Assignment 2

Objectives

In this assignment, we continue to explore parallel programming in OpenMP. We will apply the different software optimizations we learned in the lectures. First, we will cover true and false sharing and then optimize algorithms operating on two-dimensional arrays.

Part 1 – Optimizing true/false sharing

In this part, you will optimize true/false sharing similar to the example presented in the lecture. In the handout, you will find `sharing.c` that implements a single-threaded version of a simple histogram algorithm that repeatedly generates random numbers and updates the histogram based on the obtained value. Your job is to parallelize the algorithm using OpenMP, study the effects of true/false sharing in the histogram using 100M operations, and optimize them using the techniques explained in the lectures. Here is the expected output format:

```
# /bin/bash
$ ./sharing 4 100000000 32
Using 4 threads: 100000000 operations completed in 0.29s.
```

Part 2 – Heat Transfer Simulation

In this part, you will implement a heat transfer simulation. Assume we have a square surface with a heat generator at its center and a heat sink at its edges. The hot core transmits energy and remains at maximum temperature. The heat sink temperature remains zero. Initially, all other points on the surface are at zero degrees. This state is illustrated in Figure 1. Over time, the temperature changes at each point on the surface. Figure 2 illustrates what the temperature distribution will look like after some time.

We provided a reference single-threaded implementation of the heat transfer algorithm in `heatmap.c`. The algorithm iteratively updates the temperature grid, with each element's temperature being calculated from a linear combination of the temperature of its neighbors. You will write a parallel program using OpenMP that simulates the heat transfer example presented above and optimizes it using the various software optimization techniques you learned in the fourth lecture. The program will operate on a variable-sized square-shaped two-dimensional array and simulate for `N` iterations. For simplicity, the side length of the square is always even. The program must write the value of each element in the two-dimensional array in a comma-separated values (CSV) file, which is later checked for correctness. We already provided all the code necessary to generate the CSV file. The program must also report the time it took to run the algorithm for N iterations to measure performance. You cannot optimize the overall algorithm based on the symmetry of the heatmap, i.e. the overall number of calculations should remain the same as in the single-threaded version.

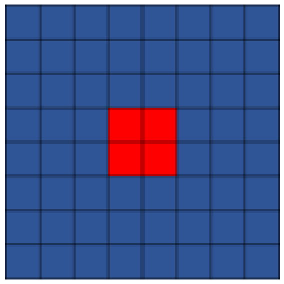


Figure 1: Initial state of the surface

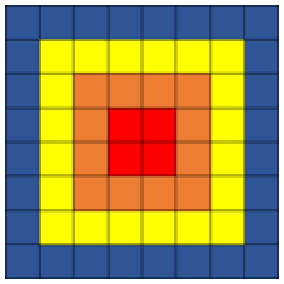
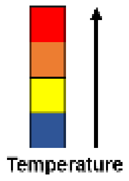


Figure 2: The surface after a period of time



An example of running the program and the output it produces is shown below:

```
# /bin/bash
$ ./heatmap 4 1000 5000 output.csv
Running the algorithm with 4 threads on 1000 by 1000 array for
5000 iterations took 10.73 seconds
$ls
output.csv ...
```

Development Infrastructure

On the course's Moodle page, we have provided the file `A2.zip`, which contains the following:

- `sharing.c` – the file containing the simple histogram implementation.
- `heatmap.c` – the file containing the `main()` function that will call your heatmap code.
- `utility.h` – which contains a set of helper functions:
  - `set_clock` & `elapsed_time`: to measure the start and end of execution.
  - `init`: initializes the two-dimensional array's values and sets the initial state.
  - `save`: saves the two-dimensional array in a CSV file.
- `algorithm.c` – which contains the `simulate` function. This function applies the algorithm in question on an input two-dimensional array for N iterations. It updates the values of the array accordingly and writes them to an output two-dimensional array. It is useful to interchange the arrays between cycles, using one to hold the previous iteration's output and the other to hold the output of the current iteration. **You must parallelize and optimize this function.**
- `Makefile` – which compiles your code; please refer to Assignment 1 to make sure the `Makefile` is suitable for your environment.
- `execute.sh` – A sample script to submit jobs to the SCITAS cluster.

Using the environment provided will guarantee that your program will comply with all specifications mentioned.

Deliverables

To submit your code and your report, create a tarball archive **(this is the only accepted format!)** called `a2_<yourGroupID>.tgz` and upload it on Moodle. Your file must contain:

- A parallelized and optimized version of `sharing.c`.
- A parallelized and optimized version of `algorithm.c`.
- A report, with the name `a2_<yourGroupID>.pdf`.

Report

In the report, we expect you to perform the following tasks:

Part 1

Explain how parallelizing the given algorithm impacted performance. Was there any true/false sharing? If yes, how did you optimize it? Report the performance differences before and after these optimizations for thread count {1,2,4,8} and 100M operations. Finally, explain how does the histogram size affects the performance of the unoptimized algorithm.

Part 2

1- Explain how you parallelized and optimized `simulate`. List the optimizations you applied and the reasons you chose to apply them. Note that the program is trivially parallelizable, so we are more interested in the memory optimizations than in parallelizing the program itself. Identify the different ways of splitting the work among threads and which result in more data locality within threads. Also, identify whether there is an issue with load balancing when dividing the work.

2- Report how much each optimization improves performance and explain the result obtained. Design and run experiments that isolate the effect of each optimization/thread organization you tried out and then report the results.

3- Present the execution time you measured in a graph for the following set of thread counts {1,2,4,8,16} running for 100 iterations and a side length of 10000.

The report should be as precise as possible, addressing the questions above. Keep the descriptions of the algorithm implemented and of your reasoning in parallelizing the program concise. A regular report should not be much longer than 2-3 pages.

Grading

The code will be graded automatically by a script and checked for plagiarism. The script will check how the running time scales with the number of threads and if the results returned are consistent with what was expected. Plagiarized code will receive 0. We will escalate plagiarism cases to the student section. The reports will be read and graded by humans and checked for plagiarism automatically. The grade breakdown is as follows:

• Correctness:	50%
• Report:	50%