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 the most timely one, however each calculation directly depends on the results of the previous iteration, and thus it cannot be parallelised. Hence we can keep the number of steps small, and focus on other loops.

We will therefore focus on all methods which iterate over the mesh, and attempt to parallelise those. The results of all the tests are in subdirectory *results*, and a short index can be found in Appendix A.

II. Initial readings

We use gmon to get the proportion of the time each function is running, and *omp_get_wtime()* to measure total time. The first three tests done, *test.txt*, *testx.txt*, and *testy.txt* are readings taken after running deqn on the three original tests provided. In the latter two, the runtimes of the functions are negligible. We will return later to problems which have a very long and narrow, rather than an almost square grid.

For now, let us focus on problems with a square region. The most costly operations are summarised in the table below, excluding standard library, which takes over 40% of the time on a square grid, and methods which have negligible runtimes. It demonstrates a comparison between running the program on a 100^2, 1000^2, and 10000^2 (with 2 steps) grid, and the difference in the percentage of time take by each method. For comparison, we added a test on a long and narrow grid.

Function	100^2	1000^2	10000^2	10×10^7
VtkWriter::writeVtk(...)	23.34	28.21	24.16	35.22
Mesh::getTotal...	16.67	15.39	6.73	5.35
Mesh::getNx()	8.34	11.54	12.58	7.51
ExplicitScheme::reset()	3.33	5.13	7.53	6.69
ExplicitScheme::reflect...	3.33	2.56	0.00	1.63
Mesh::getU0()	1.67	3.85	3.47	2,23
Mesh::getDim()	1.67	1.28	2.87	2.30
Mesh::Mesh(...)	1.67	1.28	1.29	1.34
Diffusion::init()	0.00	0.00	9.01	10.48

Table 1: Summary of most costly methods in the program.

Mesh and *ExplicitScheme* are the two classes responsible for calculations, which are also most costly. Especially in the case of *Mesh::getTotalTemperature(...)* it isn't surprising to see that it contributes to a significant proportion of the runtime, as it sums over each cell of the mesh.

As the problem size increases, the time needed to set it up is extended. Therefore on the 1000^2 grid we begin to notice the impact of methods such as *Mesh::Mesh()* and *Diffusion::init()*

It also isn't unexpected that *ExplicitScheme::reset()* takes up more time as the problem size increases. This will be our second focus in improving performance.

What is much more surprising in this table, are the three getters in *Mesh*. Closer inspection of them is in Table 2, using 1000^2 grid.

Method	total time (s)	number of calls	time per call (ms)
getNx()	0.05	42084401	0.00
getU0()	0.02	21000125	0.00
getDim()	0.01	21	0.24

Table 2: More information about the unexpectedly costly getters

As we can see, those methods take a significant proportion of the time not because of their complexity, but because of numerous calls, and so we can't do anything about it with OpenMP.

The most important part of the code is the loop in *Driver.C*, iterating over each step, in dt intervals, from start time to end time. However after inspecting the code, we notice that each iteration depends on the results from the previous one, and thus we cannot parallelise it at all. Since all other loops operate on the mesh, rather than on

the steps, we can reduce the number of steps, to speed up the testing. We will use 2 steps.

III. Speed up

A. Mesh

From Table 1 we can see that the file writer takes most time, but also it cannot be parallelised, as we want the results in a file to appear in order. Therefore we move to *Mesh::getTotalTemperature()*. It simply sums the value of each cell, and there's no dependency between cells. The only thing we will have to be careful of, is to take care of the critical section.

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.

We first add *#pragman omp parallel for private(k) schedule(static)* just before the for loop in *Mesh::getTotalTemperature()*, and also *#pragma omp critical* before summation of the temperature inside the loop. The results of this ran on the 1000^2 grid (*square1.in*). As the program runs, the results are redirected to the file. We can easily get the total run time from it, which is 153.015 s. For comparison, the sequential code runs in 116.075 s.

Instead of using a critical section, we can use reduction. Our next test is adding *#reduction(+:temperature)* to the pragma options. With that, the total time to run the program is again around 116 s.

Finally we try to change schedule to dynamic. It gives us a slightly better overall time, with 110.024 s average over 3 runs, and the results are very close together.

Nevertheless, in the gmon's output we can see that the methods now takes a much smaller proportion of the time.

Since this method is called multiple times, it would be interesting to see how big a difference it makes when the problem has more steps. We use files labelled *square1_xxsteps.in* as inputs, and compare the runtimes between the original and the parallelised program. Here's a comparison:

No. of steps	original runtime	static runtime	dynamic runtime
10	7.0223	7.12566	6.32213
20	13.0044	11.9042	11.2931
40	23.1150	23.1293	22.6300
50	29.0084	29.0960	28.9210

Table 3: Parallelising Mesh::getTotalTemperature() statically and dynamically

Although the difference isn't overwhelming, a dynamic schedule of the thread consistently performs better than the others, so we're going to keep it.

Last thing which we can do with Mesh - and which is only relevant with big problems - is parallelising the constructor. In a small problem, such as *square.in* we expect to see a lot of overhead.

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.

For comparison, we run the same code with a much larger program - such as the one in *square_mesh,in*. 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.

Gmon suggests that creating the mesh takes now about 1.03% of the runtime (1.29%-1.91% previously), which is a slight improvement. The pragma is commented out in the code, as it gives no benefits for smaller problems.

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.

B. ExplicitScheme

Similarly to *Mesh::getTotalTemperature()* above, we add the same pragma with dynamic scheduling to *ExplicitScheme::reset()*, which we identified earlier as one of the longer methods. Again, we run it a few times on *square1.in* to obtain the following average time:

C. Diffusion

IV. Discussion of Methods

Something that has not been addressed in this writeup, and is beyond the scope of the project, is optimisation at

compile time. The only change made to the MAKEFILE was adding debugging gmon flags, namely *-g* and *-pg*, 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.

V. Conclusion

Appendix A
Index of test inputs and results

The following tests were added:

- **square1.in** - 1000×1000 matrix, 20 steps.
- **square_mesh.in** - 10000×10000 grid, 1 step
- **x1.in** - 10×10^7 grid, 2 steps
- **square1_xxsteps.in** - as in *square1.in*, with a different number of steps each time

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

- **test.txt** - gmon.out output of the runtimes, using *square.in*
- **testx.txt, testy.txt** - gmon output of the runtimes, using *x.in* and *y.in* respectively
- **test1.txt** - output of gmon using *square1.in*, which is a 1000×1000 grid
- **testx1.txt** - output of gnom, using a very long and narrow grid (*x1.in*)
- **test_xxsteps_original.txt** - output of gnome, when the original code is ran on the *square1_xxsteps.in*

The following files are results of various attempts at parallelism:

- **test_xxsteps_static.txt** - code ran on the *square1_xxsteps.in* with static scheduling
- **test_xxsteps.txt**- code ran on the *square1_xxsteps.in* with dynamic scheduling
- **test_square1_scheme_testx.txt** - parallelising ExplicitScheme with dynamic schedule

For the above files, we dumped the stack to a file. It includes readings of time taken to create the Mesh and for overall calculations, more as a reference. In our considerations, we used times from gmon.

- **dump_mesh_plain_1.txt** - running the original programwith *square_mesh.in*
- **dump_mesh_1loop_x.txt** - running the program with a single pragma in *Mesh::getTotalTemperature()*, on a 1000^2 grid *square_mesh.in*
- **dump_xxsteps_original.txt**
- **dump_xxsteps_static.txt**
- **dump_xxsteps.txt**
- **dump_square20_mesh_xx.txt** - times taken to create a mesh with and without parallel sections, showing the overhead of OpenMP
- **test_square10k_mesh_xx.txt** – as above, with a large problem
- **dump_square1_scheme_testx.txt** - dump of the program with parallelism in the *ExplicitScheme::reset()* method

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