High-Performance Computing Assignment 1: Diffusion Equations

Agata Borkowska, UID: 1690550, *MSc in Computer Science, University of Warwick*

I. Introduction

We are presented with a sequential code that calculates the result of the diffusion equations, at a specified interval, at given points of a grid. We begin by providing a brief overview of the code.

The *Main* method only parses the input file to instantiate and run the *Driver*. The *Driver* class is responsible for setting up the mesh, then advancing the calculation step by step by calls to the *Diffusion* class, and writing the result to the file each time via the file writer. It contains the loop that makes call at each step - at dt intervals, rather than at each cell of the mesh. This is the longest loop in the program, and we identify it as the key to improving the performance.

More specifically, the Driver's constructor directly calls *Mesh*, *Diffusion*, and *Writer* constructors, in that order, as the created mesh is an input to the latter two constructors. Therefore, the mesh will be the focus of our first attempt at parallelism.

The *Mesh* class is responsible for creating the grid described in the input file. It runs two for loops, one for the x, one for the y coordinates. In the original code they are ran one after another, but as there is no overlap between them, running them in parallel is an obvious first step. The loops in it are very simple, and can be run concurrently, so it will help us measure the overhead, and we do not expect to achieve as much with this as with the other classes.

The *Diffusion* class sets up the scheme, and the *Diffusion.doCycle()* method acts as a wrapper for the scheme's *doAdvance()* method. The constructor has a nested for loop, however we'll leave that one for later, and focus on speeding up the calculations.

The scheme makes a call to three methods in each step. The first one is *diffuse(dt)*, which is key to the calculations. It contains two nested for loops, that iterate over each cell of the grid. The calculations are independent of each other, and that will be our next focus in an attempt to speed up the code.

The remaining two methods, *reset()* and *updateBoundaries()*, update the Mesh with the result of the calculations. The former simply iterates over each cell in the grid. The latter performs calculations on the boundaries of the region in the problem. Thus, it has a for loop, iterating over each of the four boundaries, which should also be easily sped up.

The *VtWriter* class is responsible for writing the values in each cell to a file, one per a time step. Thus it also iterates over the cells of the mesh at each step.

Thus we recognized three areas for improvement in performance: creating the mesh, the three methods iterating over each cell of the mesh (which will all be treated in a similar manner), and most importantly the loop that advances the calculations step by step.

The results of all the tests are in subdirectory *results*, and a short index can be found in Appendix A.

II. Setting up the tests and measuring overhead

Before we can objectively measure the improvement in performance, it is essential that we assess the overhead of using OpenMP.

To begin with, we measure the time taken to create the mesh and perform the calculations of the provided *square.in* problem. To get a reading of the time taken, we use the *omp_get_wtime()* just before and after creating the mesh, and likewise at the start and end of the *Driver.run()* method. The readings can be found in the files *test_square20_xx.txt*. The average time to create the mesh is 4.8640×10^{-5}, and to run the Driver is 0.165044.

It is clear that the problem size is not sufficient to give much room for improvement. Furthermore, increasing it will aid us with the statistical analysis of the results. For this reason we have also used a variety of machines. The files mentioned earlier are recorded on a personal laptop. Similar tests were ran for comparison on the DCS machines, and the results were found to be similar (3.1258×10^{-5} average mesh time, 0.257189 average calculation time). There is also very little variance, which is why we deem a sample of 10 tests to be enough.

After some trial and error, we found that increasing the grid size to 1000×1000 and the number of steps to 100 gives us execution time of around a minute. It is a reasonable balance between obtaining meaningful results and repeating the tests. However the time taken to create the mesh was still of the same order of magnitude.

We will use the simple loop to create mesh to measure the overhead of OpenMP. To do this, we will put each loop as a section in *#pragma omp parallel sections*, and record the time through *Driver* right before and after creating the mesh. The results are recorded in files *test_square20_mesh_xx.txt*.

The average time taken to create the mesh is 0.00570391, which is 117 times more than without parallelism - 2 orders of magnitude. The difference - 0.00566 s. - is the overhead of using OpenMP.

III. Speed up

A. Mesh

As shown before, OpenMP creates a very significant overhead, so perhaps rather than asking "how much can we speed up creating the mesh?", we should work out how big the problem needs to be, to overcome the overhead.

We found that using a 10000×10000 is the maximum problem size our machines can deal with. Anything larger, even by one order of magnitude will cause integer overflow. To deal with that, we could use arrays of long long int, however it is worth considering the memory required to hold such array. Take an array of integers, each one being 32 bits. An array of size $(1 \times 10^4) \times (1 \times 10^5)$ - just 10 times larger than the problem in *square_mesh.in* - will require 32×10^9 bits, which is 4 GB. Therefore not much can be achieved here with the aid of just OpenMP.

The times have been recorded in the file *test_square10k_mesh_00.txt*. The average is 0.005053462 with parallelism. For comparison, the same test was performed with the original code. The average was 0.0002356191, with results that can be found in *test_square10k_mesh_01.txt*. The difference is smaller but still non-negligible, being of one order of magnitude, and there's more variance.

It has to be stressed that this is the best case scenario for creating the Mesh, where the problem is calculated on a square region, and each one of the two for loops has the same amount of work to do. There would be no point in parallelising this part of the program, in problems with a big difference between the loops, for example with a matrix of size 1×1000.

Thus, we won't be able to achieve any speed up in the mesh.

B. Diffusion

Another attempt we can make is to parallelise the scheme.

After a brief inspection of the *ExplicitScheme.C* and the loops in it, we conclude that there are no dependencies between the iterations. Each iteration of the nested loops calculates the value at a given cell of the Mesh, independently of the other cells. Therefore we can not worry with a critical section, which would just create an additional overhead.

To take an initial reading for the time taken to calculate the diffusion, we use *square2.in*. It is a 100×100 grid with 100000 steps. The average time taken by the Driver is 68.0825 s, and the results are recorded in the files named *test_square100_1ksteps_0x.txt*.

We use *#pragma omp parallel for private(k) schedule(static)*, which schedules the iterations of the loop between available threads - chosen automatically based on the architecture. In the case of my laptop, it picks 4 threads. The average is 89.1293 s, with the results recorded in the files *test_square100_1ksteps_1x.txt*. This is significantly worse than a sequential performance.

For comparison, we specify the number of threads, from 1 to 4, by adding *num_threads(x)* at the end of the pragma. The results are presented in the Table 1 below.

	Run 1	Run 2	Run 3	Run 4	Run 5	Average
1	73.6722	68.6994	68.2287	67.4667	68.2951	69.2724
2	80.5171	74.4021	75,1654	74.5040	77.9940	76.5163
3	81.0917	79.0574	78.0048	79.5261	78.0110	79.1382
4	97.8915	89.3469	88.6477	89.2015	90.1243	91.0424

Table 1: Comparison of Runtimes with Pragma in ExplicitScheme.C

It is clear that with this problem size, there is no point in attempting to parallelise the remaining loops in the *ExplicitScheme.C*. The runtime increases with each additional thread.

C. Driver

The most important part of the code where we can improve performance, is the loop in *Driver.C*, iterating over each step, in dt intervals, from start time to end time. We also take more readings than previously, as after some initial trial and error, we noticed that we can achieve quite significant speedup here.

Before we can add the pragma before the for loop, we need to make sure to iterate over an integer, rather than a double as it currently is. It is done by creating a new variable, i, running from 0 to $(t\_start - t\_end)/dt$ (cast into int).

While the results are looking very well, we've noticed a worrying trend: there is a little variance in the total temperature, which should stay constant. It is in the range of ± 0.03.

IV. Discussion of Methods

Something that has not been addressed in this writeup, and is beyong the scope of the project, is optimisation at compile time. The only change made to the MAKEFILE was adding debugging flags, and everything else was left as it was.

It was expected that background processes running on the machine while the code was executing, could affect the results. Hence to mitigate that, multiple readings were taken.

A note on writing the result files: to avoid any performances difference in the program, the results weren't recorded by changing the location of the output of the program in *Driver.C*. Instead, we redirected it from the terminal to a specified file through bash.

V. Conclusion

Appendix A
Index of test inputs and results

The following tests were added:

- **square1.in** - 100×100 matrix, 100 steps.
- **square_mesh.in** - 10000×10000 grid, 1 step
- **square2.in** - 100×100 matrix, 1000 steps

The following files were the first readings, without amending any code, other than to print out time taken:

- **test_square20_xx.txt** - readings taken with *square.in* input on personal laptop
- **test_square20_xxa.txt** - readings taken with *square.in* input on lab machines through SSH
- **test_square100_xx.txt** - readings taken with *square1.in* (1000×1000 grid, 100 steps) input on personal laptop
- **test_square100_xxa.txt** - as above, from the lab machines
- **test_square100_1ksteps_0x.txt** - measuring time taken by the Driver, 10000 steps.
- **test_square100_1ksteps_1x.txt** - measuring time taken by the Driver, 10000 steps, including the pragma in *ExplicitScheme.C*
- **test_square100_ythread_xx.txt** - measuring time taken by the Driver, using y threads over a single loop in *ExplicitScheme.C*

The following files are various attempts at parallelism:

- **test_square10k_mesh_00.txt** - running *square_mesh.in* with the for loops in *Mesh.C* in parallel
- **test_square10k_mesh_00.txt** - running *square_mesh.in* with the for loops in *Mesh.C* using sequential code

The following files are discarded attempts at parallelism:

- **Mesh_parallel.txt** - running the two for loops in parallel, using *# pragma omp parallel sections*. It was used to measure the overhead of using OpenMP.
- **Driver.txt** - original version of the Driver Class