12562	0.33257	0.74152
8	0.11311	0.26571	0.60715
12	0.06582	0.18504	0.48465

Table 8: Effect of number of CPUs on efficiency

4 Results

The efficiency and scaling plots are obtained for coarse, fine, and finest meshes.

Scaling plots:

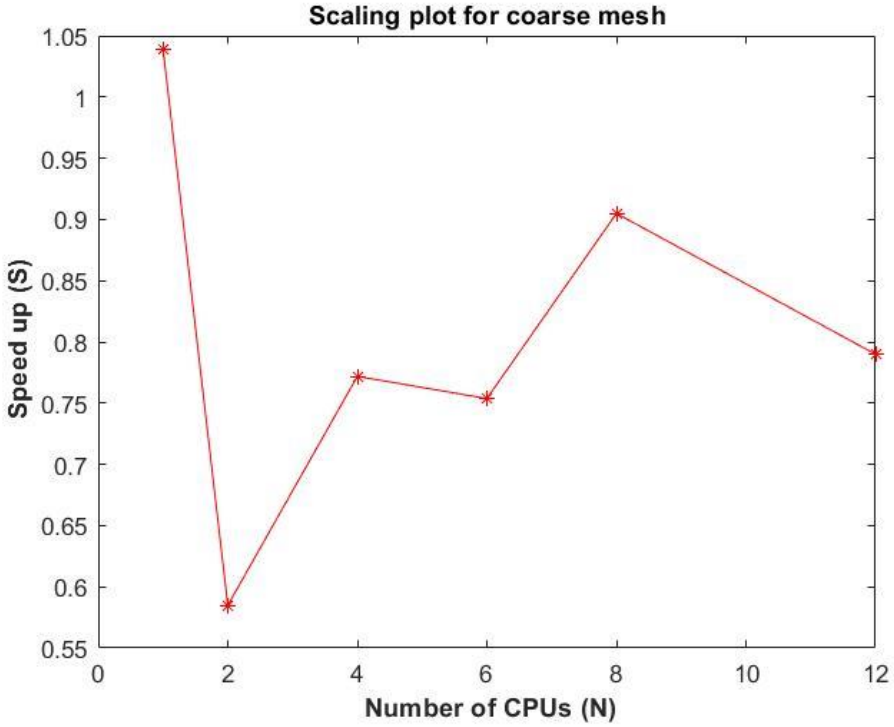

Figure 2: Scaling plot for coarse mesh

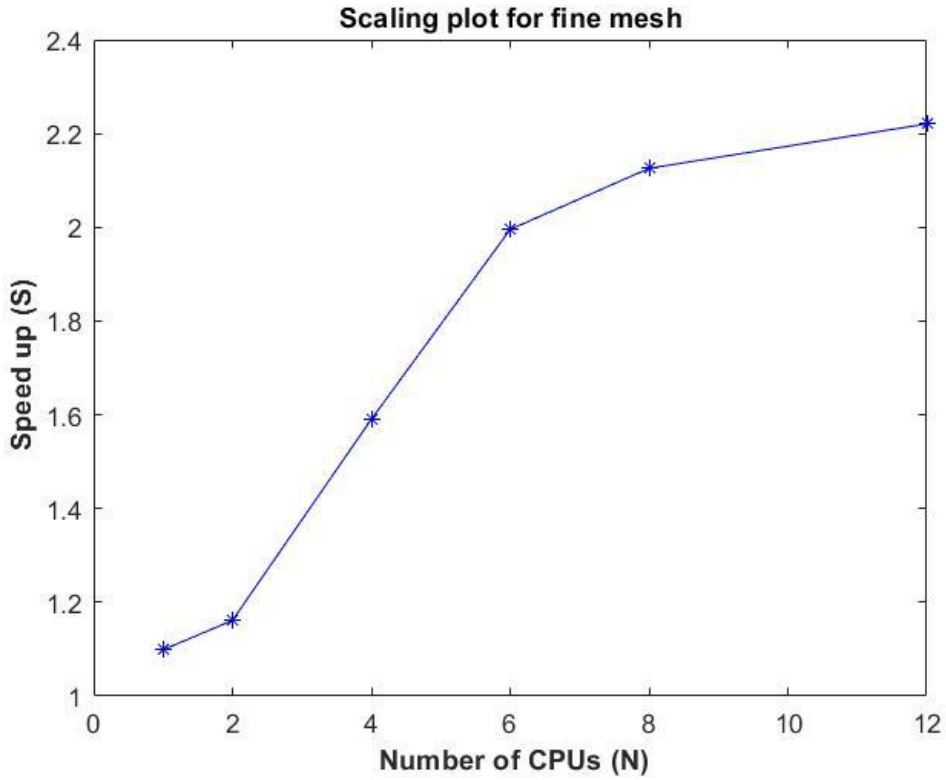

Figure 3: Scaling plot for fine mesh

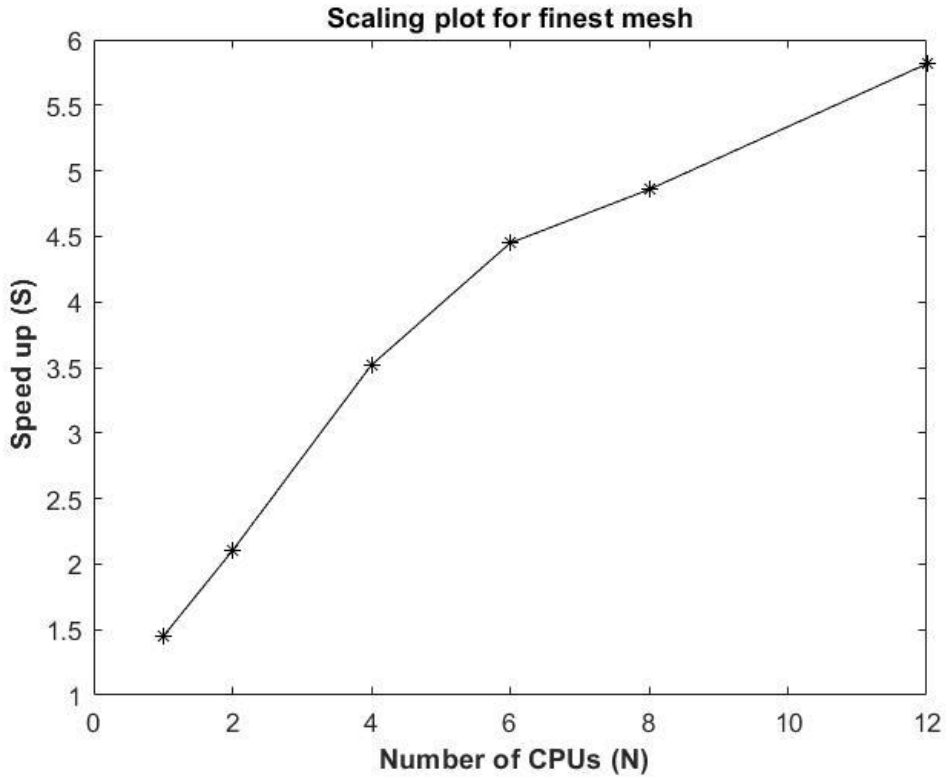

Figure 4: Scaling plot for finest mesh

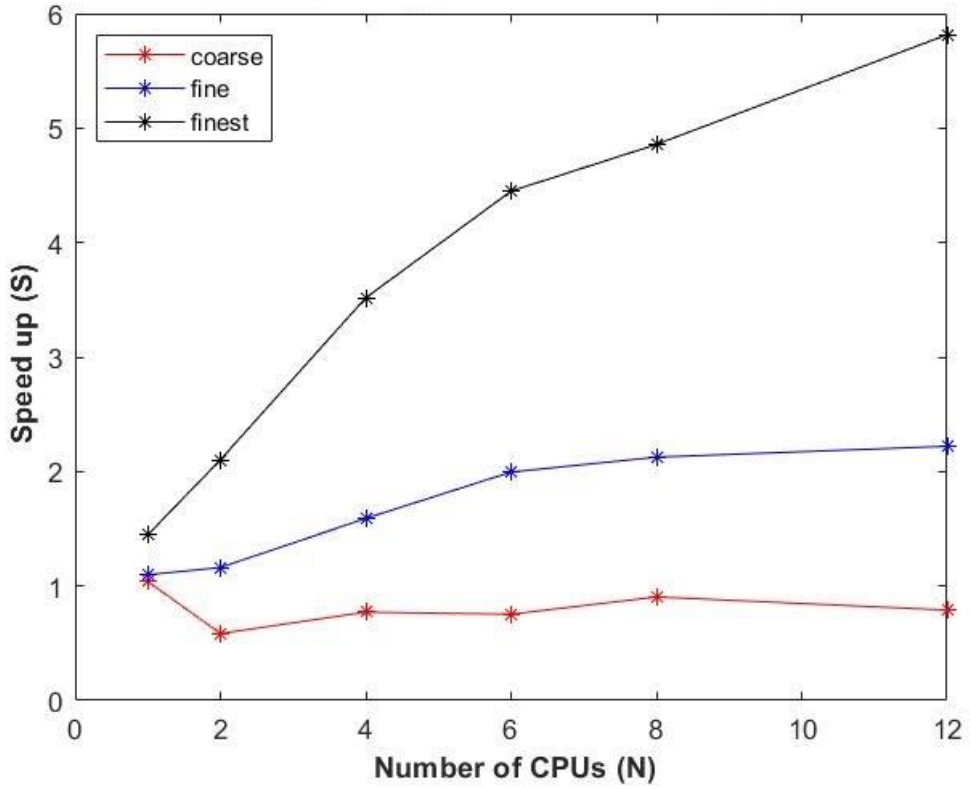

Figure 5: Scaling plot for all meshes

Efficiency plots:

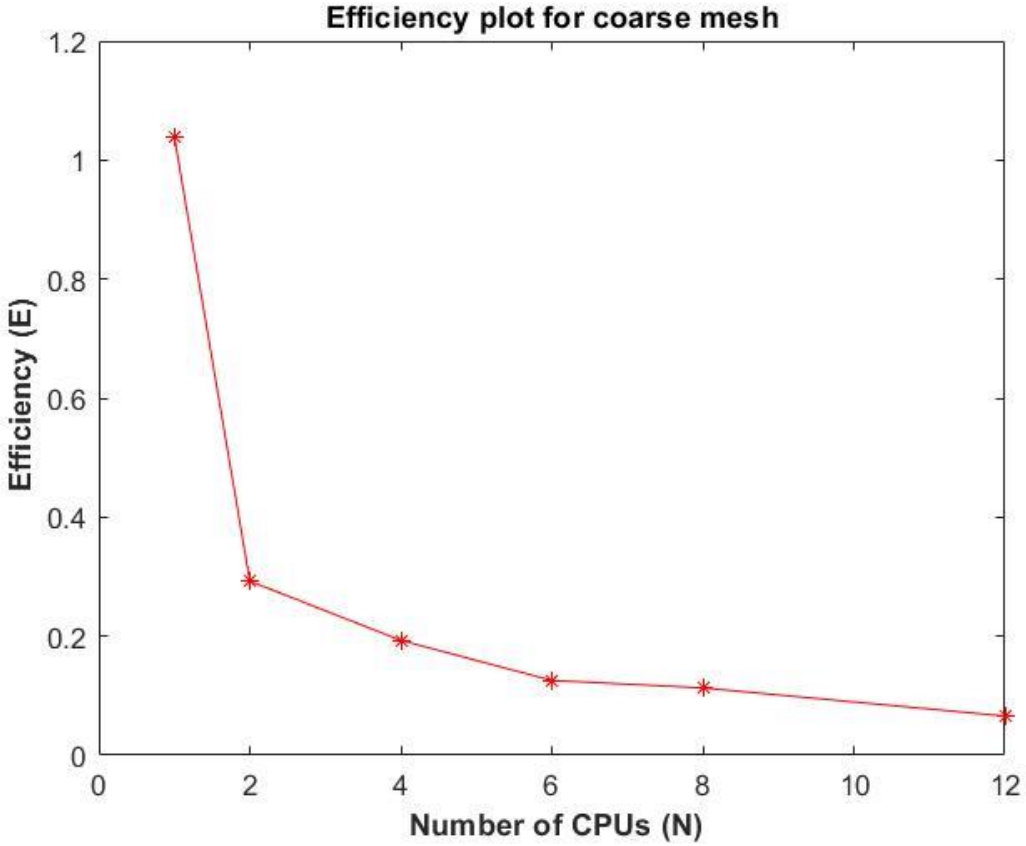

Figure 6: Efficiency plot for coarse mesh

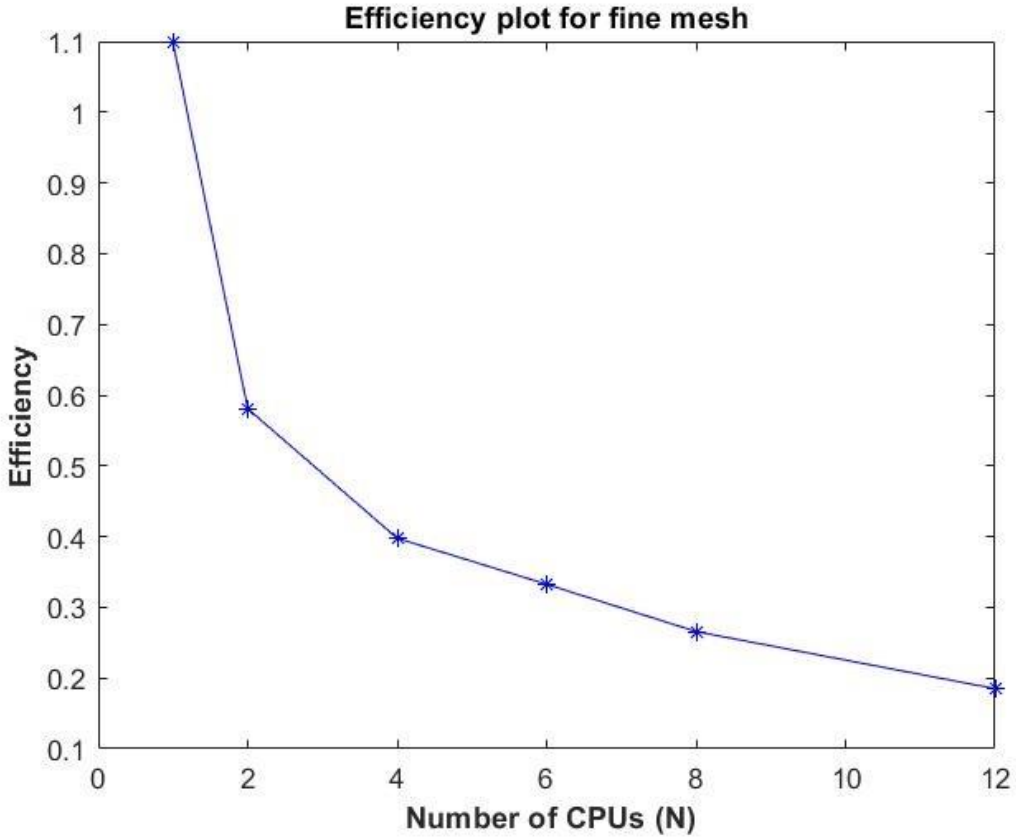

Figure 7: Efficiency plot for fine mesh

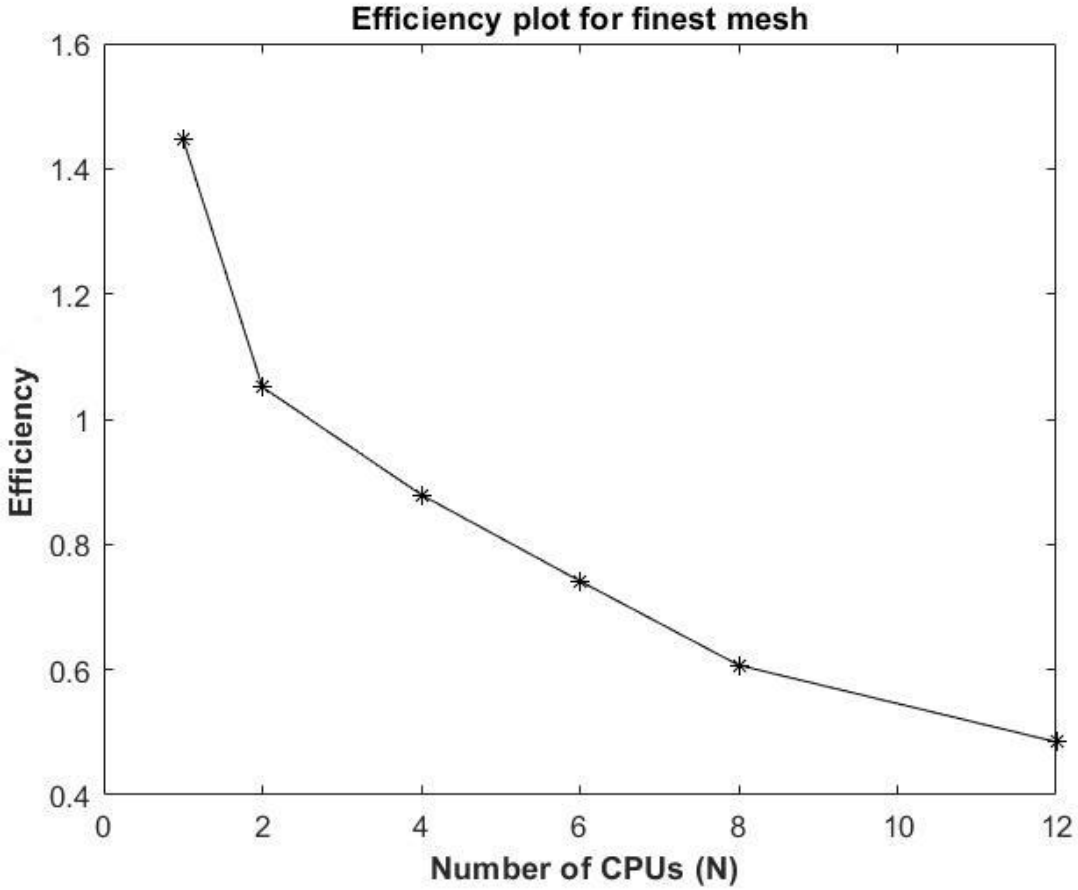

Figure 8: Efficiency plot for finest mesh

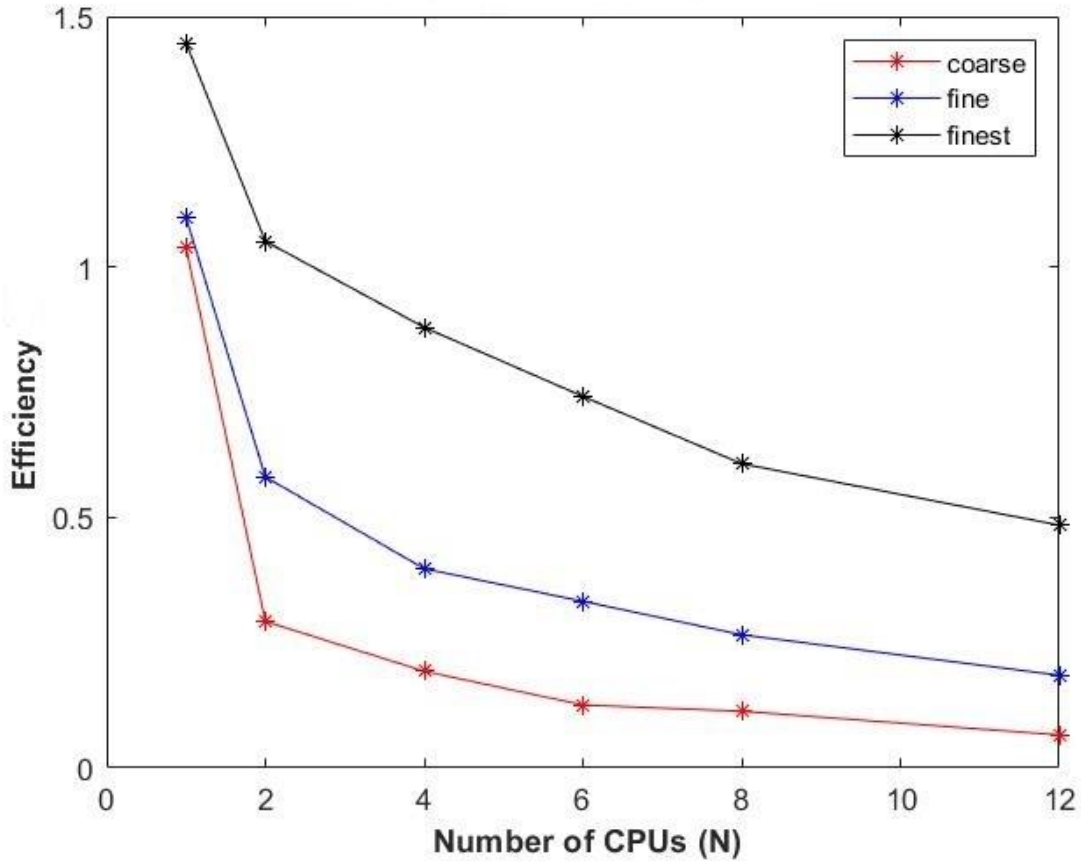

Figure 9: Efficiency plot for all meshes

To verify the results, temperature distribution is plotted for all meshes (coarse, fine, finest). The images are shown below. Distance is the diameter of the disk in the plots of Temp vs Distance.

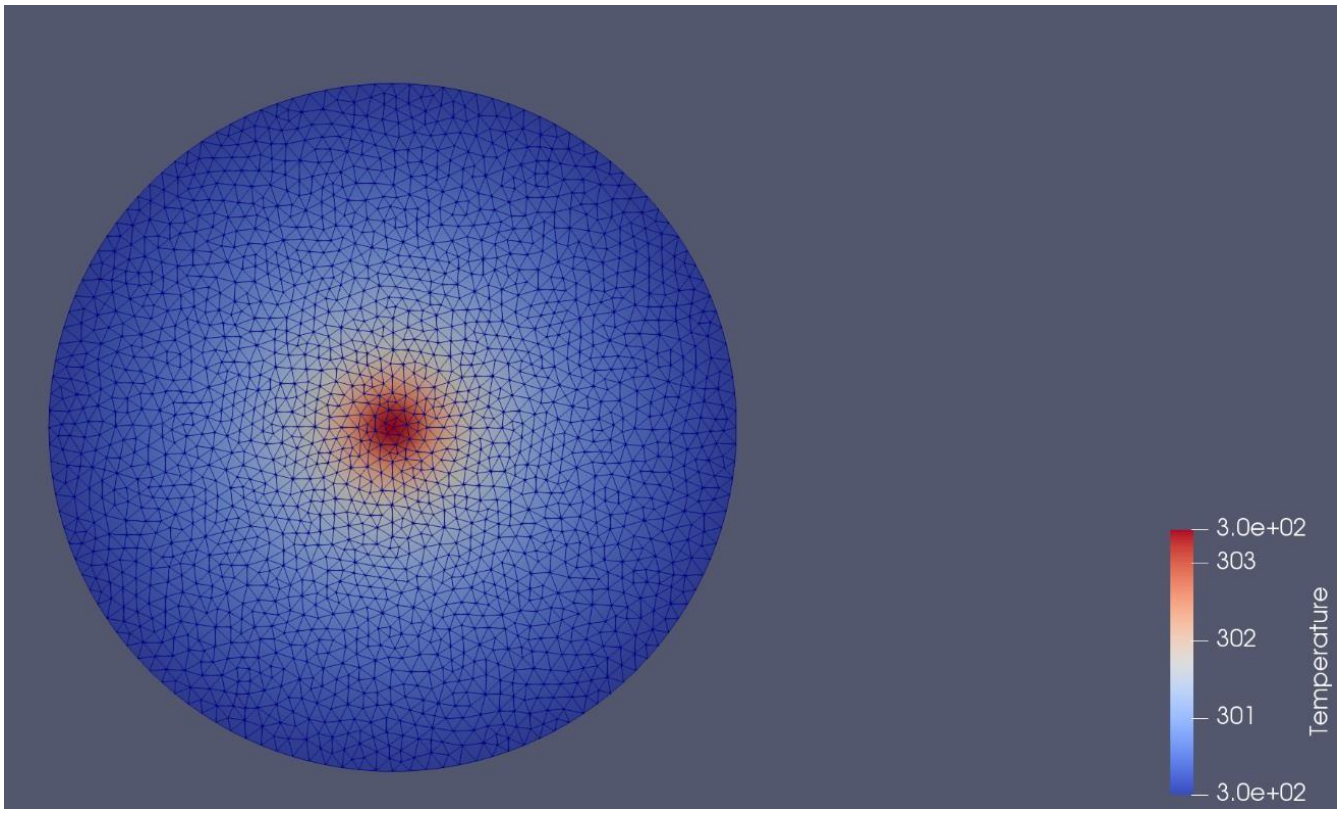

Figure 10: Coarse mesh with temperature distribution

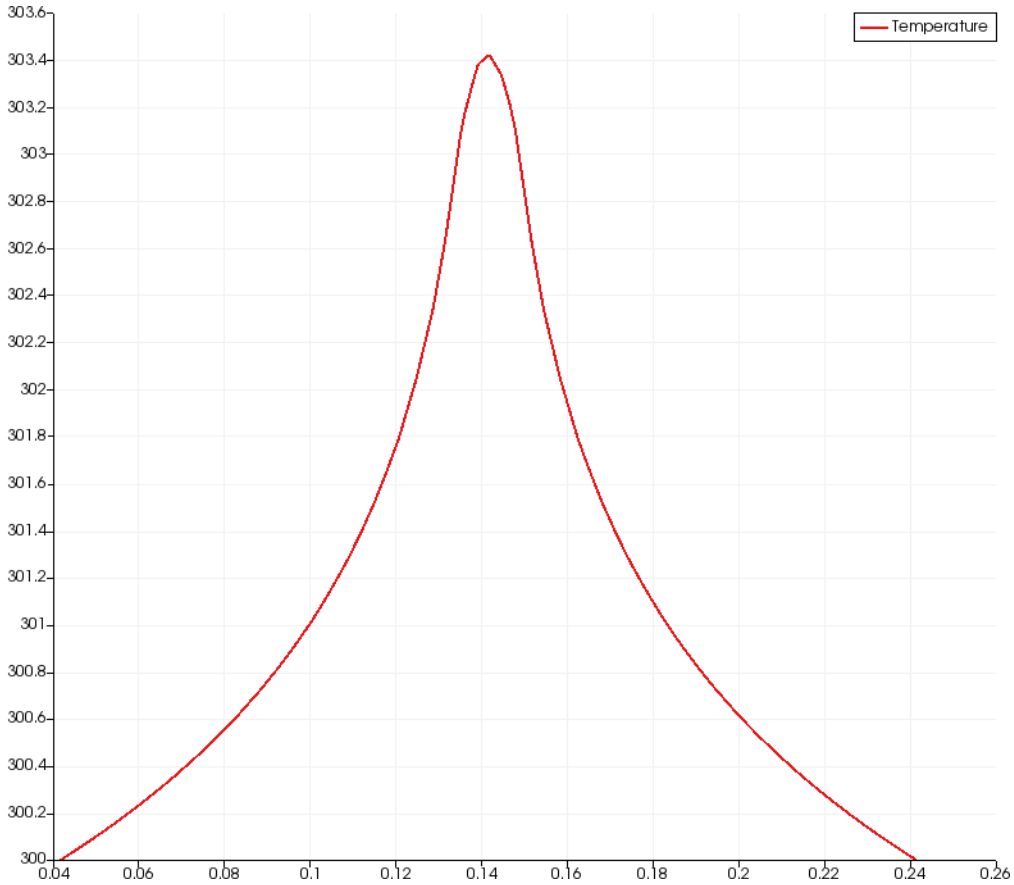

Figure 11: Temperature(K) vs Distance(m) for coarse mesh

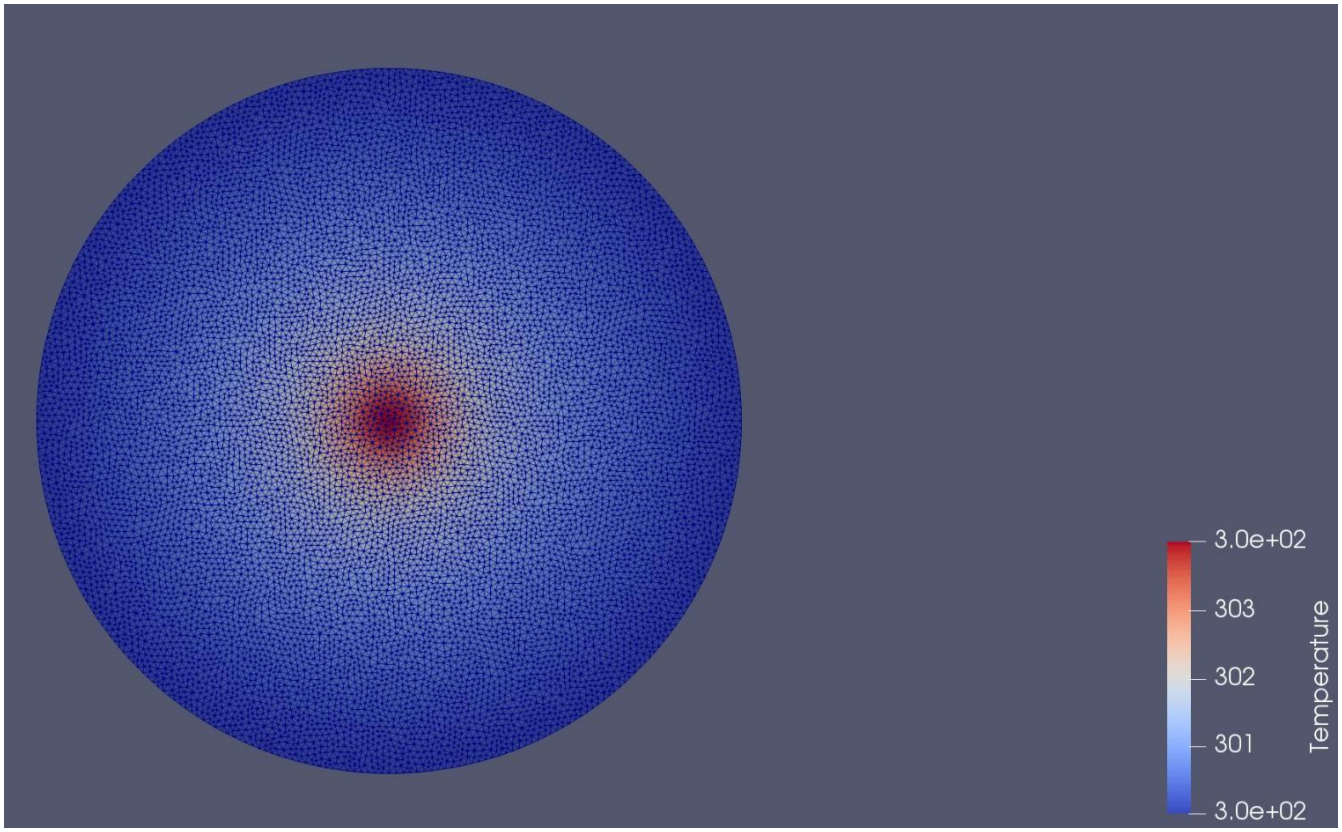

Figure 12: Fine mesh with temperature distribution

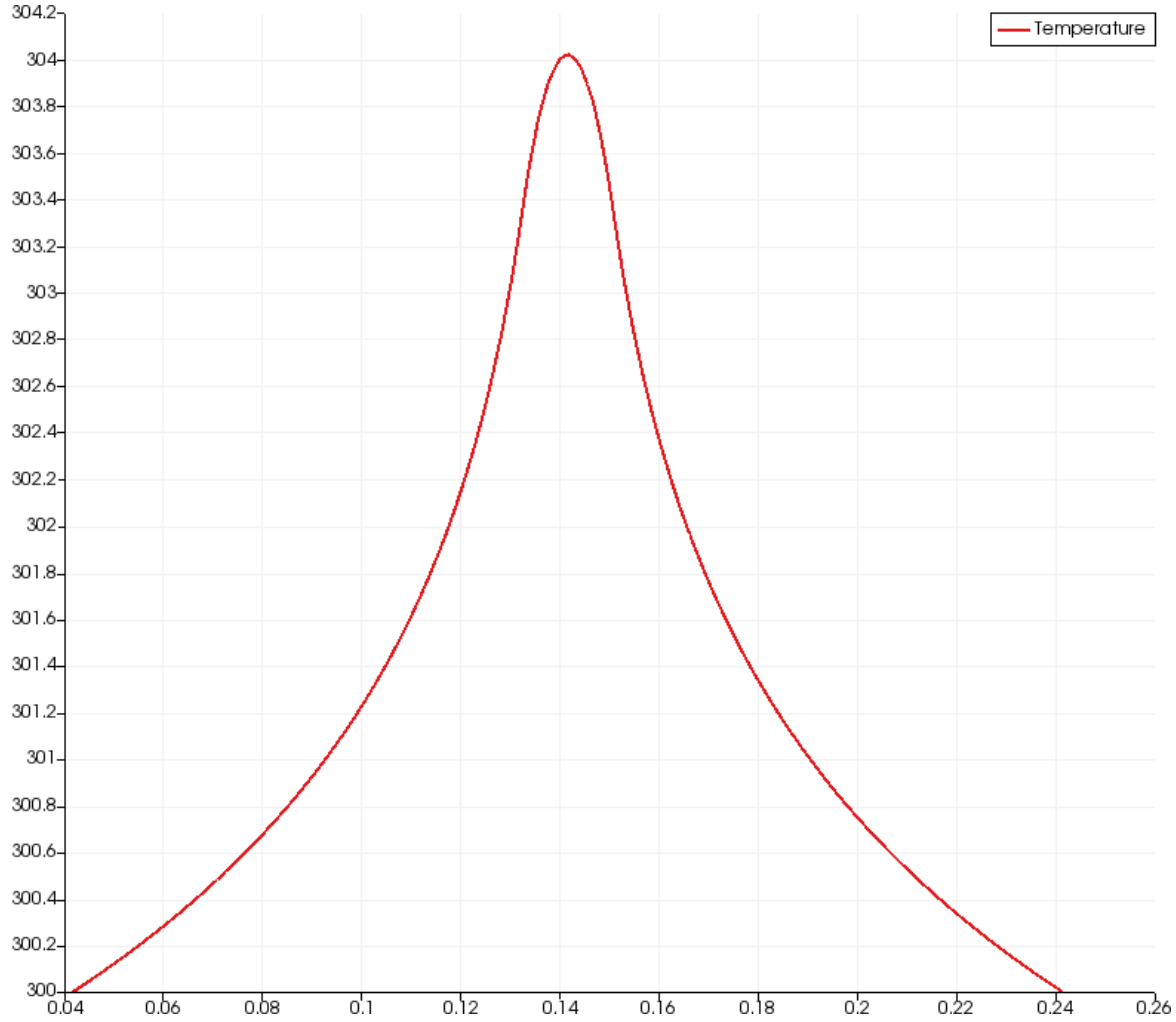

Figure 13: Temperature(K) vs Distance(m) for fine mesh

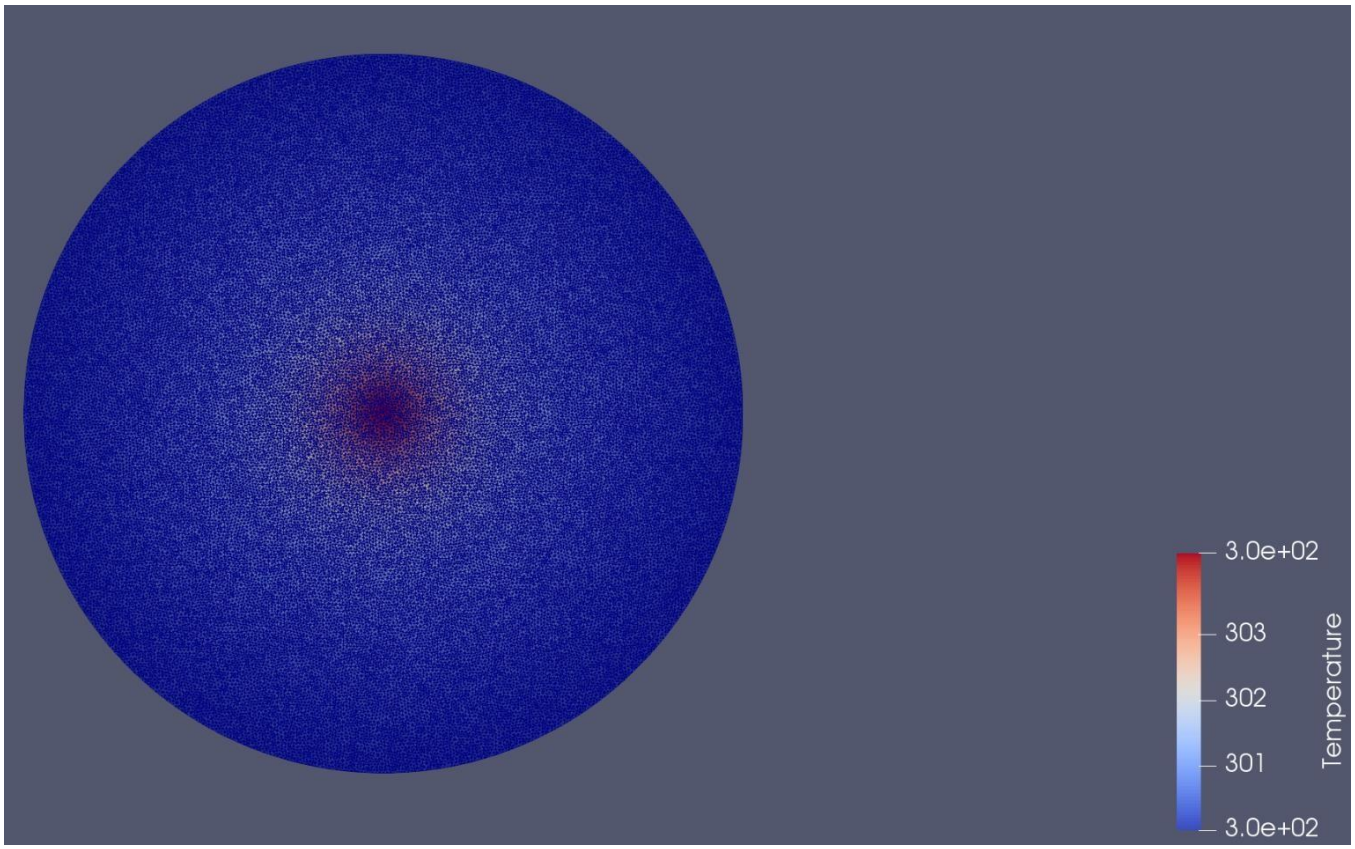

Figure 14: Finest mesh with temperature distribution

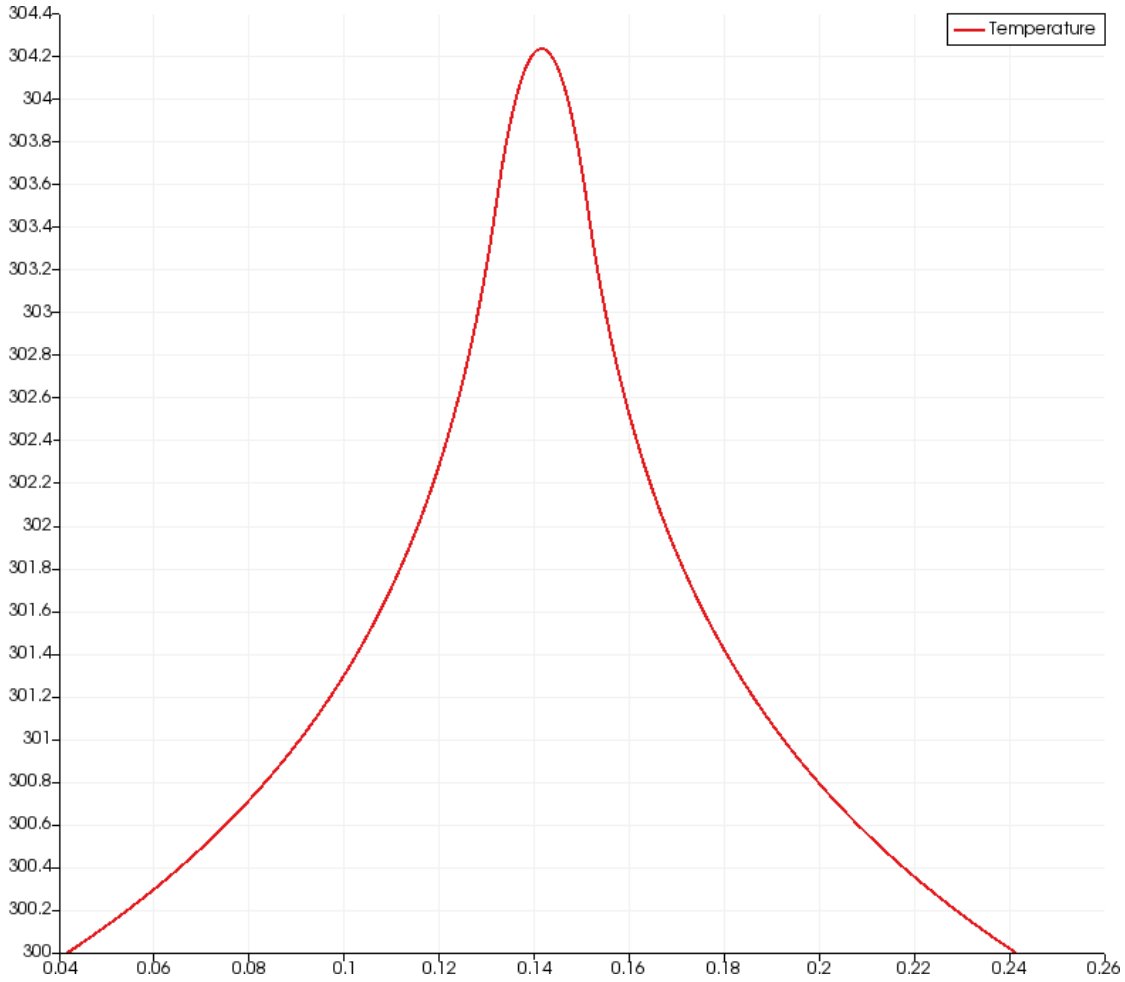

Figure 15: Temperature(K) vs Distance(m) for finest mesh

It is seen that, there is no change in the value of RMS error before and after application of serial and parallel optimizations.

5 Conclusion

Basics of scalar and parallel optimizations have been studied in this paper. For the given 2D heat diffusion heat equation solver, various methods have been discussed to speed up the execution of the solver. To improve serial algorithm and cache utilization, appropriate compiler flags are added which in turn decreased runtime of the solver. Next, parallel optimizations are performed using OpenMP statements. In parallel optimization, most time-consuming loop is parallelized, and effect of different scheduling is discussed. To further speed up the runtime, number of CPUs are increased which result in drastic decrease in runtime.

Finally, the implementation of compiler optimization and OpenMP statements along with increase in number of CPUs resulted in speeding up the solver execution without changing the RMS error which means that results obtained are accurate and converging.

References

[1] https://stackoverflow.com/questions/54186268/why-should-i-use-a-reduction-rather-than-an-atomic-variable/54188140

[2] http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

[3] https://software.intel.com/content/www/us/en/develop/articles/openmp-loop-scheduling.html

[4] https://stackoverflow.com/questions/42970700/openmp-dynamic-vs-guided scheduling#:~:text=The%20whole%20point%20of%20dynamic,load%20balancing%20than%20guided%2C%20k

[5] https://www.smcm.iqfr.csic.es/docs/intel/ssadiag_docs/pt_reference/references/sc_omp_anti_dependence.htm

[6] https://www.irjet.net/archives/V3/i5/IRJET-V3I5110.pdf

[7] https://www.openmp.org/wp-content/uploads/SC17-Kale-LoopSchedforOMP_BoothTalk.pdf

[8] https://www.dartmouth.edu/~rc/classes/intro_openmp/schedule_loops.html

[9] http://www.icl.utk.edu/~luszczek/teaching/courses/fall2016/cosc462/pdf/cosc462openmp_directives.pdf

[10] http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/OpenMP_Dynamic_Scheduling.pdf

[11] https://stackoverflow.com/questions/20413995/reducing-on-array-in-openmp

[12] https://stackoverflow.com/questions/3775147/is-it-possible-to-do-a-reduction-on-an-array-with-openmp/40470790

[13] https://www.openmp.org/spec-html/5.0/openmpse14.html

[14] https://slurm.schedmd.com/sbatch.html

[15] https://michaellindon.github.io/lindonslog/programming/openmp/openmp-tutorial-critical-atomic-and-reduction/index.html

[16] https://www.ijcsmc.com/docs/papers/February2017/V6I2201713.pdf

[17] https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/compiler-options/alphabetical-list-of-compiler-options.html

[18] https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-environment-variables?view=vs-2019#omp-schedule