Advanced Computing – 2021/2022

MsC in Informatics – ESTiG/IPB

Practical Work 2:
Mandelbrot Set

Goal: To parallelize the generation of the Mandelbrot Set using the MPI programming model.

Preliminary Note:
This work is accompanied by a ZIP archive with related resources (ac-mandelbrot-resources.zip).

Introduction [1]:
"The Mandelbrot set is the set of complex numbers *c* for which the function $f_c(z)=z^2+c$ does not diverge to infinity when iterated from z=0, i.e., for which the sequence $f_c(0)$, $f_c(f_c(0))$, etc., remains bounded in absolute value. [...] Images of the Mandelbrot set exhibit an elaborate and infinitely complicated boundary that reveals progressively ever-finer recursive detail at increasing magnifications; mathematically, [...] the boundary of the Mandelbrot set is a ***fractal** curve*. Treating the real and imaginary parts of *c* as image coordinates on the complex plane, pixels may [...] be coloured according to how soon the sequence $|f_c(0)|$, $|f_c(f_c(0))|$, … crosses an arbitrarily chosen threshold [...]. The Mandelbrot set has become popular outside mathematics both for its aesthetic appeal and as an example of a complex structure arising from the application of simple rules. It is one of the best-known examples of mathematical visualization, mathematical beauty, and motif."

Details and Methodology:

In this work you are given a serial implementation of a Mandelbrot Set generator and you are required to develop a parallel version based on the MPI programming model, faster than the serial version. The serial version is the PPM Interactive variant provided in [2] with some modifications.

You can use your personal computer if you have Linux installed (either a bare-metal installation or in a virtual machine). Mind you, however, that a small number of CPU-cores will limit the scalability tests. Thus, for the final benchmarking it is recommended to use the classes cluster frontend or at least a computer system that supports >=8 CPU threads (allowing >=8 parallel tasks).

a) Compiling and Running the Serial Code

The companion ZIP archive contains 2 files: a project file (Makefile), the source code of a GUI-based serial version (mandelbrot-gui-serial.c), and the starting point for the source code of the GUI-based parallel version (mandelbrot-gui-mpi.c, a copy of mandelbrot-gui-serial.c). Thus, differently from the previous work, in this you must preserve the graphical interface in the parallel version.

Before trying to compile the serial version, make sure you have the freeglut library [3] installed in the system you chose to develop the work. In a Debian/Ubuntu-based system, you can install it using the command sudo apt install freeglut3-dev. If you chose to develop the work in the frontend of the cluster used in the classes, then the library is already installed there. Extract the ZIP archive file in the target system, edit the Makefile and comment/uncomment the appropriate lines, and execute make to compile. Then, to execute the serial version just type ./mandelbrot-gui-serial .

Note: to run the program in the cluster frontend and see the images generated, you need to use a client that supports a local X server; if you are using Windows, connect to the frontend via mobaxterm; if you using Linux or MacOS, connect through the terminal with the ssh client using the -X option (e.g., ssh -X ac_group01@172.31.254.5); note, however, that the graphical performance may be low, specially if you are connected from home, or from a wireless network.

Take some time to explore the various options offered by the program (these are shown in the terminal where you launch it). Start by using the mouse to "navigate" in the fractal: mouse-click on any point of the current image to dive/zoom in (left click) or to dive/zoom (left click). Verify the effect of changing the number of iterations with options 'i' and 'I'; the 1st will make the fractal generation faster (though less precise), and the 2nd will do quite the opposite; you can also use the '<' and '>' keys to do one step decrements or increments in the number of iterations. Finally, explore the 'z' and the 'Z' options; both allow to dive in the fractal through a predefined path of 18 steps (the corresponding <x,y> coordinates are in the GLOBAL_zoomin vector of the program); the 1st option generate one frame at a time, so you need to press 'z' 18 times to reach the last frame; the 2nd option generates all frames in a row and automatically saves the last frame in a file.

Note: the predefined resolution of the fractal images is 1024x768; this should be kept unchanged.

The goal of this work is to parallelize the generation of the fractal frames, so that the work involved in generating each frame is uniformly distributed by the available MPI tasks (all tasks, including the one with rank 0, should participate in the generation of a frame). Thus, if you mouse-click, or if you use the 'z'/'Z' keys, you should see the next frame to appear faster; however, note that although it may be generated faster, it may not be rendered faster; that will depend on the system where you are running the program: if are connected to the cluster via SSH, the rendering speed may be affected.

b) Profiling with Valgrind

Before developing the parallel version you should inspect the file mandelbrot-gui-serial.c carefully, to apprehend its basic structure. Note that you are not required to understand the mathematical details of the Mandelbrot set nor the details of its implementation; just focus on the general structure of the code. Also, study the code components that deal with the GUI, once you are required to preserve the GUI functionality in the parallel version that you will develop.

Then, modify the Makefile to compile the code properly to run through valgrind [4], and use the callgrind tool to capture the profiling data when running the program with the default global variables values (leave the number of iterations at 256), using the option 'Z'. Then, use the kcachegrind [5] utility to visualize the call graph and identify the most relevant code hot-spots.

In your report: i) provide the command used to run mandelbrot-gui-serial.exe via valgrind; ii) provide screenshots of the kcachegrind windows that show the hot-spots (left panel) and the call graph of `main`; iii) identify the biggest hot-spots and their relative importance (%); iv) identify which of the previous hot-spots are parallelizable (and explain why they seem to be parallelizable).

c) Applying Amdahl's Law

Considering the aggregated weight (%) of the code that you decided to parallelize, apply Amdahl's Law to calculate the theoretical speedup S_T of the parallelization for a number of tasks N=1,...,8. Explain and justify, in your report, the value given to the different factors of the Amdahl's Law formula and provide a table like the following with the missing values (?) of the S_T speedups.

N	1	2	3	4	5	6	7	8
S_T	?	?	?	?	?	?	?	?

Table 1 - Theoretical Speedups

Based on the Table 1 values, provide also a table with the corresponding theoretical efficiency E_T :

N	1	2	3	4	5	6	7	8
$E_T(\%)$	?	?	?	?	?	?	?	?

Table 2 - Theoretical Efficiency (in percentage)

Finally, determine the theoretical speedup limit and also present that calculation in your report.

d) Run the Serial Version without Profiling

Later, to evaluate the correctness and performance of the parallel version, you will need to compare its execution time and its output with that of the serial version, in the most demanding scenario.

Therefore, before developing the parallel version do the following: edit the serial version and instrument the code in order to measure the time elapsed from the moment 'Z' is pressed to the moment just before the last frame is saved; that time should be printed in the screen before that frame is saved; also, change the number of iterations (GLOBAL_max_iter) to 4096.

Then, edit the Makefile and change it to compile the instrumented serial version for benchmarking (not for profiling); remake the executable (make clean; make), run the instrumented serial version at least 3 times and take note of the smallest time spent by the option 'Z'; also, preserve the file of the last frame to be later compared with the output of the parallel version.

e) Parallelization

e.1) Development

Divide the work (the hot-spots to parallelize) by MPI tasks in a way that each task performs the same amount of work (to ensure uniform load distribution you may assume only certain numbers of tasks may be used). Ensure a proper task synchronization. Explain your strategy in the report.

You must ensure that after each new frame is generated, it is shown in the screen by task 0. Also, ensure that, before generating a new frame, any parameters that may affect the behavior of the program, and are changed by the user in the console of task 0, are also passed to all the other tasks.

The code for the MPI version must be put in the file mandelbrot-gui-mpi.c. This is supplied as a copy of mandelbrot-gui-serial.c and you must change it progressively to produce the MPI version.

Hints:
- start with 2 tasks and compare the output of the simulation with the serial version, to ensure the results are the same (or at least very similar - a perfect matching will depend on some choices you will have to make during the implementation); then, increment the number of tasks;
- during development, use the same parameters used for profiling, allowing for a faster execution
- use collective operations when possible, to improve performance and simplify the code
- try to minimize data copies (for instance, check if the MPI_IN_PLACE parameter is applicable)
- verify if derived data types can be used, in order to simplify the code and/or make it more robust

e.2) Benchmarking

When ready for benchmarking, set GLOBAL_max_iter to 4096 and make sure that you are still able to measure the time spent in the 'Z' option, in the same way as it was measured when profiling the serial version. Then, edit the Makefile, change it for benchmarking and generate the final mandelbrot-gui-mpi.exe executable. Finally, run this program 3 times, through the 'Z' option and for each different number of tasks N=1,...,8 take note of the smallest real time T_R for each N. In you report, provide the following table filled with the smallest real times (? in seconds) observed:

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
T_R (s)	?	?	?	?	?	?	?	?

Table 3 - Real Execution Times of the Parallel Version (in seconds)

Now it's time to calculate the real speedups S_R. Provide in your report the following table:

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
S_R	--	?	?	?	?	?	?	?

Table 4 - Real Speedups of the Parallel Version

Provide also in the report a table with the real efficiency E_R achieved by the parallel version, and the ratio E_R/E_T (this ratio tells how close is the parallel implementation to the Amdahl's Law prediction):

N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
E_R (%)	--	?	?	?	?	?	?	?
E_R/E_t (%)	--	?	?	?	?	?	?	?

Table 5 - Efficiency of the Parallel Version (in %) and closeness of the real to ideal efficiency (in %)

For each table, provide also a companion graph, to make the data easier to understand and interpret.

f) Discussion

Discuss the results achieved. Do they correspond to the predictions ? Are they behind expectations (and if so, what could be the motive(s)) ? Are there some more optimizations that can still be done ?

-- / --

Groups: keep the same groups of work 1, unless you were an individual group and found a pair.

Deadline: The report (PDF) and code (C) should be sent to rufino@ipb.pt until February 10th 2023.

Plagiarism:
Plagiarism will not be tolerated and will be firmly handled by the current regulations of ESTiG/IPB.

References:
[1] https://en.wikipedia.org/wiki/Mandelbrot_set
[2] https://rosettacode.org/wiki/Mandelbrot_set#C
[3] https://freeglut.sourceforge.net/
[4] https://valgrind.org/
[5] https://kcachegrind.github.io/