Assignment 2

Objectives

In this assignment, we continue to explore parallel programming in OpenMP. We will apply different software optimizations we learned to algorithms operating on two-dimensional arrays.

Background

Many scientific (ex: heat transfer), image processing (ex: edge detection), and cryptographic algorithms operate on two-dimensional arrays, where the algorithm updates each element in the array based on the state of its neighbors. The algorithm usually runs for N iterations. In each iteration, n, the value of each element in the array is updated based on the value of a fixed set of its neighbors (and its own value) from the previous iteration, n-1.

In a simple example, consider an algorithm that updates each element as the average of its value and all coinciding neighbors. Assume that such an algorithm is operating on the two-dimensional array shown in Figure 1. In each iteration, the element (3,3) is updated as the average of elements {(2,2), (2,3), (2,4), (3,2), (3,3), (3,4), (4,2), (4,3), (4,4)} from the previous iteration. This set is marked in red. All elements in the array are updated based on their neighbors’ values in a similar fashion.

	0	1	2	3	4	5	6	7
0	2.7	4.6	7.2	7.1	2.3	5.2	4.8	8.5
1	1.9	6.6	8.3	2.7	6.8	2.0	1.7	8.7
2	4.5	2.1	2.8	1.2	8.4	9.7	2.4	1.1
3	8.9	4.3	5.6	1.9	9.7	2.8	1.0	5.2
4	5.0	7.7	7.9	6.4	2.6	8.0	2.7	8.6
5	7.3	1.2	4.3	1.6	1.5	4.2	6.1	3.0
6	6.5	1.0	4.3	4.0	5.6	4.9	9.5	7.0
7	2.6	4.9	8.2	6.3	1.2	8.2	4.4	7.9

Figure 1: An example of simple algorithm operating on a two-dimensional array

Assignment

Let us consider a heat transfer example. 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 is illustrated in figure 2. Over time, the temperature changes at each point on the surface in accordance to the algorithm discussed in the background section. Figure 3 illustrates how the temperature will appear after a period of time.

In this assignment, you will write a parallel program in OpenMP that simulates the heat transfer example presented above and optimizes it using the various software optimization techniques you learned in Lecture 4. The program will operate on a variable sized square-shaped two-dimensional array and will run the simulation for N iterations. For simplicity, the length of the array is always even. The program must write the value of each element in the two-dimensional array in a comma-separated values (csv) file, to check for correctness. It must also report the time it takes to run the algorithm for N iterations to measure performance.

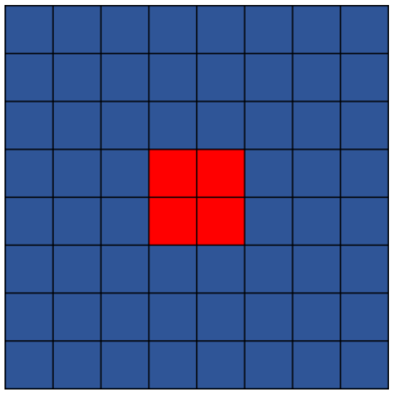
Figure 3: Initial state of the surface

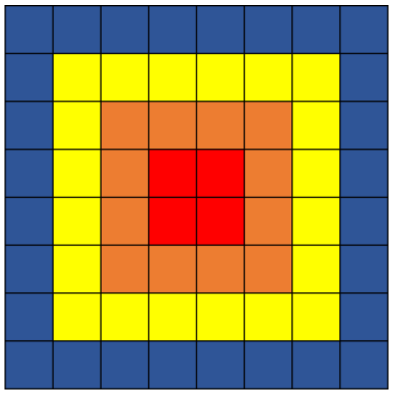
Figure 2: The surface after a period of time

Temperature

Development Infrastructure

On the course's Moodle page, we have provided the tarball A2.tgz, which contains the following files:

- `assignment2.c` – the file containing the `main()` function that will call your code.
- `utility.h` – which contains a set of helper functions that the environment uses:
  - `set_clock` & `elapsed_time`: to measure the start and end of execution.
  - `init`: initializes the two-dimensional array's values.
  - `save`: saves the two-dimensional array in a csv file.
- `algorithm.c` – which contains `simulate`. 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 output of the previous iteration and the other to hold the output of the current iteration.
  You must parallelize and optimize this function.
- `Makefile` – which compiles the environment using the current gcc version; 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.

The environment will operate as follows:

1. The program file is called `assignment2.c`.
2. Compiling the program using the provided `Makefile` produces an `assignment2` binary.
3. The input arguments are, in this specified order:
   a. Number of threads
   b. Side length
   c. Number of iterations
   d. Output file name
4. The program will save the output, after running N iterations, in `<OutputFileName>`

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

```
# /bin/bash

$ ./assignment2 4 100000 500000 outputx.csv

Running the algorithm with 4 threads on 100000 by 100000 array for
500000 iteration takes 10.73 seconds

$ls

outputx.csv ...
```

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

Deliverables

You need to turn in a tarball archive called `a2_<yourSCIPER>.tgz` to the Moodle link specified for Assignment 2, containing:

- A parallelized and optimized version of `algorithm.c`.
- A report, with the name `a2_<yourSCIPER>.pdf`, that complies to the following description.

The tarball should not contain a directory. Create your tarball using:

```
$tar -cvzf a2_<yourSCIPER>.tgz /path/to/a2_<yourSCIPER>.pdf /path/to/
algorithm.c
```

Report

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

1- Explain how you parallelized and optimized `array_process`. List the optimizations you applied and the reason you chose to apply them.

2- Report how much each optimization improves performance and explain the result obtained.

3- Present the execution time you measured for the following set of thread counts {1,2,4,8,16} running for 500 iterations and side length of 10000.