The “High Performance Computing” (HPC) assignment analyses the run time performance of the **deqn** code and attempts to improve the run time performance by using OpenMP to parallelize the code as much as possible. Before beginning the parallelization process, it’s important to first get a benchmark of the performance of the code. The overall performance of the code can be measured by looking at **main.c** file’s **main** method. Over here, an instance of the **Driver** is created, and it’s **run** method is invoked. As the code is executed extremely quickly, I have added a for-loop with a fixed number of iterations to repeatedly execute the **run** method. The total time taken to run the iterations is divided by the number of iterations to calculate the average time taken to run each iteration.

Preparations for benchmarking

For a fair comparison in the benchmarking of the performance of the simulation before and after the OpenMP directives were added, some changes are required to be made to the code. Any changes made here is purely for a fair comparison purpose and has been applied to both versions of code (before & after). The following changes to the code was made:

1. Stop writing to .vtk files: Writing to the .vtk files is an expensive process as IO is required and this acted as a severe bottleneck in the performance. Removing the printing to files would allow just the CPU code to be benchmarked accurately.
2. Stop printing to console: Just like writing to files, printing data to the standard output is IO and acts as a severe bottleneck. Removing printing to standard out will allow for a fair comparison.

Parallelization methodology

Mesh::getTotalTemperature

The class Mesh has a method called **getTotalTemperature**. In the method, the temperature is calculated by adding up the values stored in the array, **u0**. Calculating the sum of the array is a very simple operation. As a result, this was initially parallelized by using **#pragma omp parallel for** on both the outer and the nested for-loops. However, the use of OpenMP here caused a race condition resulting in the incorrect value of the temperature being calculated. To solve the problem, **reduction** has been used on the outer for-loop. A private variable, **_temp** is declared with the reduction function working on the addition operator with the **temperature** variable.

ExplicitScheme

The class ExplicitScheme has three methods with for-loops. Out of these three methods, **reset** and **diffuse** methods took the longest time during the testing phases. As a result, these are the two methods in this class which were parallelized. For both the methods, the outer and the nested for-loops were parallelized using **#pragma omp parallel for schedule(dynamic, 100)**.

Diffusion::init

The initialization method in the Diffusion class contains two sets of nested for-loops. All of these for loops have been parallelized with **#pragma omp parallel for schedule(dynamic, 100)**.

Non-OpenMP related change

A change has been made to the **Driver** class. Over here, the **t_current** variable was being divided by **dt** in every single iteration of the for-loop in the **run** method. It is a well-known fact that a division operation is more computationally expensive compared to a multiplication operation. As a result, I have calculated the value of **1 / dt** and stored the value in a variable, **one_div_dt**. After that, for the division operation, **t_current** is multiplied by **one_div_dt**. This achieves the same result as the division but is less computationally expensive.

Performance benchmarking

For the overall performance benchmark, the **square.in** file has been used as it requires the most time to execute. The file has been updated to increase the value of **nx** & **ny** to **1000** and the value of **xmax** & **ymax to 1000.0.** Furthermore, the **driver.run()** method is called multiple times using a for-loop to increase the total time taken. In the initial project without any OpenMP directives, it is assumed to run in a serial manner with only one thread. Therefore, the controlled variables in this experiment are:

- Number of threads: 1, 2, 4, 8, 20, 100, 500, 2000
- Input file: square.in
- Number of iterations: 200
- Scheduling: Static: 1, 100, Dynamic: 1, 100

The four performance graphs showing the run time in seconds against the number of threads is given on the next page. It is evident that the best performance is achieved between **4 and 20 threads** with **8 threads** being the most optimal number for the machine on which this was tested. The number of threads varies with the machine. From the graphs, it is very clear that a low number of threads such as 1 and higher number of threads (100 and over) are detrimental to the performance of the algorithm. Having too few threads means there is not enough parallelisation. On the other hand, having too many threads is problematic due to the overheads of thread management and cost of switching to different threads.

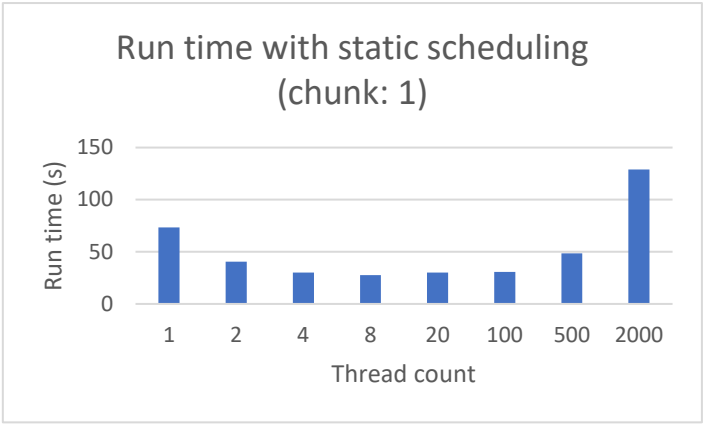
The overall best performance was obtained with dynamic scheduling chunk size 100 utilizing 8 threads. Further benchmarking has been performed. In the additional benchmarking process, the average time of each of the parallelized loop was measured in detail against the number of threads being used. The data table is given after the graphs.

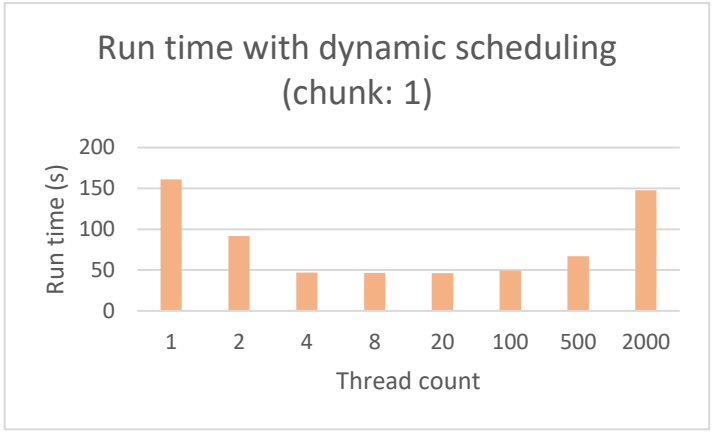
Benchmarking of each loop against the number of threads being used

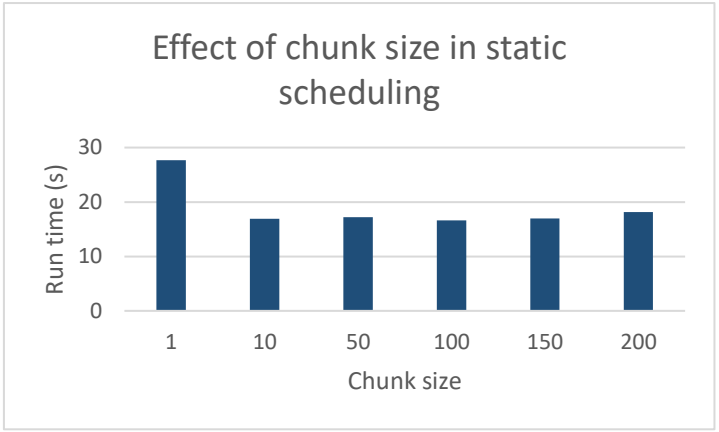
Thread count	Diffusion	ES:Diff	ES:Res	Mesh:GetTotTemp
1	0.014939	0.003341425	0.00282465	0.002321725
2	0.008731	0.0019701	0.00165304	0.00128156
4	0.011527	0.001665495	0.001422025	0.00095015
8	0.012868	0.00157045	0.000139998	0.00095088
20	0.015462	0.00159878	0.001414345	0.000939695
100	0.023755	0.001875255	0.00163264	0.001135655
500	0.078902	0.00302142	0.00278971	0.002143885
2000	0.0573856	0.008513005	0.008118813	0.005710405

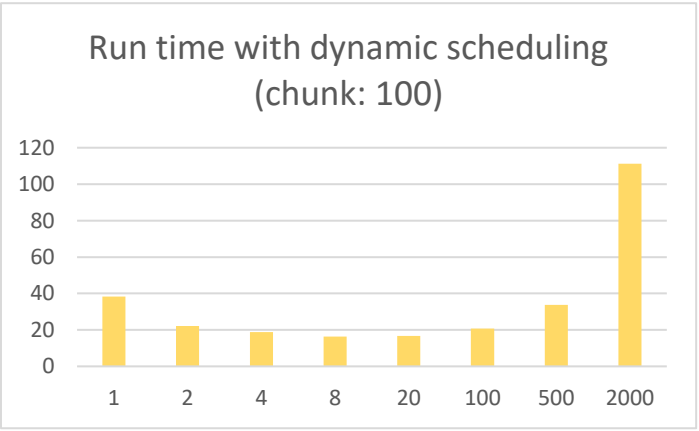
The table above shows the time taken on average by each of the optimized loop against the number of threads that is being used at the same time. As expected, the table shows the most time is taken when the thread count is very low at 1 and when the thread count is very high at 2000. The best case scenario in most of the times is when the thread count is 8.

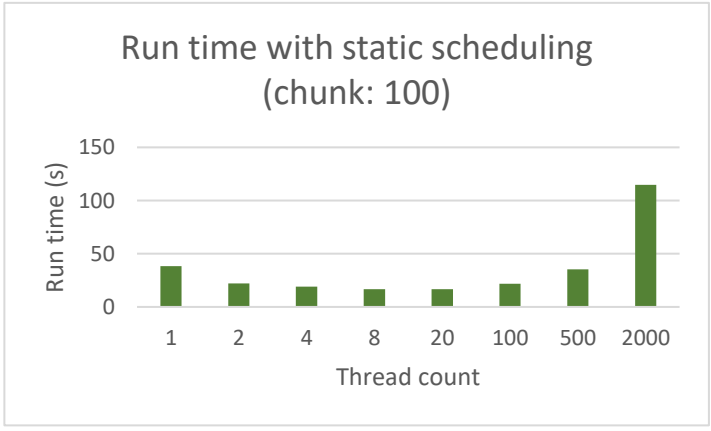
All the information was printed out to a text file. Then a Python script **calculateAverage.py** was used to calculate the mean time taken for each of the loops. The text files containing the times are in the format {thread}.txt in the root directory.

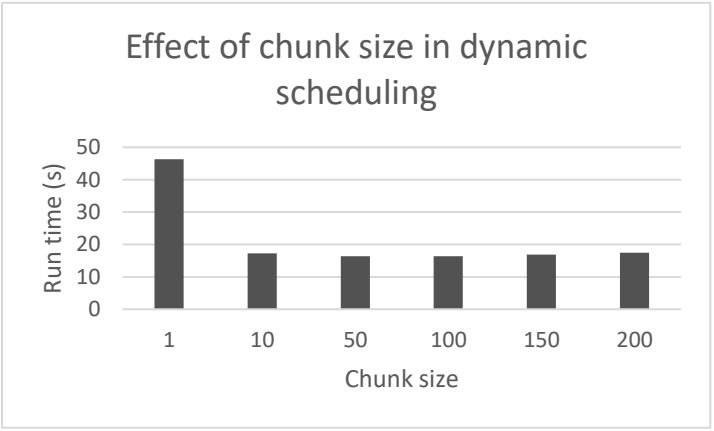












Argha Sarkar

Iteration scheduling

In the benchmarking process, the data showed that approximately 8 threads with dynamic scheduling and chunk size of 100 had the best performance. In order to get a better idea of the optimal iteration scheduling strategy, I created an experiment where the number of threads was held at a constant value of 8. The scheduling was changed along with the chunk size. The table below shows the results:

Scheduling	Chunk	Run time (s)
Static	1	27.7
Static	10	16.91
Static	50	17.21
Static	100	16.65
Static	150	16.99
Static	200	18.13
Dynamic	1	46.36
Dynamic	10	17.22
Dynamic	50	16.42
Dynamic	100	16.34
Dynamic	150	16.85
Dynamic	200	17.46

The data in the table shows that there is a severe performance penalty when using a chunk size of 1 regardless of the scheduling type. The performance gain of using chunk size of anything between 10 to 200 is minimal. However, there is a trend for both dynamic and static scheduling which shows that the best performance occurs at chunk size of 100. Anything less or greater than this value takes longer.

Measuring the time taken for code execution

The method **omp_get_wtime** must be used from the OpenMP library to measure the time for execution in this instance. The time can be measured by getting the time at the start and the end. The difference between the start and end times is the time taken for the code to run.

OpenMP for parallelization

OpenMP is the parallelization library used for the task of optimizing the code. By adding OpenMP directives such as **#pragma omp parallel for** before for-loops, it instructs the compiler that it is possible to improve the performance of the code by running it simultaneously on multiple CPU cores rather than running it sequentially. The number of threads used for this parallelization is the number of cores available on the machine. It is possible to find out the number of threads by using the function, **omp_get_num_threads** function. Generally, this number varies from 4 to 8 on modern machines. To override the default number of threads and to measure the performance of the program against the number of threads used, **omp_set_num_threads** function has been used at the beginning of the file **main.C**.

Measuring the overhead of OpenMP

The overhead can be measured using Amdahl's law.