REPORT: PROJECT 2
Parallel solver for unsteady 2D Heat Equation using OpenMP

German Research School for Simulation Sciences GmbH
Laboratory for Parallel Programming

Rohan Krishna Balaji.
rohan.balaji@rwth-aachen.de

1 INTRODUCTION TO THE PROEJCT:

In the project one, serial version of 2D Unsteady Heat Equation was modeled and solver was implemented. Taking a step further into the same solver, in the project two the serial solver is more optimized using appropriate compiler flags. Further, the code is parallelized used in OpenMP by avoiding data-race, various parameters are compared and presented.

1.1 BRIEF OVERVIEW

The objective of the project as given in the assignment document **[2],** Firstly **Compiler flags** has to be chosen, generally compilers can be optimized to perform well on specific types of operations, and these options can be specified by using compiler flags. These optimizations can be based on hardware, type of operations or utilizing options like optimizing for prefetching, loop unrolling and etc…**[1].** Various types of compiler flags are compared and suitable flags for this particular code are tabulated in further sections.

In the next stage of the project, the optimal compiler flag is chosen and time taken for various sections of code (loops) is measured. Most **time consuming loop** is identified in the explicit solver function of solver.cpp file, this section is chosen for parallelizing.

Now, in the third part of the project the code is parallelized using OpenMP by avoiding Race-Condition or Data-Race. There are many ways to do so, but here I have chosen **Reduction** and **Atomic** clauses to deal with race condition.

In the last major part, the time reduction is compared between the two above mentioned ways, since the former method takes less time it is executed with **more cores** and the improvement in the performance is presented.

2 METHODOLOGIES

In this section of the report, detailed methodology, concept, theory, reasoning and code written is explained in precisely. The whole section is divided into four separate subsections, namely Compiler Flags, Finding Most Time Consuming Loop, Data-Races and Scheduling, Effect of cores.

2.1 COMPILER FLAGS

In this section, effects of compiler flags on the code are discussed. “Intel produces compilers that produce highly optimized code for their CPUs”**[3].** By choosing appropriate flags the optimization can be improvised. There are both aggressive and less aggressive flags.

General Compiler Optimization Flags

Flags	Description
-g	-g compiler flag changes the default optimization from-O2 to –O0
-O1	-O1 may improve performance for applications with very large code size, many branches, and execution time not dominated by code within loops as loop unrolling is disabled.
-O2	It is the optimization level recommended for the general code
-O3	More aggressive than -O2 with longer compile times. It implicitly involves prefetching, loop unrolling, scalar representation and etc. it is recommended for codes that loops involving intensive floating point calculations.

Table 1: Description of various compiler flags compared in this project. [3],[4]& [5]

Other than the General optimization mentioned above, some of the CPU hardware specific flags such as, **-axSSE4.2,SSSE3,SSE4 [5]** are also used. There is also **–ipo flag,** which specialize in inter procedural optimizations across source files .Since this code involves rigorous floating point calculations, **-fp-model fast=2,** flag is used as it optimizes the code involving floating values.

Comparison among various combinations of compiler options is performed to the serial code for coarse mesh and few notable results are presented.

Flags	Elapsed Time (in Seconds)
-g or –O2	2.75
-03	0.59
-03 -, **-axSSE4.2,SSSE3,SSE4 -fp-model fast=2**	0.53
-03 -, **-axSSE4.2,SSSE3,SSE4 -fp-model fast=2 –ipo**	0.57

Table 2: Time Comparison among various flags for serial 2d-Diffusion code using coarse mesh.

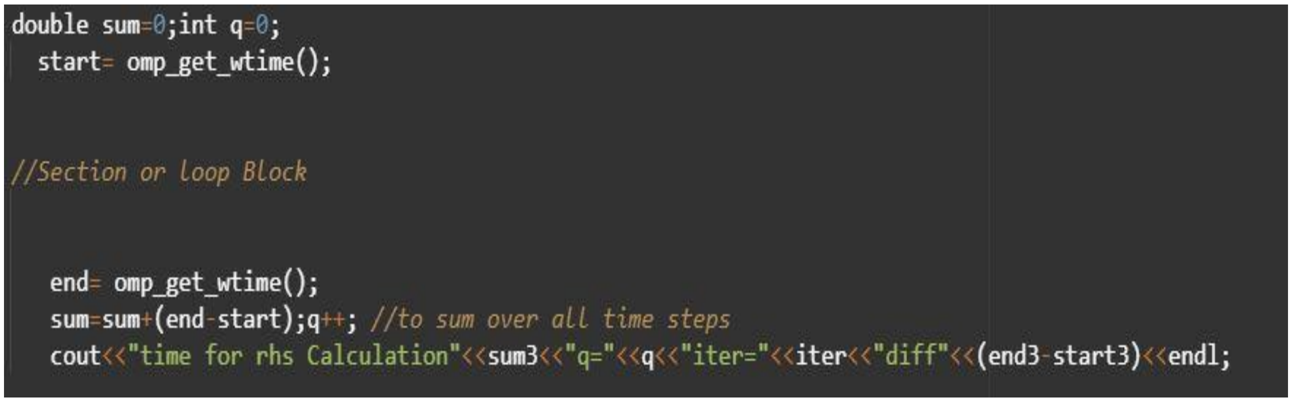
From the above comparison we can see that, there is significant in time, with using –O3 flag, the main reason is that, it involves **prefetching**, it refers to "the using and storing of data into cache before the processor requires the data."**[6]** Since that data is already loaded into the cache, the access of this data can be quick as the processor requires. Further optimizations such as loop unrolling, floating point optimization etc. resulted in improvement in time. So from here on the third flag in the table is used for all the cases. Impact of these flags is same as listed in Table1.

2.2 IDENTIFICATION OF THE MOST TIME CONSUMING LOOP

In the solver.cpp file, these are many loops for calculating Jacobian, element matrices, calculating the RHS part in the explicit solver function, temperature calculations etc. But work-sharing constructs of openMP performs well if there is decent workload in the loops, otherwise it will result in high **overhead.** Thus it is important to identify which loop takes maximum time, so that we can parallelize in the later stage.

First, by inspection I realized that in the **calculatejacobin(e)** and **calculateElementMatrices(e)** function, loop is spanned across elements or quadrature points which are less iterations and takes less time, so parallelizing it won't be very effective. But when we see **explicitsolver()** for loop iterate over all time steps and loop to calculate right hand side at the element level could be most time consuming, thus its worth parallelizing.

But this needs to be reconfirmed, this can be done by measured by using function **omp_get_wtime().** The difference in time between two points in code gives the time taken to execute that section**.** The basic idea for the time calculation is shown below.



```
double sum=0;int q=0;
  start= omp_get_wtime();


//Section or Loop Block


   end= omp_get_wtime();
   sum=sum+(end-start);q++; //to sum over all time steps
   cout<<"time for rhs Calculation"<<sum3<<"q="<<q<<"iter="<<iter<<"diff"<<(end3-start3)<<endl;
```

Code excerpt 1: Finding the most time consuming part of the code

By calculating, time taken at various above mentioned loops, the most time consuming loop is identified as the loop which calculates right hand side at the element level, it takes more than eighty percentage of total elapsed take. Also, in the 2D_Unsteady_Diffusion.cpp file the above function is used to calculate elapsed time.

2.3 DATA RACE:

"A data race occurs when two threads access the same memory without proper synchronization. This can cause the program to produce non-deterministic results in parallel mode"**[7].**

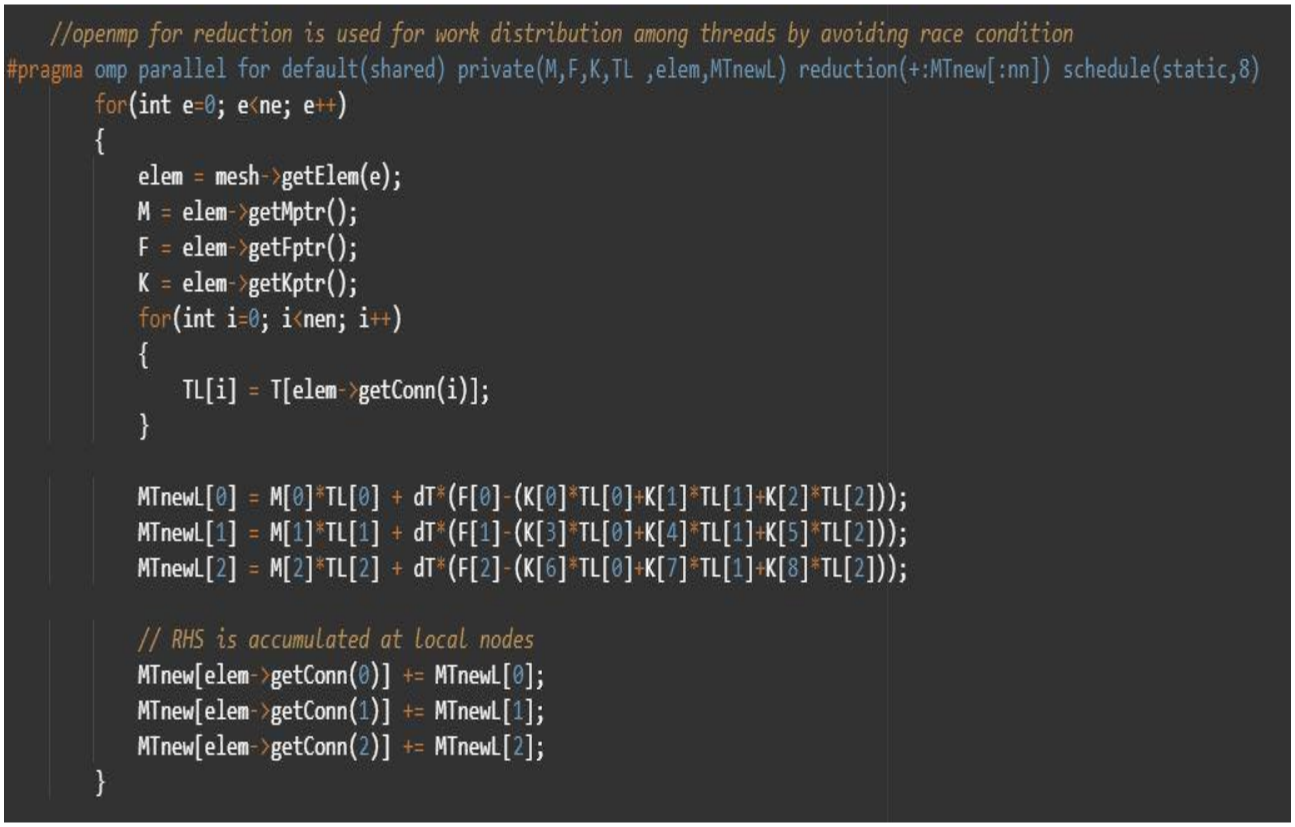
Usually within a parallel region if the variable is having a scope as shared that can be accessed by all the threads within the parallel construct. In such cases the values processed by one thread could by altered by another thread before it completes its operation unintentionally. Thus Race Condition is reached which means the result can't be predicted with certainty.

There are several ways to avoid data races, but here I am considering Reduction an Atomic clauses to avoid such data races.

2.3.1 REDUCTION

In solver.cpp while calculating the right hand side at the element level, if we just use #pragma omp for construct the threads will share the work individually, but when sum performed there is no synchronization amongst the threads, thus any thread can sum first or last before the value is stored and updated creating data race. By using Reduction clause, when the loop is executing, each thread will keep track of the MTnew variable which will be of shared scope, this will be added at the end of the loop, thus avoiding data race.

Implementation of such code is shown below :

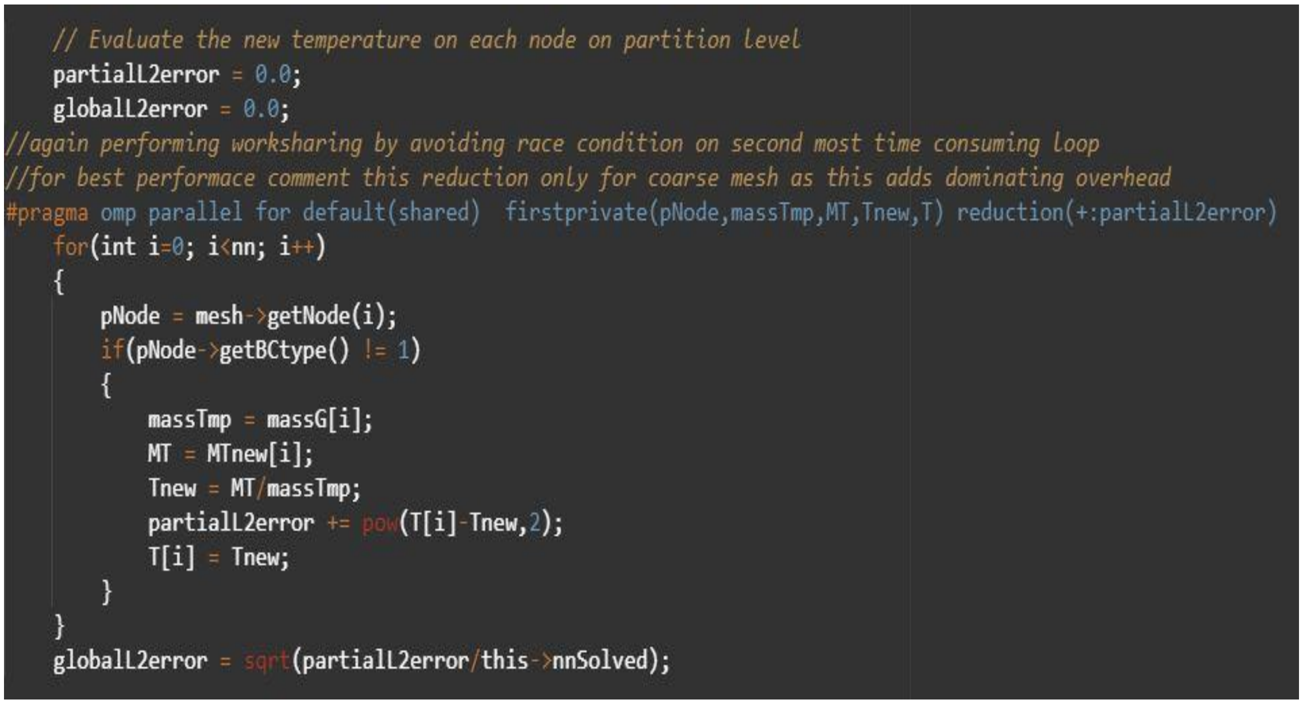


```
    //openmp for reduction is used for work distribution among threads by avoiding race condition
#pragma omp parallel for default(shared) private(M,F,K,TL ,elem,MTnewL) reduction(+:MTnew[:nn]) schedule(static,8)
        for(int e=0; e<ne; e++)
        {
            elem = mesh->getElem(e);
            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];
        }
```

Code excerpt 2: Reduction clause is implemented for worksharing construct by avoiding data race

In this code, variables M, K, F, TL, elem, MTneL are treaded as private (could be firstprivate as well) because each thread that shares the work here should have a mutually exclusive variables, but MTnew must be shared, because that value is summed in the end by all the threads synchronously, by avoiding race condition.

By doing so there is a significant improvement in the performance in terms of time taken to run the code. Although there was significant improvement we could also further parallelize the part of code which evaluates the new temperature on each node on partition level, which is shown below.



```
    // Evaluate the new temperature on each node on partition level
    partialL2error = 0.0;
    globalL2error = 0.0;
//again performing worksharing by avoiding race condition on second most time consuming loop
//for best performace comment this reduction only for coarse mesh as this adds dominating overhead
#pragma omp parallel for default(shared)  firstprivate(pNode,massTmp,MT,Tnew,T) reduction(+:partialL2error)
    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;
        }
    }
    globalL2error = sqrt(partialL2error/this->nnSolved);
```

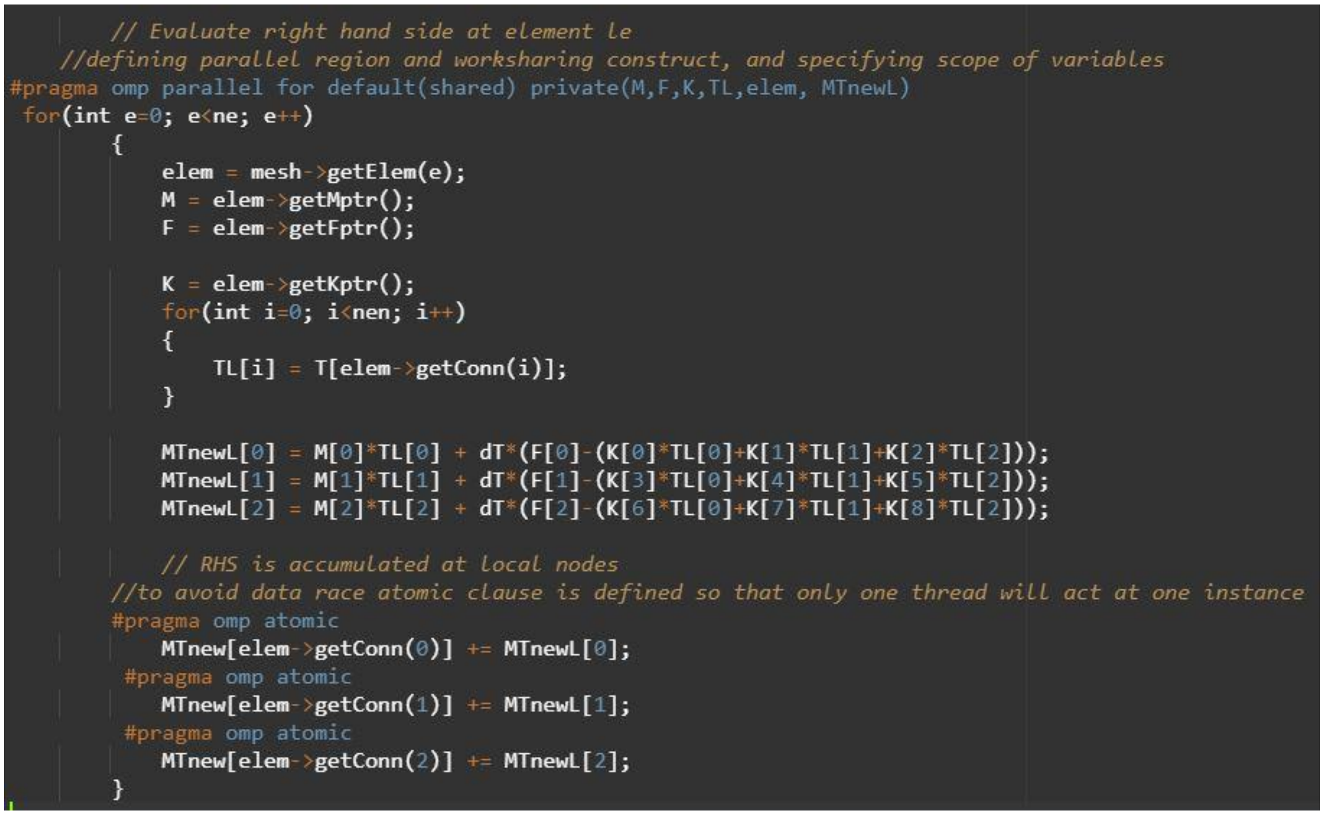
Code excerpt 3: Reduction clause is implemented for worksharing construct by avoiding data race

However, it is important to understand that for the coarse mesh, due to overhead adding of reduction clause in the “code excerpt 3” will decrease the performance. But for the fine and finest mesh, there is definite improvement by adding this additional reduction clause.

2.3.2 ATOMIC

Another way to avoid the race condition is by using “#progma omp atomic” directive inside the omp parallel construct. Atomic updates cannot write arbitrary data to the memory location, but depend on the previous data at the memory location. “It ensures that race conditions are avoided through direct control of concurrent threads that might read or write to or from the particular memory location”**[8]**.

Implementation of such an atomic clause is shown below:



```
        // Evaluate right hand side at element le
    //defining parallel region and worksharing construct, and specifying scope of variables
#pragma omp parallel for default(shared) private(M,F,K,TL,elem, MTnewL)
 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
        //to avoid data race atomic clause is defined so that only one thread will act at one instance
        #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];
        }
```

Code excerpt 4: Atomic clause is implemented for worksharing construct by avoiding data race

Difference in performance is significant between the reduction and atomic clauses, to show this coarse mesh is used. By default, number of cores are equal to number of threads.

Coarse Mesh:

Number of Cores(threads)	Elapsed time Reduction (sec)	Elapsed time Atomic (sec)
1	0.64	1.18
2	0.68	1.8
4	0.66	1.3
6	0.55	1.19
8	0.58	1.08
12	0.57	1.12

Table 3: Elapsed time comparison between reduction and atomic using coarse mesh.

From the above table it is clear that, although both methods give correct results, reduction performs better than atomic in terms of performance. The main reason for atomic to be slower is synchronization among threads, which is more expensive operation. Where as in reduction, partial values are calculated and summed up in the end. So from here on we continue our investigation by only using the reduction for fine and finest meshes.

2.4 SCHEDULING

Scheduling refers to how the iterations associated with loops are divided into chunks and how it is assigned to threads.[9] There are various types of scheduling options available in OpenMP, few are discussed.[10], **Static**: Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. **Dynamic:** Iteration space divided into blocks of chunk size, blocks are scheduled to threads in the order in which threads finish previous blocks. **Guided:** Similar to dynamic, but block size starts with implementation-defined value, and then is decreased exponentially down to chunk.

For Fine Mesh with 12 cores:

Schedule Type	Elapsed Time (in Sec)
Schedule(static,1)	6.55
Schedule(static,12)	5.58
Schedule(dynamic,12)	10.25
Schedule(guided,12)	6.81
Schedule(static,8)	5.49

Table 4: Elapsed time comparison between various schedule types.

From this comparison, scheduling with static and higher chuck size seems to be giving good results (around 5 seconds), it might be because, and workload is evenly distributed among the threads.

2.5 FINE AND FINEST MESHES ON MULTI CORES:

From the previous sections, we concluded that for the best performance of the parallel code, Reduction and Static scheduling should be chosen. By this, we can proceed further by running on more cores (threads) for fine and finest meshes

Number of CPU’s	Elapsed time Fine (sec)	Elapsed time Finest (sec)
1	14.60	452.83
2	11.64	183.17
4	7.80	123.78
6	7.17	95.22
8	5.86	87.69
12	5.58	77.12

Table 5: Elapsed time comparison for multi cores for fine and finest meshes.

To understand the listings in the table clearly, scaling and efficiency plots using these values are made and the explanation and reasoning is given

3.1 SCALING AND EFFICIENCY PLOTS:

Elapsed time for running code in serial is 0.52 sec, 13.84 sec, and 429.82 seconds for the coarse, fine and finest meshes respectively. The time taken for the parallel code on various cores is listed in table 5. Using this data the plots for scaling/speed up and efficiency are made from Matlab.

Speed Up: $S_p = T_1/T_p$, where 'T_1' is time required by the code on one processor and 'T_p' is time required by the code on 'p' processor

Efficiency: $E_p = S_p/p$ with same notations.

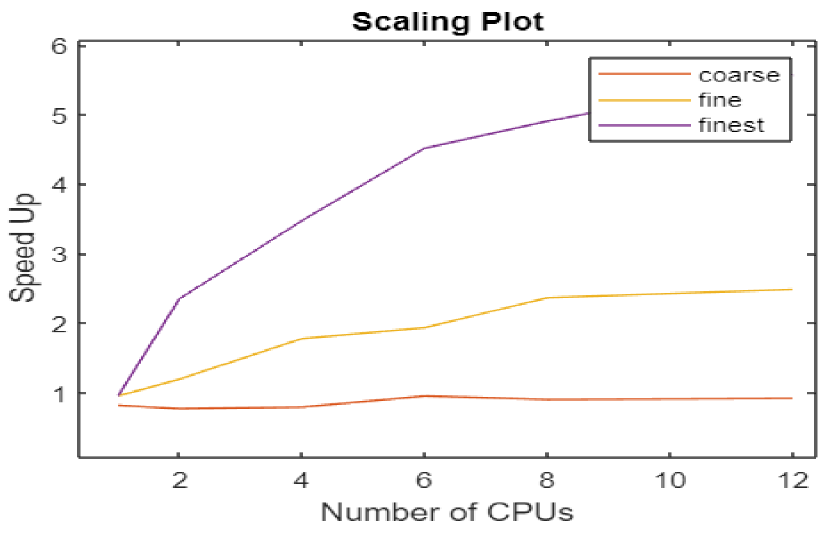


Fig 1: Speedup v/s cores plot for various meshes.

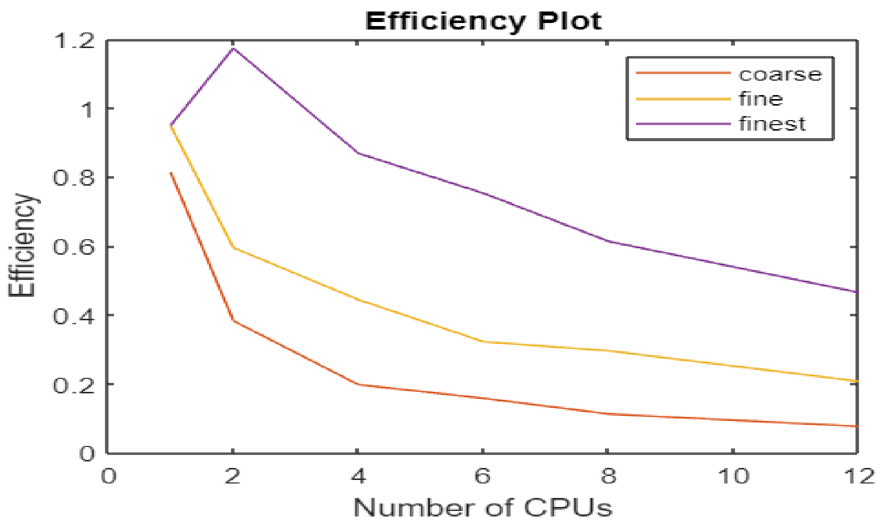


Fig 2: Efficiency v/s cores plot for various meshes.

From Scale-up plot figure 1, we can observe that for the coarse mesh, there is no real speedup in spite of increasing cores, this is maybe because, even after parallelizing, there is not significant reduction in runtime due to high overhead and less work distribution. On the other had for fine and finest meshes there the speed up increases, because they have more number of elements, the work is well distributed. Thus by increasing CPU’s threads can work on each chunk effectively and give better results,.

In the Efficiency plot figure 2, the general trend is that as the number of CPU’s increase, the efficiency decreases. Because as the number of processes and cores increases, the synchronization becomes an important issue, even though the speed up is observed in the fine and finest meshes, the ratio of that with cores is not high as number of cores increases.

3.2 TEMPERATURE DISTRIBUTION:

The accuracy of the result is the most important aspect; the improvement in performance is useful only if the results fall in the acceptable range. One way to verify the parallelization is be observing and comparing the temperature distribution.

Here, I will consider coarse meshe and present the temperature distribution obtained by serial code, parallelization by reduction and atomic. Same can be extended to fine and finest meshes.

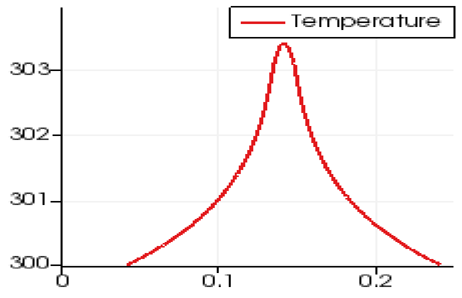


Fig 3: Coarse mesh with serial code.

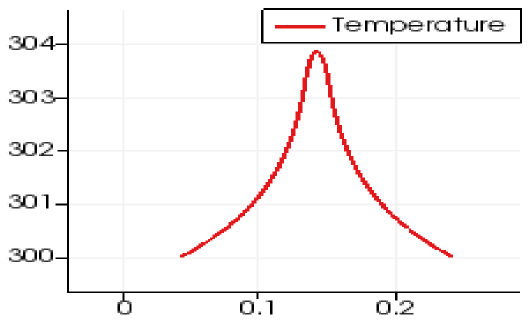


Fig 4: Coarse mesh with reduction code.

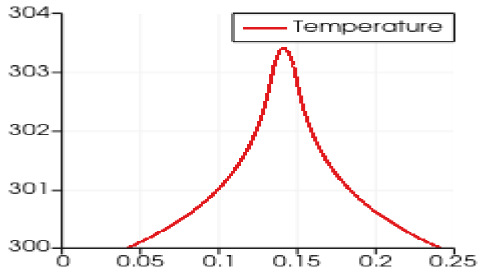


Fig 5: Coarse mesh with Atomic code.

From the above temperature distribution we can observe that, there is no change in final result obtained by serial, reduction or atomic codes. So, that means the data race is avoided and results are accurate.

4 CONCLUSION

From the above discussion, analysis and results presented, we can make few conclusions -O3 based compiler flags are most optimal for this type of code. Further, while parallelizing by work-sharing methods data races can be avoided. In particular, reduction clause performs better than atomic. Static scheduling with decent chunk is optimum for this code since work is uniformly distributed among threads. Further we can conclude that the time of execution can be decreased by increasing CPU's, in other words scale up is possible, but the efficiency for achieving the same reduces as the number of CPU's increases. Finally, the results obtained by parallel code are accurate when compared with serial code.

5 REFERENCES

1. https://scc.ustc.edu.cn/zlsc/sugon/intel/compiler_c/main_cls/bldaps_cls/common/bldaps_using_options.htm
2. Assignment pdf uploaded for the project in moodle.
3. http://www.bu.edu/tech/support/research/software-and-programming/programming/compilers/intel-compiler-flags/
4. https://www.hpc2n.umu.se/documentation/compilers/flags
5. Lecture 7 Notes of course parallel computing in computational mechanics- sisc 2020, moodle
6. https://en.ryte.com/wiki/Prefetching#:~:text=In%20computer%20architecture%2C%20prefetching%20refers,very%20short%20period%20of%20time.
7. https://scc.ustc.edu.cn/zlsc/sugon/intel/ssadiag_docs/pt_reference/references/sc_omp_anti_dependence.htm#:~:text=Data%20race,same%20memory%20without%20proper%20synchronization.&text=In%20OpenMP*%2C%20loops%20are%20parallelized,loop%20iterations%20to%20different%20threads.
8. https://www.ibm.com/support/knowledgecenter/SSGH2K_13.1.2/com.ibm.xlc131.aix.doc/compiler_ref/prag_omp_atomic.html
9. https://www.openmp.org/wp-content/uploads/SC17-Kale-LoopSchedforOMP_BoothTalk.pdfs
10. Introduction to high performance computing notes, sisc 2019.