Project Report

SF2568 - Parallel Computations for Large Scale Problems
Professor: Michael Hanke

Abgeiba Isunza (ayin@kth.se) - Sergio Liberman (liberman@kth.se)

December 16, 2019

1 Introduction

The objective of this project is to replicate an image using the least amount of semi-transparent colored polygons. To solve this problem we will follow Roger Johansson's [1] solution and implement a Genetic Algorithm (GA) that will adjust the position, color, number of vertices and size of a set of polygons according to a loss function that computes the similarity between the objective image and the approximated image. Furthermore, we will improve the total processing time of the algorithm by distributing the workload in multiple processes on particularly heavy parts of it. Finally, we will discuss and analyse the performance of the distributed and not distributed algorithms in theory and experimentally.

2 Methodology

The methodology consist of the following steps:

1. Program and evaluate the algorithm's processing time on a strictly sequential manner.
2. Select the parts of the algorithm that required the most amount of processing and design an alternative for multiple processes.
3. Evaluate and compare the performance in time of the sequential and parallel algorithms.

2.1 Algorithm

The algorithm starts by initializing the parameters of the approximated image from the input(target) image such as the width, height, number of channels, and the parameters of the GA, such as initial number of polygons and size of the population. Then, an initial population is formed by N individuals. Each individual is a candidate image that contains a

[1] Johansson, R. (2008). Genetic Programming: Evolution of Mona Lisa. Retrieved from https://rogerjohansson.blog/2008/12/07/genetic-programming-evolution-of-mona-lisa/

DNA represented by a set of polygons. In our test, the population is set to three individuals initialized with one randomly colored and positioned polygon each. The size, color, position, and amount of vertices of each polygon as well as the number of polygons in the DNA changes as the algorithm evolves.
For each epoch the algorithm generates the image of each individual given by the set of polygons. Then, the resulting image of the individual is compared pixel by pixel (in each RGBα channel) with the target image. The relative difference of both images is set as the fitness value of the individual.
The "best individual" is chosen as the one with the highest fitness (lowest difference) and it composes the first element of the population for the following epoch (the Next_Generation). To create the rest of the individuals for the Next_Generation, $N-1$ individuals from the current population are randomly selected conditioned on their fitness (rank selection). With a probability of 30% each of the selected individuals undergoes a two point crossover, and with a probability of 70% one polygon of their DNA is mutated. During the crossover a new individual is formed by combining parts of the DNA from the previously selected individuals, while the mutation consists of randomly changing the parameters of a chosen polygon as well as the possibility to remove or add a polygon to the DNA. The resulting individuals are added to the Next_Generation that will be the starting population of the following epoch.
After the algorithm reaches the maximum amount of epochs specified by the user, the image from the individual with the best fitness is returned. The pseudo-code of the described algorithm is shown in Algorithm (1).

```
Data: $Target\_image, Max\_Epochs$
Result: $Best\_image\_approximation$
begin
    $Initialization\ Parameters()$;
    $Initialization\ Population()$;
    for $epoch\ in\ Max\_Epochs$ do
        for $I_n\ in\ Population$ do
            $Canvas\_I_n = draw\_canvas(I_n)$;
            $Fitness[n] = \Delta(\ Target\_image - Canvas\_I_n)$;
        end
        $Best\_ind \longleftarrow Population[\text{argmax}_n\, Fitness]$;
        $Next\_Generation[0] \longleftarrow Best\_ind$;
        $Parents \longleftarrow []$;
        for $n = 1\ to\ |Population|$ do
            $Parents[n] \longleftarrow fps()$;
        end
        $New\_inds \longleftarrow Crossover\&Mutate(Parents)$;
        $Next\_Generation.Add(New\_inds)$;
    end
    $Best\_image\_approximation \longleftarrow Best\_ind$;
end
```

Algorithm 1: Applied Genetic Algorithm

A profiling of the code was done to identify the functions that took most of the computation time, and therefore to find out which functions should be parallelised. For this, the `gprof` utility was used on the sequential implementation. The profiling showed that two functions of the implementation concentrated 85.71% of the total time consumed by the program. The two functions that took 72.72% and 12.88% of the total time were `draw_canvas` and the evaluation of the fitness (`diff`), respectively.

These functions resulted to be very scalable to be applied in a parallel manner. A distributed memory version was very easily achieved thanks to the high level MPI functions broadcast, scatter, reduce and gather. A non blocking version of these methods was used given that some of the operations could be performed before actually needing the image being received in its proper destination.

2.2 Implementation details

2.2.1 Coordination of workers

For the coordination of the workers with the root process, a broadcast was implemented which would "wake" them up when an execution point was reached in which they were needed. This broadcast signified increasing the time in $t_{comm} + t_{data} \times \log_2 P$ every time a worker was awaken.

2.2.2 Implementation of `draw_canvas`

The sequential `draw_canvas` function was implemented using the high level graphics processing library OpenCV[2]. It was chosen because of its high robustness and speed. A less sophisticated way of drawing was previously attempted and it is included in the code (named `draw_CPU`), but in the end it was not used because its performance was very poor compared to that of OpenCV.

2.2.3 Sending/retrieval of Individuals

The sending and reception of the individual to be drawn, including its points and colours which determine the polygons to be evaluated later on, was implemented with a serialization of the Individual object (an implementation for the easier manipulation of polygons and the building block of the Populations, a higher level of abstraction) and an additional broadcast for its distribution to the rest of the processes.

2.3 Theoretical performance estimation

In the following performance analyses, the following constants were considered:

- t_a: time of one operation in CPU.
- t_{data}: time for sending 1 byte.

[2]This library was used also for allowing a robust and fast way of loading images from the hard drive and for debugging purposes.

- t_w: time of writing one byte to memory.
- $M = W \times H \times n_{channels}$: data size of one image.
- P : total number of processes.
- $t_{startup}$: latency of establishing communication

2.3.1 `draw_canvas()`

Sequential version

The steps involving the sequential drawing are the following:

1. Initialization of the canvas with zeros (M assignations)
2. Cloning of canvas ($n_{polygons} \times M$ assignations)
3. Creation of CV colour variable ($n_{polygons} \times 3$ assignations)
4. Creation of CV point variable ($n_{polygons} \times 3 \times 2$ assignations)
5. Fill polygon (proportional to M assignations)
6. Add two separately weighted images (proportional to M operations)

Total time $t_s = M \times t_w + (M + 3 + 3 \times 2) n_{polygons}\ t_w + O(M)t_w + O(M)t_a$

Parallel version

The main difference between the serial and the parallel implementations are in the heights of the drawn canvases. Each process is responsible of initializing and drawing an image of length M/P. After processing it, all processes send (with the gather instruction) their image to the root process, which reunites them and returns the full image. For this analysis we assume that H is divisible by P, although the general case was implemented by sending the process $P-1$ the residual consisting in the final lines of the image.

The other difference of this function with the sequential version is the need to send to every process the current individual for its subsequent plotting into the canvas. This was done through a serialization using a broadcast, as it is explained in the section 2.2.3.

The parallel time spent per process when the number of processes is P is the following:

$$T_p = M/P \times t_w + (M/P + 3 + 3 \times 2) n_{polygons}\ t_w + O(M/P)t_w + O(M/P)t_a + t_{comm}$$

where $t_{comm} = \underbrace{t_{startup} + 8t_{data} \log_2 P}_{\text{wake up worker}} + \overbrace{t_{startup} + t_{data}(1 + 10n_{polygons})}^{\text{individual}} + \underbrace{t_{startup} + t_{data} \log_2 P}_{\text{gather}}$.

2.3.2 `diff()`

Sequential version

The steps involving the similarity measure between the drawn canvas and the original image are the following:

1. Compute M (3 operations)
2. Compute the difference ($1 \times M$ operations)
3. Compute the absolute value ($1 \times M$ operations)
4. Sum the absolute values ($1 \times M$ operations)
5. Calculate the relative similarity score (3 operations)

The total time of the sequential time is $t_s = (3 + 3M + 3)t_a$.

Parallel version

The parallel version is an embarrassingly parallelisable algorithm since the P processes receive a part of the canvas of which they calculate the differences from. Each process then sends to process root their sum, which reduces it. This process is then responsible to do the similarity computation from the sum and the previously computed value M. The total time of the parallel `diff` (per process) is $T_P = \overbrace{(3M/P + 8)t_a}^{\text{local computation}} + \underbrace{t_{startup} \times 3 + t_{data} \times (8) * \log_2 P}_{\text{communication}}$.

3 Results

The expected speedups for both `draw_canvas` and `diff` are proportional to P, at least when the dimensions of the image M tend to be very big.

$S_p = \frac{t_s}{T_p} \approx \lim_{M\to\infty} \frac{O(M)}{O(\frac{M}{P})+O(\log_2 P)}$

Empirically, it was determined that the computation time for different process numbers is that which is shown in the figure 1. The curve's pace of decrease throughout the x axis is dramatic. Therefore, it can be implied that even though some of the code, namely the mutations and sampling of next generations remain being serial, the parallelised code really represents most of its running time. This results justifies the decision of parallelising just these two functions out of the whole algorithm.

The curve was obtained by setting the number of epochs to $10,000$ which, although insufficient for a good reconstruction (as shown in the figure 2a), is enough for understanding the benefits of parallelising in this case: the serial implementation would have been too heavy for achieving high amounts of iterations efficiently. As an exemplification of this, the figure 2b was achieved in little more than 20 minutes with 48 processes (2 nodes of Tegner), while serially it would have taken more than 20 hours!

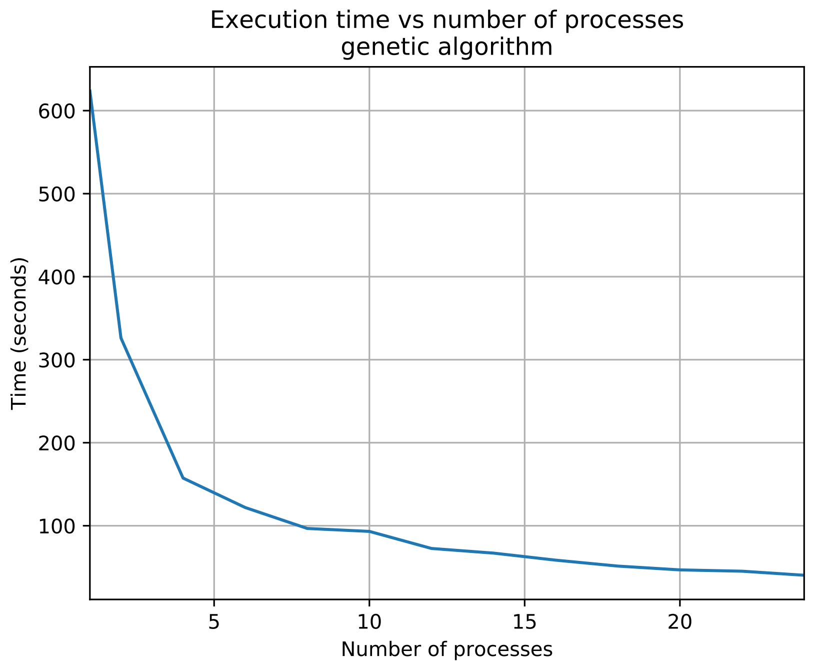

Figure 1: Time spent for the complete program when running the parallel code in multiple cores

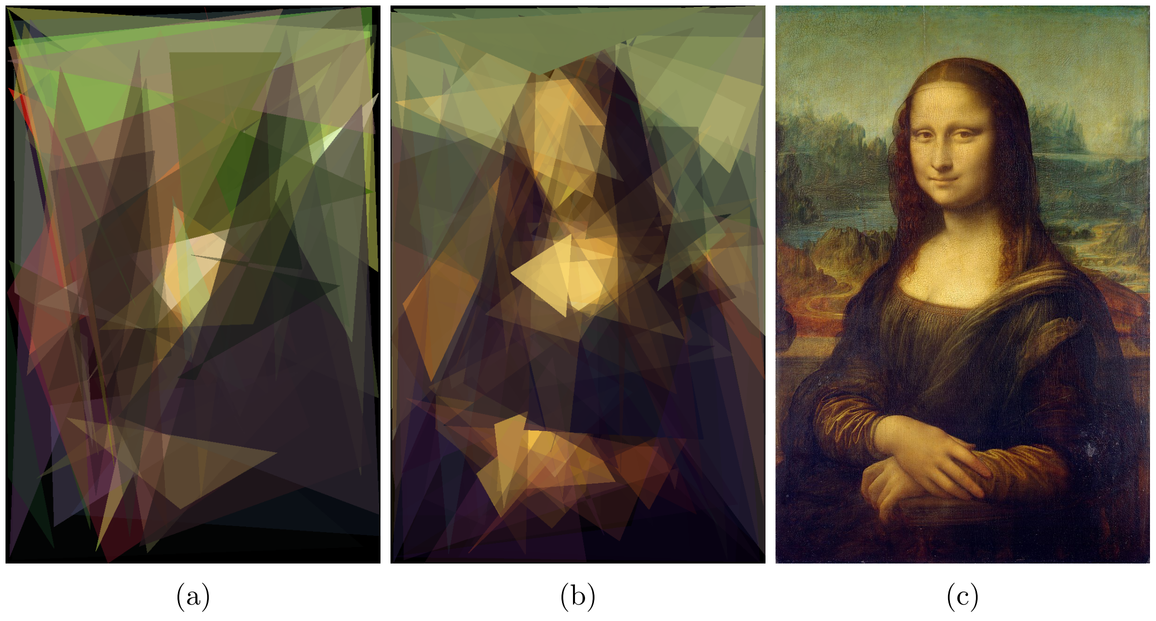

Figure 2: **(a)**: image for 10,000 iterations. Shows 48 polygons. **(b)**: reconstruction after 300,000 iterations shows only 63 polygons. Ran in 2 nodes and 48 processors. 93.83% similarity between central and right images. **(c)**: original image

4 Conclusion

In this project the benefit of parallelising a scientific code such as a genetic algorithm was approached. This particular use of a genetic algorithm constitutes only a toy example of its power, but it was proven that we are now able to parallelise a complex implementation by focusing on its constructive parts separately. This presents a huge advantage for any scientific computing project as the time gains can be gigantic. Figure 2b shows this promising results.

Nowadays, the access to powerful clusters or super computers such as Tegner is less a limitation for the scientific community than ever. This course has shown us that it is not only a matter of hardware, but also the software that has to be adapted to run in a parallel way.

We were able to replicate a genetic algorithm and parallelise the most time consuming parts of it. The theoretical gains were in the order $O(P)$ when M and P are big, but the communication term, which is logarithmic, appears in the denominator and deforms the gain curve when the processors are less. Empirically, we have shown that the gains for a large amount of processors are very substantial given that the algorithm normally needs very big amounts of iterations for convergence. As it is usual in parallel algorithms, in this case more processors allow bigger images and more iterations, which in the end improve the obtained results.