University of Tromsø

Assignment 2 - Successive Over-Relaxation

INF-3201

Camilla Stormoen

Department of Computer Science

October 14, 2016

1 Introduction

This report describes the implementation of three techniques for doing Successive Over-Relaxation by using the Frost Trap Algorithm.

1.1 Requirements

- First requirement is to implement and parallelize three techniques for doing Successive Over-Relaxation.
  1. Simple
  2. Red-black Ordering
  3. Double Buffering
- Next requirement is to use OpenMP.
- Last requirement was to evaluate and analyze the implementation and different techniques.

2 Technical Background

2.1 Successive Over-Relaxation

Successive over relaxation, also called SOR, is a method of Gauss-Seidel for solving a linear system of equation. Successive over relaxation is used to calculate the temperature for each point for every iteration based on its neighbour,as shown in Figure 2.[1]

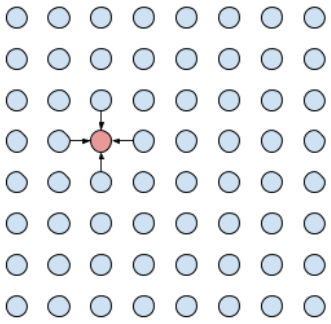

Figure 1: Shows each point and their neighbors.

2.2 OpenMP

OpenMP is a high-level parallelism language in Fortran or C/C++ that is performant, productive and portable. OpenMP use shared memory.

2.3 Uvrocks and Cluster

A computer cluster consists of connected computers that work together, but they can be viewed as a single system. Computer clusters have each node set to perform the same task, controlled and scheduled by the software.[2]

Uvrocks is a small cluster at UiT.

3 Design

3.1 Simple SOR Algorithm

The simple SOR algorithm is a basic non deterministic version of the Frost Trap. It will iterate through each column and calculate new pixels. Each iteration will be parallel and will gather the new values to get the total sum of each calculations.

3.2 Red-black SOR Algorithm

The red-black SOR algortihm is determenistic and should return the same result with different computing environment. The red-black algortihm is based on the simple SOR algortihm, but is improved by dividing the dots into two different colors, red and black. A red dot is updated only by looking at the black dots and a black dot is updated only by looking at the red dots, as Figure 2 shows. Each iteration for the red and black dots will be executed in parallel in the same way as the simple solution.

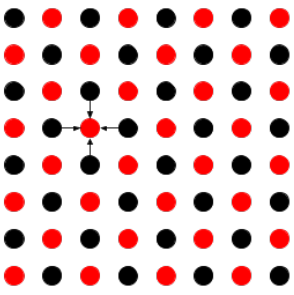

Figure 2: Shows each point and their neighbors.

3.3 Double Buffering

Double buffering is also a deterministic algorithm and also based on the simple SOR algorithm. It works by having two copies of the system, e.g. A and B. These copies calculates and updates using each others values. The values on copy A is updated by using the values from copy B and in next iteration updating the values on copy A with the values from the previous updated copy B. Copy A and Copy B is divided in the same way as the red- and black dots, and they run in parallel the same way as the two other solutions.

4 Implementation

The implementation is written in C language and run on a cluster (uvrocks) at UiT, executed using the Linux terminal from a computer. The testing and profiling is provided from the source-code and matplotlib. The machine/node used is the *"compute-1-8" (figure 3 and figure 4)* with 4 cores, 8 processors and 12GB RAM and *"compute-3-35" (figure 5 and figure 6)* with 2 cores, 2 processors and 8GB RAM.

5 Discussion

The program is built with three different approaches, however how to parallize the different versions are pretty similar. By using the OpenMP library and using both private and reduction to filter the parallelization. The private variable is used to make each thread to create a private copy of the a variable name, e.g. *private(x)*, which iterate through the width of the picture.
The reduction variable should also be used for every solution to collect the sum all the conputations, e.g. *reduction(+: delta)*.

5.1 Evaluation

The program is executed when iterating through both rows and columns. As shown in Figure 3, the double buffer will use most time when iterating through rows while it will have the fastest iteration time, around 2-2.5 seconds, when iterating through columns as shown in Figure 4.

A reason for this may be that the cache is stored in columns, so when iterating through columns it does not need to jump in memory or use a long time to find what it is looking for. However, iterating through rows might be jumping and use more time to find what it is looking for because it is iterating in a horizontal way and not vertical.

As Figure 4 also shows is that execution time is limited to number of threads. It also shows that when running on maximum 8 threads, the execution time will decrease with an increasing number of threads. When number of threads are over the maximum number of threads, it will be a overhead for the other threads because now they have to share the threads and schedule tasks between each other. This can also be seen in Figure 5 and Figure 6 where the best performance is when number of threads are 2, which is also the maximum number of threads. Performance for execution time and number of threads are limited to the hardware of the computer.

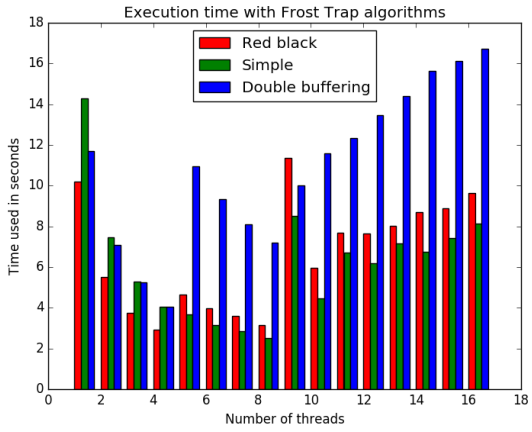

Figure 3: Performance iterating rows on 4 cores and 8 processors

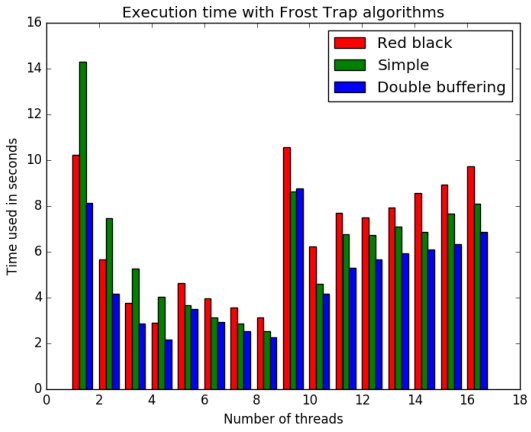

Figure 4: Performance iterating columns on 4 cores and 8 processors

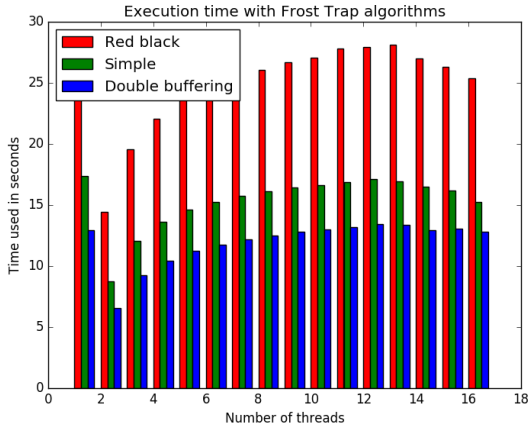

Figure 5: Performance iterating rows on 2 cores and 2 processors

Figure 6: Performance iterating columns 2 cores and 2 processors

Figure 5 and Figure 6 shows the performance of iterating with 2 threads, which will lead to a lot of overhead when scheduling tasks around, especially with the red-black implementation since it has two buffers and two iterations to compute.

Different behaviors

In shared memory will the processors share main memory, but have their own cache. In the simple solution where every computation is put in the main memory and changes all the time, must each cache be updated when a value is updated. This will lead to more time spent on going all the way to main memory to fetch a value than only going to the cache. However, the simple solution would performe pretty good in average because the pixels in each iteration will not interfere with their neighbors and the pixel values don't need to be updated.

The red-black solution is overall the slowest solution looking at the graphs. This solution have iterations divided into black and red dots, as seen in Figure 2. The reason that this implementation may take longer time is because of more overhead than e.g. the simple solution as a result of dividing the computations into two different parts.

Double buffering seems to be the best solution when running on 8 threads and iterating through columns. A reason why this solution is faster than the others, is that it use two different buffers. It will read from one buffer and write to another. This will reduce the chache coherence, as well as the process only write to the same buffer and read from the same buffer.

6 Conclusion

This report concludes the implementation and design of a Successive Over-Relaxation, using three different versions of the Frost Trap algorithm for calculating pixels. Aswell as discussing the performance of the system in a specified range of threads and their behavior, where the results shows that execution time is limited by the number of threads.

References

[1] Wikipedia, "Successive over-relaxation — wikipedia, the free encyclopedia," 2016. [Online; accessed 12-June-2016].

[2] Wikipedia, "Computer cluster — wikipedia, the free encyclopedia," 2015. [Online; accessed 12-September-2015].