High Performance Programming and Systems

Assignment Five

The Heat Equation in Two Dimensions

Set: *8th of January 2023*
Due: *14th of January 2023 @ 23:59 CEST*

Synopsis:
Write a parallel implementation of a simulation of the heat equation in two dimensions.

1 Introduction

This is the fifth of five assignments in the *High Performance Parallel Systems* course. For this assignment you will be implementing and parallelising the heat equation in two dimensions.

The heat equation is a simulation of heat dispersal in a material. The idea is that the heat stabilizes over time and we can simulate the heat dispersal in discrete timesteps using a stencil approach. In this assignment we will use a method called successive over-relaxation (SoR). While we apply the method exclusively to simulating heat, the general approach has other uses as well.

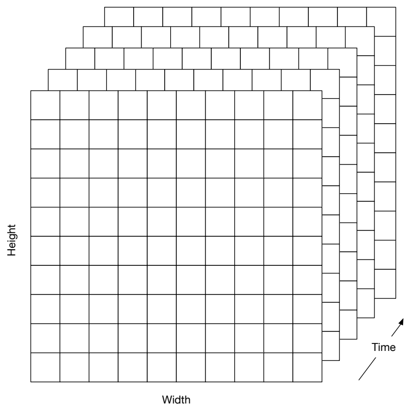
To be more specific, we investigate a 2D surface and divide the surface into cells, in which we store the current temperature. To complete a timestep, we need to read the values of the four neighbour cells[1], as well as the cell itself. The new temperature of the cell is then the average of the values.

Or put into a formula where $temp^{t}_{(i,j)}$ is the value of the cell at position (i, j) at time step t:

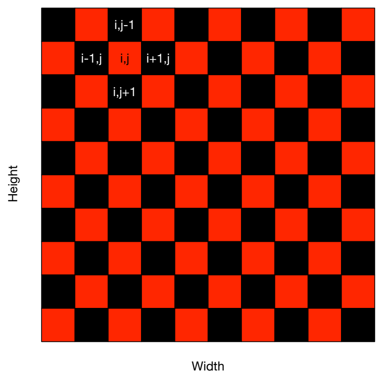
$$temp^{t}_{(i,j)} = \frac{temp^{t-1}_{(i,j)} + temp^{t-1}_{(i-1,j)} + temp^{t-1}_{(i+1,j)} + temp^{t-1}_{(i,j-1)} + temp^{t-1}_{(i,j+1)}}{5}$$

Observing the formula, we can see that we need $i \times j$ cells in the grid, and t grids, as shown in fig. 1a. However, since we are (mostly) interested

[1]In practice we might use a larger halo, but for this simulation, the direct neighbors suffice.



(a) Grids for timesteps.



(b) Red-black division of cells.

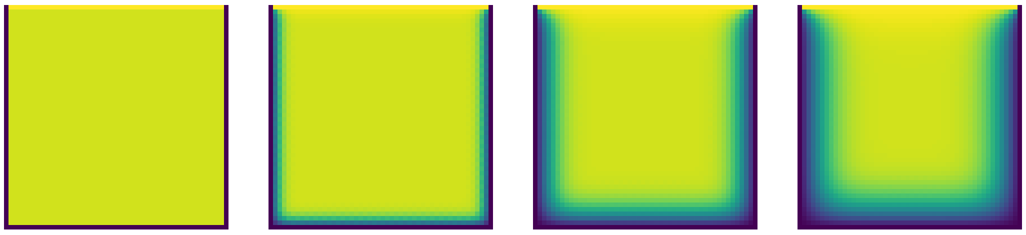
Figure 1: Simulation discretization and setup

in the final grid, we can implement this with a standard buffer method so we only have 2 grids (for t and $t-1$ respectively). After each simulated timestep we simply swap the two (i.e. t becomes $t-1$ in the next step, and the other grid becomes the target for the new t).

We can improve the memory usage further with a scheme known as red-black successive over-relaxation. In this scheme we divide the grid into into red and black cells, in a checkerboard pattern, such that a red cell will have 4 black neighbours (east, west, north, south) and vice versa for each black cell.

With the red-black approach we only update half the grid in each iteration, but since we only read the *other* color cells, we can do so without guarding against concurrent access. In each iteration we then switch the color we update. The division of items is shown in fig. 1b.

As is often the case, the borders present a problem, in that there is nothing beyond the borders, so we cannot calculate a meaningful new value for the border cells. The handling of boundary conditions will often have significant impacts on the results of the simulation, but to keep things simple, we just *ignore* the border cells when updating. In other words, the values assigned to the border cells will remain the same throughout the simulation.



(a) Initial values. (b) After 10 steps. (c) After 50 steps. (d) After 100 steps.

Figure 2: Visualisation of the heat equation steps

2 Implementation

To get you started on the assignment we have provided a Python implementation of the code (`heat2d.py`) that you need to *port*[2] to the C programming language and then parallelize. Do this in the file `heat-equation.c`.

The file `heat-equation.c` can be compiled in two ways: to an executable `heat-equation`, and to a shared object `heat-equation.so`, that can be loaded by a Python program. The former is useful for testing, and the latter is a common technique for writing scientific programs (as discussed in the course material). Both are defined as targets in the handed-out Makefile.

Running the `heat-equation` executable as

```
$ ./heat-equation N M S out.bmp
```

should produce a visualisation of simulating an `N` $\times$ `M` grid for `S` iterations and produces an image in `out.bmp`, similar to the ones in fig. 2.

This is a typical workflow for a high-performance application, where a scientist develops a model in Python or MATLAB and then needs it to run with a larger dataset and much faster. To make it slightly easier for you, the Python program is implemented without using slicing and based on a single flat array that is updated repeatedly. Also, each function only operates on its arguments[3], which makes it simpler to implement in C.

[2]In this context, *porting* means to translate the program to run within another programming environment.

[3]There are no global or shared variables, except the data array and no composite data structures.

2.1 What is required of your implementation

Your task is to first write a sequential program that matches the Python program, then to parallelize the sequential code. You must use the OpenMP programming model to do so, enabling the code to utilize multiple compute cores. Both computation of the the *delta* value as well as the red-black update should be parallelized. The file `heat-equation.c` contains skeleton code that you should modify. All places you need to fill in are marked with `assert(0);`. You may also wish to modify `heat2d.py` to add timing code.

2.2 Using MODI

When you are happy with your design, you can test it on a larger system. UCPH provides access to the MODI queue based system where each node has 256 GiB of memory and two 32-core CPUs. Using the MODI system, you will be able to simulate larger systems than what your laptop can handle.

Note: working with a queue based system means there will be a delay between the submission of your experiment and receiving the result. As we approach the hand-in date, the load on the system will likely grow, increasing the delay. We suggest you take this into account when planning when to run the experiments.

Since each MODI instance has two 32-core CPUs, it would make sense to choose a grid and iteration size that works well for one to 64 cores (i.e. if it takes 1 second on a single core, it will not take $\frac{1}{64}$s with 64 cores). Then submit jobs with different thread counts and plot the result times in a speedup graph, where the x-axis is the number of cores, and the y-axis is the time spent.

If you have time for it, you can consider scaling the workload and keeping the number of cores constant, similar to Gustafson-scaling.

Running on MODI

To help you run the code on MODI, the handout contains two helper files, `work.sh` and `submit-all.sh`. These two files are meant to help you run your code on MODI, and are not meant for use locally. The use of MODI and the helper files is described here:

`https://github.com/diku-dk/hpps-e2021-pub/blob/master/modi-guide/README.md`

And the specifications for MODI itself are here:
`https://erda.dk/public/MODI-user-guide.pdf`.

2.3 Validation help

When writing the program in C, it can be difficult to validate that the program works as expected. Due to the floating point instability and resolution issues you have worked with earlier, you cannot simply look at the values and check that they are identical.

To assist in visual validation, we provide a helper function that can create a bitmap (`.bmp`) file from an array of floats. Using this method, you can create step-by-step images of your values and visually verify that the code works as expected.

The code handout also contains code for performing an animated visualisation of the simulation. You don't need to understand how these work, and you don't need to modify them, but they can perhaps help with debugging—or at least provide motivation. The implementation is contained in two files:

`server.py` implements a web server that listens on port 8080 on your local machine, and which imports the `heat2d.py` module to perform the simulation.

`view.html` is a simple HTML page that contains JavaScript that communicates with the web server.

You use it by running the server part with `python3 server.py` and then opening `http://localhost:8080/view.html` in a browser.

You can edit `server.py` to modify the size of the simulated grid, the number of steps per frame, and whether to use your C-based implementation of the model. **Note:** To make the image refreshing reliable after you've implemented the C version, you will almost certainly have to increase the width and height.

2.4 A note on timings

Any decent analysis of a parallel system will include timings of the sequential and parallel implementation. However, care should be taken when deciding what you time. Timing the entire program runtime may be useful, but so could only timing smaller sections to illustrate exactly what speedup has occured in the parallelised sections. Make sure you justify why that timing is informative, whatever timings you decide are important to present.

3 The structure of your report

Your report must be structured exactly as follows:

Introduction: Briefly mention very general concerns, your own estimation of the quality of your solution, whether it is fully functional, and possibly how to run your tests.

Sections answering the following specific questions:

a) What is the memory access pattern of each thread? Does it exhibit good locality?

b) How many floating point operations does your code perform per memory access (e.g. FLOPs per byte). **Hint:** The problem is memory bound.

c) What fraction of the code can be parallelized (i.e. the `p` value in Amdahls law)? Do note that you cannot calculate this from your speedup.

d) What is the speedup of your C version, running on a single thread, compared to the original Python version?

e) What is the speedup of your C version running with various numbers of threads to the C version running single-threaded? Explain the results. **Hint:** use `OMP_NUM_THREADS` to tweak how many threads are used.

It is up to you to decide on proper workloads (i.e. input sizes) that can provide good answers to these questions. Make sure to report the workloads you use.

All else being equal, **a short report is a good report**.

4 Deliverables for This Assignment

You are encouraged to work in groups with 3 people for this assignment. We strongly encourage you to participate in the lab sessions, where we will use time to discuss the design, implementation etc.

You should submit the following items:

- A single PDF file, A4 size, no more than 5 pages describing each item from report section above
- A single ZIP/tar.gz file with all code relevant to the implementation

5 Handing In Your Assignment

You will be handing this assignment in using Absalon. Try not to hand in your files at the very last-minute, in case the rest of the students stage a DDoS attack on Absalon at the exact moment you are trying to submit. **Do not email us your assignments unless we expressly ask you to do so**.

6 Assessment

You will get written qualitative feedback, and points from zero to 4. There are no resubmissions, so please hand in what you managed to develop, even if you have not solved the assignment completely. **Note that this assignment does not count towards exam qualification.**