A Mandelbrot set explorer

March 16, 2023

Introduction. The project is personal or by group of two. A short report (one page should be sufficient) with the requested information must be provided. The objective of the project is to get a practical knowledge of parallel programming on a small example.

The goal is to modify a Mandelbrot set explorer program to make use of parallelism, in particular with multicore computers. The program is provided, with the source, on the eCampus space of the course. In principle, you do not have to bother with geometrical/fractal aspects. All language and GUI library aspects (e.g., interacting with the keyboard, displaying images) are already demonstrated in the provided sequential program. According to the pedagogical commitment (*engagement pédagogique*) of the course, you must have access to a computer under Linux with X11. The code is library-independent enough to run on a variety of systems with minimal effort. A Raspberry Pi 3 with a Debian based distribution is sufficient to run and experiment with the software.[1] Virtualization is another possibility, but the performance gain from parallelism might be unexpected and surprising. Alternatively, several ms8xx computers are available in the Montefiore building, as well as remotely using ssh. Again, thanks to the fact that the software only uses the X11 library, remote ssh execution will work with minimal effort.

It is not asked to improve the running times by modifying the Mandelbrot set computing algorithm.[2] The work essentially focuses on parallelism aspects only.

Please do not change the indentation of the existing code, adopt our coding indentation. You might like your indentation better, but we do have many programs to read, and adapting to each of your indentation preferences takes time.

The provided code. The provided code is in C. This code is not perfect. Some imperfections are even intentional to evaluate how you will deal with these problems. To compile the software on Linux, you need `gcc` or `clang`, and `xlib` (e.g. provided by the Debian package libx11-dev); a `Makefile` is provided, so in principle, you just need to type `make` to compile. The program works on the ms8xx network, and it is possible to run remotely, even with the graphical interface (you will have to configure any X11 server on your machine, and use X11 forwarding with ssh). It might be possible to run using Windows Subsystem for Linux, but we do not guarantee support for that. If you are able to compile and run directly on Windows or MacOS, we are grateful if you can provide us with a short tutorial.

The program must be run with three arguments, the width and height of the window, and the number of threads, e.g.

[1] At the current time, it is possible to build a full working RP3-based computer for less than 50 €, to which you would just need to add a keyboard/screen/mouse to have a full working environment.

[2] Although there are numerous opportunities to do so, please do not waste time on improving the computation itself, since we will not evaluate that in any way.

```
./mandelb 800 600 1
```

It draws a picture representing the Mandelbrot set. The current keys are supported by the user interface:

- the arrow keys are used to move horizontally and vertically in the figure;
- the space key is used to zoom in, and minus or equivalently the "b" key is used to zoom out;
- the "r" key toggles rotation;
- the "q" key is used to exit.

Your mission (should you choose to accept it). The sequential program is usable, and interaction is working. However, interaction with the user is clumsy, and no usage is made of the multicore architecture of modern computers, resulting in a low frame count per second. Furthermore, there is no control of the frame rate, the speed of zooming or rotation being a function of the speed of the computer.

The objectives of the project are to learn how to

1. use the available cores on the computer in order to increase the amount of computation per unit of time, i.e., to increase the number of frames per second (fps);
2. organize code to decouple computation and interaction aspects in the program, for a better user experience;
3. understand that the structure of the program has to be conceived with parallelism in mind, e.g., functions should be thread safe;
4. use a cluster of computers, again to increase the number of frames per second.

According to the objectives above, the project has several phases:

1. Use OpenMP to improve the frame count per second. This should be a few lines change in the code, so that the computation of one image is done in parallel. It might take time for you to locate where to operate the change, but if it needs more than a few lines, you are doing it wrong. This simple phase serves as an entry point into the code. The hands-on exercise on OpenMP should have prepared you for this easy task.
2. In the first phase, several threads compute a same image. This allows to make use of parallelism in an easy way, but this has limitations. In particular, since the units of computation are small, it is not extensible for use on a cluster of machines rather than a multicore machine. Now, we will use different threads to compute different images. Use one thread for displaying images, dealing with user interaction, and require images to be computed. Then a certain amount of threads ($n-1$, where n is the number of real cores in your machine) are used for the proper computation of images. Use a thread pool, that is, do not create/destroy a thread each time an image is computed. Get some inspiration of the producer-consumer on a circular list to organize the exchange of information, and control the computation of future frames as well as the frame rate.

Rotation must be of one cycle every 30 seconds. Zooming in (resp. out) must result in a zoom factor of 1.1 (resp. 1/1.1) every quarter of a second.

Busy waiting is allowed, with appropriate sleep instructions to avoid taking too much cpu resource while busy waiting.

Suggestion: use a circular queue to communicate work between the computing threads and the main (GUI) thread. To each task (i.e., image to compute) in the queue, assign a future time and parameters, as well as the status (e.g. to compute/computing/computed/unassigned). Each element in the queue will have its associated image buffer and XImage. An idle thread would look for work to do (to compute), change the status (computing), do the work, change the status again (computed). Appropriate locks will ensure that there is no data race, and at the same time allow parallel computation of images.

The GUI thread would set the next images to compute with a "to compute" status, setting the first images (in number equal to the number of computing threads) at an interval of 1 second with the appropriate parameters (scale, x, y, angle). It will also busy wait for available images (computed status) and each time an image is available, it would request another image (to compute) at a time equal to the current time plus the number of threads divided by the number of frame per seconds (fps variable, initially equal to 1). Each time an image is ready before its time, fps is multiplied by 1.1. Each time an image is ready too late, fps is divided by 1.1. fps is capped to 30.

Again this strategy above is not the best one, but our intention is to develop and evaluate your capabilities to deal with a parallel program, not particularly to make you experts in psychedelic video generation.

3. Remove busy waiting in the version above.

   Suggestion: Use semaphores.

4. One a multicore machine, the picture is reasonably quickly computed if the resolution is very low. With a higher resolution, it might be useful to use a cluster of machines to compute images, because it is becoming advantageous to use the network to transmit images. Use MPI to also use distant computers to help computing frames. If you use MPI together with threads, remember to initialize MPI with the `Init_thread` version of the initialization function.

   Suggestion: Start from the previous version, keep the worker threads, but only make them communicate the tasks to MPI workers and retrieve the images. MPI workers must only wait for work, compute the image, and send back the image. Some mechanism has to be found so that all processes gracefully quit when the user exits the program.

   Ignore the thread argument, use the appropriate MPI commands to know how many computing units are available.

5. Impress us (but remember that only the parallelism aspects are evaluated).

Deliverables. In the short report (roughly one page), explain what you did for each phase (no need to explain phase 1). Report the number of real cores on your experimentation machine, and the number of threads you used for the computation. Provide fps counts for the (given) sequential version, and your version at the several phases. If you are proud of some feature of your program, do not forget to mention it in the report!

A zip file providing a snapshot of your program (with source) **for each of the four phases** has to be uploaded to eCampus.

Organization. The last sessions of the course are dedicated to help you tackle all tasks in this project. Please do not leave all the work for the last days, because it will increase the total amount of time you will have to invest in this project.

Cores, hyperthreading, and all that. There are system dependent ways (i.e., it differs whether you are using Windows, Linux, or another system) to allocate a thread to a particular logical cpu. This means you can ensure that the computation of frames is done on different cores. We do not ask you to use system dependent features, because it will make checking your code difficult. Leave the scheduling of the threads to the kernel, and use a macro to specify the number of cores the program should use.

Plagiarism. Cooperation (even between groups) is allowed, but all cooperation must be clearly referred to in the report, and we will have **no tolerance at all for plagiarism** (writing together, reusing/sharing code or text). Code available on the web that serves as a source of inspiration must also be properly referred in the report. ChatGPT, Copilot, and the like are strictly prohibited.

Participation. Each student has to participate in the project, and do her/his fair share of work. Letting others in the group do all work will be considered as plagiarism.

Assignment. If something is unclear, if you find a mistake, or believe something must be better specified, notify us. This text is not perfect, we will improve it with your comments.

Important final remark. This project is not meant to be a huge amount of work, or to leave you alone to find, e.g., obscure options of MPI. Ask for help, come to the Hands On sessions on Fridays, use the Forum, request a Q/A zoom session, if needed.