INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

Practical Work 2

Enrico Manfron

Giuliana Martins Silva

Report presented as a partial requirement for obtaining a grade for the Advanced Computing course of the Master's in Informatics at the Polytechnic Institute of Bragança.

Practical work performed under the supervision of

Professor José Carlos Rufino Amaro

Bragança

February 2023

Summary

Introduction

1. Objectives

This practical work aims to apply the knowledge acquired during the classes of the Advanced Computation course by developing a parallel version based on the MPI programming model of a serial implementation of a Mandelbrot Set generator.

Development

1. Profiling the serial version with Valgrind

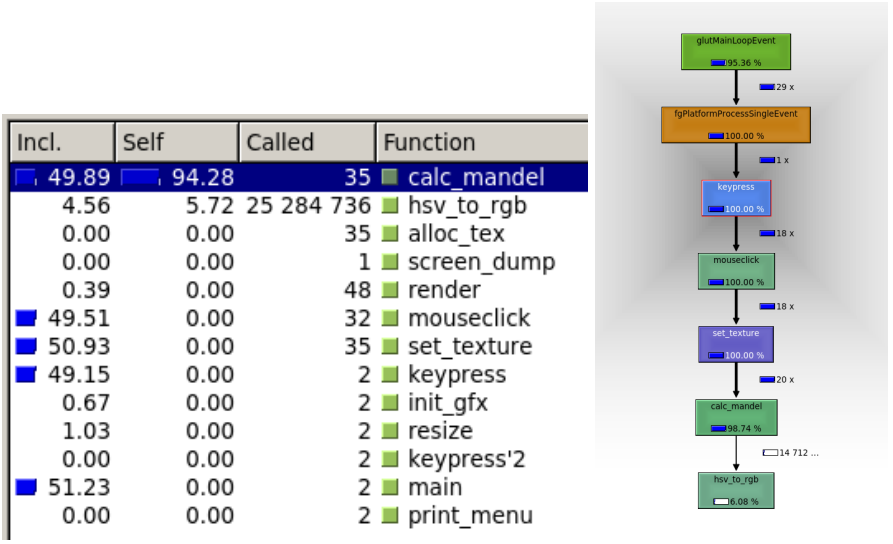
To profile the code with Valgrind, we utilized the commands:

```
gcc -Wall -O0 -g mandelbrot-gui-serial.c -lm -lGL -lGLU -lglut -o mandelbrot-gui-serial.exe

valgrind --tool=callgrind ./mandelbrot-gui-serial.exe

kcachegrind
```

As it is possible to notice in the Image 1, disregarding the main function, the methods that consume the biggest tasking time is calc_mandel that will be our focus in this work.



Incl.	Self	Called	Function
49.89	94.28	35	calc_mandel
4.56	5.72	25 284 736	hsv_to_rgb
0.00	0.00	35	alloc_tex
0.00	0.00	1	screen_dump
0.39	0.00	48	render
49.51	0.00	32	mouseclick
50.93	0.00	35	set_texture
49.15	0.00	2	keypress
0.67	0.00	2	init_gfx
1.03	0.00	2	resize
0.00	0.00	2	keypress'2
51.23	0.00	2	main
0.00	0.00	2	print_menu

Image 1: Valgrind results

2. Applying Amdahl's Law

The calc_mandel method is responsible for 94.28% of the program's tasking time, based on this data, we can calculate the theoretical speed-ups based on Amdahl's law expressed by equation 1, where P corresponds to the number of taskors, F is the fraction of the code that can be parallelizable and S is the theoretical speed up.

$$S(P) = \frac{1}{(1-F)+\frac{F}{P}} \quad (1)$$

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
S_R	1	1,8918	2,6920	3.4141	4.0690	4,6656	5,2114	5,7126

Table 1 - Theoretical Speedups of the Serial Version

Given the calculated speed-ups, we can also obtain the theoretical efficiency of the code given by the equation 2, where P corresponds to the number of taskors, E is the theoretical efficiency and S is the theoretical speed up.

$$E(P) = \frac{S}{P} \quad (1)$$

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
E_R(%)	100	94,59	89,73	85,35	81,38	77,76	74,45	71,41

Table 2 - Theoretical Efficiency of the Serial Version (in %)

3. Run the Serial Version without Profiling

The next step was to modify the code to print the time elapsed from the moment 'Z' is pressed to the moment just before the last frame is saved, change the number of iterations to 4096, and run the serial version using the commands:

```
gcc -Wall -O2 mandelbrot-gui-serial.c -lm -lGL -lGLU -lglut -o mandelbrot-gui-serial.exe

./mandelbrot-gui-serial.exe
```

We ran the code 3 times and got the smaller time that was 15,67 seconds.

4. Parallelization

4.1. Development

To parallelize the serial version, we divide the amount of work between the tasks as equally as possible. The logic we followed was to divide the height of the image generated by the Mandelbrot Set generator by the total number of tasks. In case the division is not exact, the rest is added to the partition of the main task. This way, each task is responsible for calculating the pixel values that make up its part of the image.

The main task was responsible for displaying the image window, printing the menu, handling the behavior when a key was pressed and broadcasting the data, and gathering information of other tasks. The other tasks only had to concern with the pixel values calculation of their partition, which was made in the set_texture() method.

Every time the Mandelbrot Set Image needed to be recalculated, the main task would broadcast to the other tasks the data necessary to perform the calculations correctly. The data contained information such as:

- **color_rotate**, used to change the image's color palette.
- **invert**, used to invert the image colors.
- **height** and **width**, which give the dimensions of the image.
- **max_iterations**, represents the max number of iterations used to calculate the pixel values.
- **saturation**, used to change the image colors' saturation.
- **tex_h**, **tex_w** and **tex_size**, used to allocate the memory in which will be stored the calculation result.
- **refresh**, used to indicate if the memory needs to be reallocated.
- **cx** and **cy**, that indicate the mouse coordinates.
- **scale**, that provides the image scale.

To broadcast all this information, we created a struct type and an MPI datatype corresponding to the struct. When a parameter is changed, the struct data is updated and broadcasted to the other tasks using MPI_Bcast function.

When the tasks receive a broadcast, the memory is reallocated if the refresh parameter is 1, and the pixel values are calculated. To allocate the data for each task,

disregarding the main task, we use the same procedure used for the serial version, but instead of using the original image height, we divide the image height by the total number of tasks and use this value as the height to allocate the amount of memory that corresponds to their partition. The main task will always allocate the memory that will store the whole image.

Similarly to the previous step, the work will be divided according to the height of the image, so each task will calculate the Mandelbrot Set values corresponding to a range of pixels that goes from h_1 to h_2, represented by the equation 1 and 2 respectively:

$$h_1 = \frac{total\ image\ height * current\ task\ rank}{total\ number\ of\ tasks} \quad (1)$$

$$h_2 = h_1 + \frac{total\ image\ height}{total\ number\ of\ tasks} \quad (2)$$

If the division is not exact, the rest will be handled by the main task, then the rest is added in h_2 if the $current\ task\ rank$ is 0, otherwise the rest will be added in h_1 for the other tasks calculate correctly which height it should start. After this, it is necessary to convert these calculated values to RGB. The conversion needs to normalize the values between the minimum and maximum values calculated. So that each task have vary between the same range of pixel values, we use the MPI_Allreduce method in MPI_MIN and MPI_MAX mode to broadcast the min and max values, respectively.

After the conversion, we used the MPI_Gather method to combine all the calculated ranges from each task in the memory previously allocated by the main task. As the data sent by the tasks through the MPI_Gather function needs to have the same size, if the main task has more lines than the other tasks, we just have to make the pointer of the receiving buffer correspond to the number of additional lines that the main task calculated. For example, if the height of the image is 738, and the total number of tasks is 5, we don't have an exact division of work between the tasks because mod(738,5) = 3, so the task main is responsible for calculating 3 additional lines. Once it is finished, when calling the MPI_Gather method it will skip the corresponding 3 lines in the image on the receive buffer, making the size shared between all tasks correctly correspond to the integer division between the image size and the number of tasks.

Using this idea with the MPI_INPLACE option, the main task will receive the other parts correctly even if the other tasks calculated fewer rows than the main task. Finally, the main task will display the image in the window and wait for another key to be pressed.

Image 2, presents a diagram that illustrates the total cycle of the program described in this section.

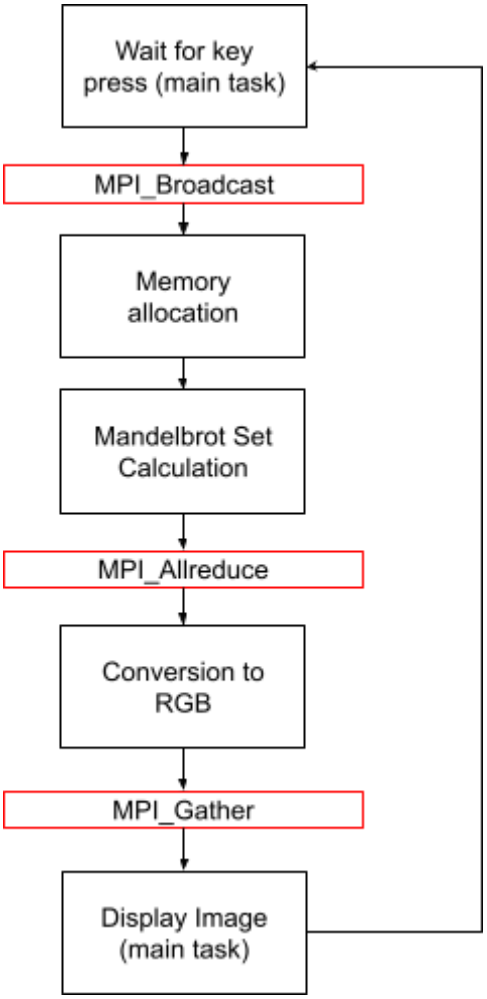


Image 2: Diagram of the total cycle of the parallelized program.

4.2. Benchmarking

The time of the executions of the parallel implementation for N tasks is given in Table 2 and Image 3, the real speedup calculated for each scenario is presented in Table 3 and Images 4 and 5, and the efficiency of the parallel version and the fraction between the real efficiency and the ideal efficiency is given in Table 4 and Image 6.

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$\mathbf{T_{R\,(s)}}$	28.08	23.61	18.96	16.04	13.98	12.04	10.85	9.44

Table 2 - Real Execution Times of the Parallel Version (in seconds)

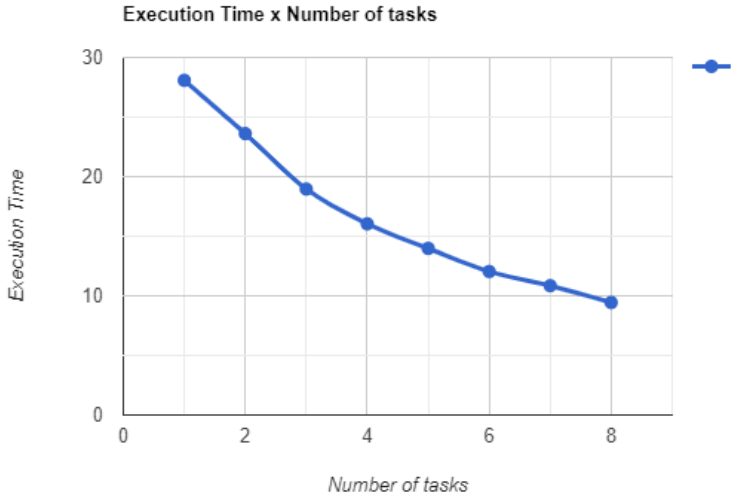


Image 3: Graph representing the execution time given the number of tasks.

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$\mathbf{S_R}$	0.5566	0.6620	0.8244	0.9744	1.1180	1.2982	1.4405	1.6557

Table 3 - Real Speedups of the Parallel Version

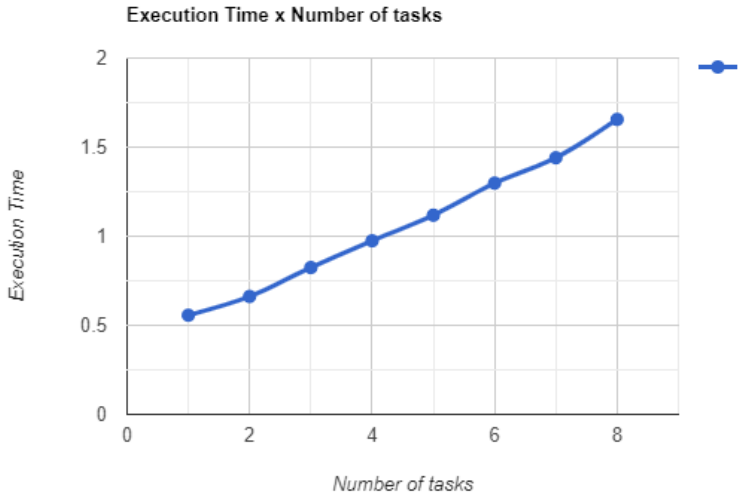


Image 4: Graph representing the real speed given the number of tasks.

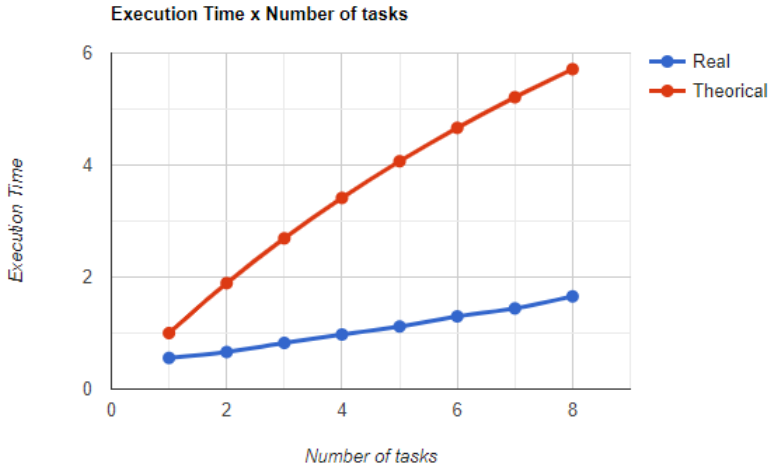


Image 5: Graph representing the real and theoretical speed given the number of tasks.

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$\mathbf{E_{R\,(\%)}}$	55.66	33.10	27.44	24.36	22.36	21.64	20.58	20.69
$\mathbf{E_R/E_T}$	0.5566	0.3499	0.3058	0.2854	0.2748	0.2783	0.2764	0.2897

Table 4 - Efficiency of the Parallel Version (in %) and closeness of the real to ideal efficiency (in %)

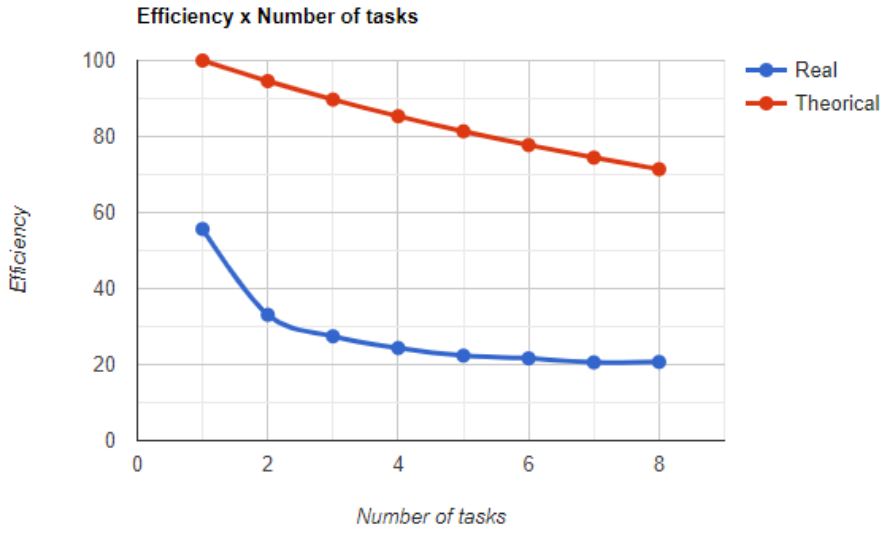


Image 6: Graph representing the real and theoretical efficiency given the number of tasks.

Análise e Discussão de Resultados

In Table 5 and Images 5 and 6, it is easy to notice that the real speed and efficiency are far away from the theoretical values calculated.

The real speed for 1 task is 1.79 times slower than the serial implementation. This phenomenon is better explained by observing Image 2, in which the blocking routines are represented by a red border. Despite the blocking message passing routines that are necessary to achieve the correct result of the program, they force one task to wait for all the others.

In conclusion, our parallelized implementation of the Mandelbrot Set generator just compensates if more than 4 tasks are used which can still be very useful in other cases.

In this practical work, we could gather the information we studied and discussed in the classroom and apply it to a real-world problem, we tried to use as much as we could of different routines presented in the classes and understand the pros and cons of our implementation decisions.