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.

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 achieving better performance.

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. However, the loops are very simple 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 updating the region.

II. 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.

III. Speed up

A. Mesh

B. Diffusion

IV. Evaluation

V. Conclusion