TDDD56 Multicore and GPU computing
Lab 1: Load balancing

August Ernstsson, Nicolas Melot
august.ernstsson@liu.se

November 10, 2020

1 Introduction

This laboratory aims to implement a parallel algorithm to generates a visual representation of the Mandelbrot set similar to Fig. 1. The algorithm we use in this lab work to compute such picture computes each pixel independently. Since this would allow to process all pixels in one parallel step, it is called an *embarassingly parallel* algorithm. However, a processor has typically much less computing units (cores) available than pixels. Thus, the entire picture to be computed must be shared among all cores and each core can sequentially compute its part, while other cores also compute theirs at the same time.

Although the algorithm is embarassingly parallel, one must take great care when distributing the work, as bad work partitioning can produce parts harder to compute than others. If all cores receive one part and one part is significantly longer to compute, then one core will takes more time to run, delaying the completion of the overall algorithm and leaving other unused. In contrast, if all cores work for the same amount of time on their part, then available cores are exploited optimally and the runtime of the algorithm is further reduced.

The work in this lab consists in three parts. First we study the algorithm we use in order to implement a first parallel version with a naive work partition. Second, we measure and analyse the performance of this first implementation, and investigate where and why the computation can be slowed down. The last step consists in programming a smarter partitionning strategy in order to overcome the difficulties identified in the second step, and measure the performance improvements.

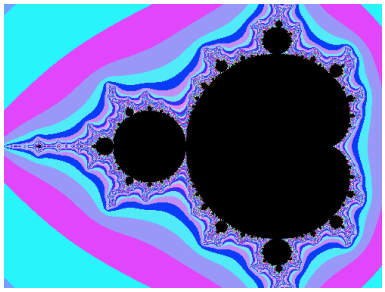

Figure 1: A representation of the Mandelbrot set (black area) in the range $-2 \leq C^{re} \leq 0.6$ and $-1 \leq C^{im} \leq 1$. The pixels in the black area require the full iteration number $MAXITER + 1$.

2 Computing 2D representations of the Mandelbrot set

The Mandelbrot set is defined as the set of complex values c, for which the norm of the Julia sequence $\|a_n\|$ shown in Eq. 1, starting with $z = (0,0)$, is bounded to $b \in \mathbb{R}^{+*}$ with $b > 0$. We can represent the 2D space of a subset of $\mathbb{C}$ with a 2D picture. Let us define the bounds of this subset with C^{im}_{min} and C^{im}_{max} respectively for the minimum and maximum multiples of i and C^{re}_{min} and C^{re}_{max} for the real counterpart. The set we want to represent is then $\mathbb{C} \cap ([C^{im}_{min}, C^{im}_{max}] \times [C^{re}_{min}, C^{re}_{max}])$; we call it P. We can attribute each pixels of a 2 dimensional picture, to exactly one point p of P.

$$\begin{cases} a_0 & = z \\ a_{n+1} & = a_n^2 + c, \forall n \in \{\forall x \in \mathbb{N} : n \neq 0\} \end{cases} \tag{1}$$

In order to give each pixel a color, we need to determine if its corresponding point p is included in the Mandelbrot set. The algorithm depict in Fig. 2 computes the Julia sequence starting at $z = (0,0)$ and returns the number of iterations n it ran before detecting the sequence diverges. If n is below a maximum constant $MAXITER$, then we consider p is not included in the Mandelbrot set and we give it a color depending on p; otherwise, the pixel is colored in black.

Note that this algorithm is an approximation: if the Julia sequence with $c = p$ diverges very slowly, then a constant number of iteration may not be enough to generate a divergence greater than $MAXDIV$. In this case, p is assumed to be part of the Mandelbrot set, whereas it is not. On the other hand, if $MAXITER$ is too high or if $MAXDIV$ is too low, then the algorithm may detect a divergence where the sequence only varies toward a convergence. Then p would not be accounted as part of the Mandelbrot set even if it should be. for a given value of $MAXDIV$, a high value for $MAXITER$ makes the decision more reliable, but it also makes the cores to run more iterations before "giving-up" and consider p as part of the Mandelbrot set.

3 Getting started

3.1 Installation

Fetch the lab 1 skeleton source files from the CPU lab page and extract them to your personal storage space. **We strongly recommend using Git or a similar tool to manage your lab source code.**

3.2 Source file skeleton

This section describes how to use and compile the skeleton source fiels provided and introduce a few key points you need to know.

3.2.1 Compiling and running

The skeleton can be compiled in three different modes you can enable or disable by passing variables when calling make:

Debugging: This mode makes the executable to compute one picture and store it in *mandelbrot.ppm*. You can use this mode and tweak other variables to make sure the algorithm runs, terminates and produces the expected result. Just call *make* without passing any variable, or only variables to tweak the algorithm.

```
int
is_in_Mandelbrot(float Cre, float Cim)
{
  int iter;
  float x = 0.0, y = 0.0, xto2 = 0.0, yto2 = 0.0, dist2;

  for (iter = 0; iter <= MAXITER; iter++)
    {
      y = x * y;
      y = y + y + Cim;
      x = xto2 - yto2 + Cre;
      xto2 = x * x;
      yto2 = y * y;

      dist2 = xto2 + yto2;

      if ((int) dist2 >= MAXDIV)
        {
          break; // converges to infinity
        }
    }
  return iter;
}
```

Figure 2: A C function to decide whether the complex number belongs to the Mandelbrot set, returning the number of iterations necessary to take the decision.

Measuring: This makes the executable to compute one picture and quit without saving it. It also makes the program to check time (in seconds and nanoseconds) before and after computating, and display these values on the terminal after completion and before exiting. This mode is used by batch and plotting scripts to generate many values, analyse and plot them. You can also use these numbers if you want to interpret them yourself. Call make with the variable *MEASURE* set to any value (*make MEASURE=1*).

Make can also take other useful options through other variables passing:

NB_THREADS: Call make with $NB\_THREADS = n$ *make ... NB_THREADS=n* where n is a non-negative integer. It instructs make to compile a multithreaded version running n threads. If $n = 0$ then it compiles a sequential version. If $n = 1$, then it compile a parallel version in which only one thread runs.

LOADBALANCE: Call make with $LOADBALANCE = n$ *make ... LOADBALANCE=n* where $n \in [0,1,2]$. It selects and compile one load-balancing method among no load-balancing (0), your load-balancing method (1) and an optional additional load-balancing method (2).

Browse the file *Makefile* to find out more variables you can use to tweak your algorithm. You can modify the maximum amount of iterations, you can move P in $\mathbb{C}$, you can change the size of the picture to generate and you can provide another color to represent points in the Mandelbrot set (black by default).

3.2.2 Structure

The skeleton provides an sequential implementation of the algorithm described in sec.2. It is divided in the four source files mandelbrot_main.c, mandelbrot.c, ppm.c and gl_mandelbrot.c with their associated header files.

mandelbrot_main.c The program starts here. Depending on the options you passed to make, it computes and maybe stores a picture, or it starts the opengl engine.

mandelbrot.c This is the only file you need to modify. *At initialization time, it spawns by itself all threads you instructed to use at compile time*. Whatever running mode, whatever amount of threads you use, computation is always started by a call to $compute\_mandelbrot(...)$. Depending on the number of threads, this runs directly $sequential\_mandelbrot(...)$ or it releases all threads which, each of them individually, run $parallel\_mandelbrot(...)$. The implementation uses function $is\_in\_mandelbrot(...)$ to check if a complex number is in the mandelbrot set and $compute\_chunk()$ to compute the whole picture or a region of it, depending on the values of parameters $parameters->begin\_h$ and $parameters->end\_h$ for height (respectively *begin* and *end*) and $parameters->begin\_w$ and $parameters->end\_w$.

By default, one call to $parallel\_mandelbrot(...)$ computes nothing. The function admits as parameters *args* and *param*. $args \rightarrow id$ gives the thread id, from 0 to $NB\_THREADS-1$. The parameter *param* points to a structure from which you can get the dimension of the picture to compute (*height*, *width*), the maximum amount of iterations (*maxiter*), the color for the Mandelbrot set (*mandelbrot_color*, black by default), the bounds defining the subset P of $\mathbb{C}$ matching the picture ($lower\_r$, $upper\_r$, $lower\_i$, $upper\_i$, respectivey for C^{re}_{min}, C^{re}_{max}, C^{im}_{min} and C^{im}_{max}) and a pointer to the ppm data structure holding all pixels (see ppm.h). Note that the function $init\_round(...)$ is garanteed to run by each thread and return before any thread begin to run $parallel\_mandelbrot(...)$. It is a preferred place to implement initializations for any shared resource your threads may use.

ppm.c This file holds all functions required to manipulate ppm files. You need to store pixels in the $args \rightarrow picture$, using $ppm\_write(args \rightarrow picture, x, y, color)$ to give a color to pixel of coordinates (x,y) and where color is an instance of the structure *color*, a triplet of three values for red, blue and green intensities (see ppm.h). You can pick a color in the global variable *color* and extract its RGB components using shifts ($<< 0,8 or 16$) and bitwise AND. All this work is already implmented in $compute\_chunk(...)$ of *mandelbrot.c*.

gl_mandelbrot.c This file includes all necessary code to run the OpenGL animation. You don't need to browse this code for the lab work. **Note: OpenGL visualization may not work using a remote shell.** If you want to try it, compile with *make GLUT=1*.

4 Before the lab session

Before coming to the lab, we recommend you do the following preparatory work

- Write a detailed explanation why computation load can be imbalanced and how it affects the global performance.
  Hint: What is necessary to compute a black pixel, as opposed to a colored pixel?
- Describe different load-balancing methods that would help reducing the performance loss due to load-imbalance. You should be able to come up with at least two.
  Hint: Observe that the load-balancing method must be valid for any picture computed, not only the default picture.

- Implement all the algorithms required in sections 5 and 6 ahead of the lab session scheduled, so you can measure their performance during lab sessions. Use the preprocessor symbol LOADBALANCE to take decision to use either no load-balancing or one or several load-balancing methods (the helper scripts assume a value of 0 for no load-balancing and 1 and 2 for two different load-balancing methods). This value is known at compile time and can be handled using preprocessor instructions such as *#if-#then-#else*; see the skeleton for example.
- Consider ways to further optimize your solution which might be outside of the lab scope (e.g., using advanced tricks taught in lectures).
- Think about how well your solutions generalize to other use-cases (task-based parallelization in general) and if not, how they could be adapted for this purpose. (Only a thought-experiment, no implementation.)

5 During the lab session

Take profit of the exclusive access you have to the computer you use in the lab session to perform the following tasks

- Measure the performance of the naive parallel implementation (naive partitioning shown in Fig. 4 and generate a graph showing execution time as a function of number of threads involved.
  Hint: You will observe more easily the load-imbalance effects if you generate only pictures of the Mandelbrot set in the range $C^{re} \in [-2;+0.6]$ *and* $C^{im} \in [-1;+1]$. *We suggest to generate a picture of* 500×375 *pixels, using a maximum of* 256 *iterations. You are encouraged to change these parameters if this helps you to have a better understanding of the problem or to find a solution.*
- Measure the performance of the load-balanced variant and produce a graph featuring both load-balanced and load-imbalanced global execution time curves.

6 Lab demo

Demonstrate to your lab assistant the following elements:

1. Show the performance (execution time) as a function of number of threads measured on the naive parallel algorithm, through a clear diagram.
2. Explain the reason why some threads get more work than others.
3. Explain the load-balancing strategies you implemented and argue why it helps improving performance. You should have two different approaches that both are improvements to the naive algorithm. *Hint: Consider using one static approach and one dynamic approach. What are the fundamental differences, and what approach best fits to describe the naive algorithm?*
4. Show the performance of your load-balanced versions and compare them to the naive parallel algorithm. Explain the reason of performance differences.

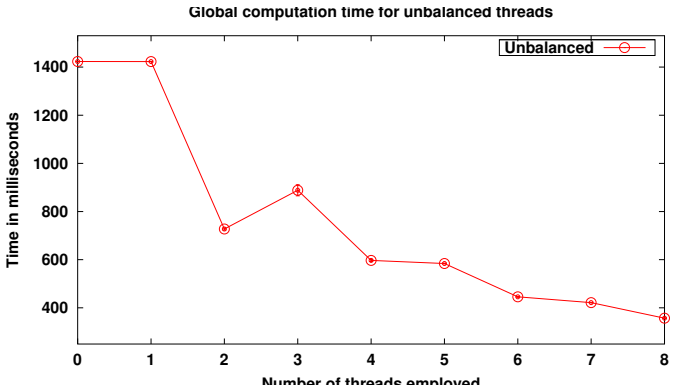

Figure 3: Performance of an unbalanced implementation. The computation time does not lower accordingly with the increasing number of threads.

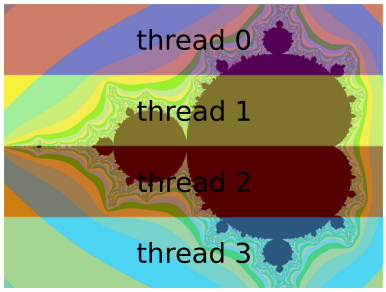

Figure 4: A naive partition for the computation of the representation of a Mandelbrot subset. Every thread receives an equally big subarea of the global picture to compute.