TDT4200: Parallel Computing

Parallel Computing - Assignment 3

October 15, 2018

Bart van Blokland
Björn Gottschall
Department of Computer and Information Science
Norwegian University of Science and Technology (NTNU)

- **Delivery deadline: October 19th, 2018 by 23:00.**
- **This assignment counts towards 5% of your final grade.**
- You can work on your own or in groups of two people.
- Deliver your solution on *Blackboard* before the deadline.
- Upload your report as a single PDF file.
- Upload your code and results as a single ZIP file solely containing the source files (src directory). Do not include binaries or additional resources provided with this assignment. Not following this format may result in a score deduction.
- All tasks must be completed using C++.
- Use only functions present up to and including C++11.
- Do not include any additional libraries apart from standard libraries or those provided.
- The delivered code is taken into account with the evaluation. Ensure your code is documented and as readable as possible.

Questions which should be answered in the report have been marked with a **[report]** tag.

Objective: Use the Message Passing Interface (MPI) to distribute work over multiple CPU nodes in different ways to speed up executions.

Parallel Computing - Threads, Mutexes Atomics and OpenMP

Processes executing on a machine are able to create additional execution flows, also known as "threads". This allows a program to execute similar or different parts of its code simultaneously, often significantly speeding up execution time. They are essential for a developer to parallelize algorithms where libraries, such as MPI or OpenMP, cannot be applied or special hardware or infrastructure (GPU, FPGA or clusters) apart from a multi-core processor(s) is missing. Avoiding race conditions and synchronizing threads are the biggest challenges when implementing multi-threaded applications. Constructs such as mutexes and atomics are helpful to approach and solve these challenges.

In this assignment you will implement multi-threaded applications based on different approaches for implementing a Mandelbrot set renderer, which can generate very colourful and diverse pictures of complex numbers such as the one in Figure 1.

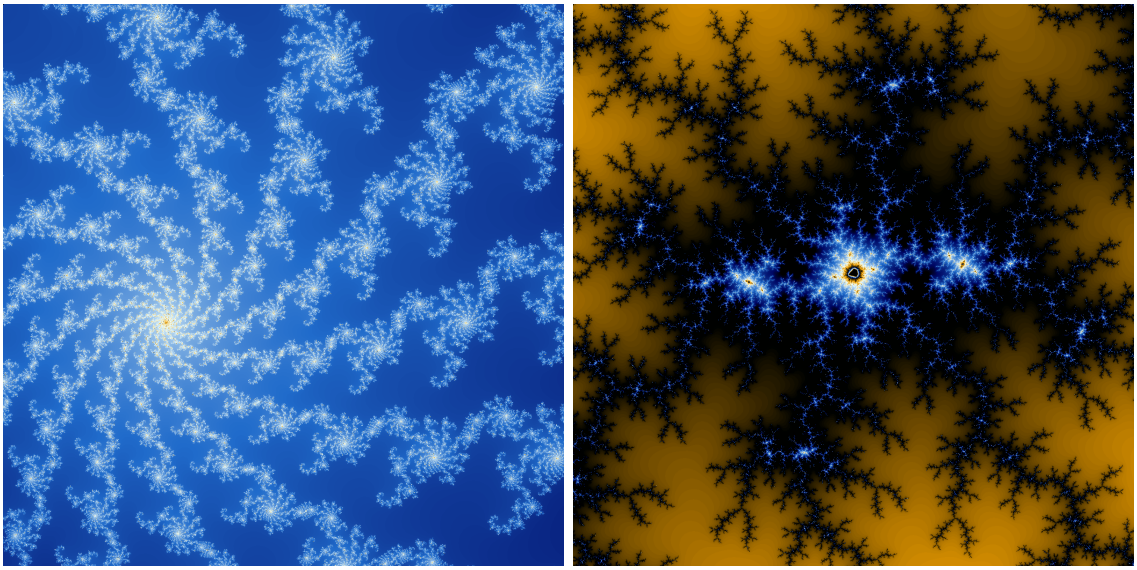

Figure 1: Examples pictures from the Mandelbrot set

The Mandelbrot set is a set of complex numbers for which an iterative function does not diverge. The Mandelbrot set calculation itself is relatively easy as the following definition show.

A complex number c is a member of the Mandelbrot set ($c \in M$) if the following applies:

$$z_0 = 0 \tag{1}$$

$$z_{n+1} = z_n^2 + c \tag{2}$$

$$\left| \limsup_{n \to \infty} z_n \right| \leq 2 \tag{3}$$

If we'd like to draw an image of this set, we can test for each pixel whether it is a member of the set. Unfortunately, doing this accurately requires an infinite number of iterations. As you might imagine, this is not feasible on any hardware. We'll therefore limit n to some value n_{max}.

The needed number of iterations to determine whether a point is a member of the Mandelbrot set is also called the *dwell* value. Consequently, n_{max} is called the maximum dwell value. In our case, we convert a pixel coordinate to a complex number c, and test whether it is a member of the set (where the x and y coordinate become the real and imaginary parts, respectively).

If we were to plot only membership of the Mandelbrot set, the result would be a black and white image. However, we can abuse the dwell value to also assign colours to the complex numbers. The colour mapping of the dwell values is mostly eye candy. Colours are chosen arbitrarily, and can be very different across implementations.

The amazing property of the Mandelbrot set are the rich variety in forms and shapes that can be found by zooming deeper and deeper into the set. The zooming factor is only limited by the hardware and implementation of the renderer, which is in our case the precision of the double precision floating point type. However, the provided renderer allows you to easily reach a zoom factor of 7.5×10^{14} %. Have a look at the provided interactive script *mandelNav.sh*, which gives you a comfortable way to explore the depths of the Mandelbrot set (Task 0e)!

Although all operating systems can be used for solving this assignment, Linux is highly recommended and best supported. Keep in mind that you are not allowed to use any additional libraries apart from the C++ standard library or those provided. You are allowed to create additional source files, but your main focus should be working with the existing ones. Please leave the original command line parameters intact.

This assignment will contribute with 5% to your final grade.

Remember that at the very basic level, we're looking for whether you've understood the concepts and methods this assignment touches upon. Make sure you show this when answering a question.

If you have any further questions please ask them first on the blackboard discussion board, as it is likely others have them too. However, make sure not to post large quantities of code there. You can send them by mail to Björn and/or Bart if necessary.

Task 0: Preparation [0 points]

a) Ensure the following tools are installed:

- CMake or make
- Git
- A C++ compiler, such as G++ or MSVC++.

These have already been installed for you on the lab machines.

b) Clone the assignment repository using the command:

```
git clone https://github.com/bgottschall/TDT4200-Assignment-3.git
```

c) Compile the project. You are provided with three ways for getting started. The included Makefile, which takes care of the compilation, run arguments, CMake for generating automatically a *new* Makefile solely compiling the project or the manual way doing everything on your own. You are free to choose, adapt and extend any of these.

- Make

  ```
  make all
  ```

- CMake

  ```
  cd build
  cmake ..
  make
  ```

- Manually

  ```
  g++ -Wall -Wextra -Wpedantic -std=c++11 -O3 -fopenmp  \
      -c src/utilities/lodepng.cpp -o src/utilities/lodepng.o
  g++ -Wall -Wextra -Wpedantic -std=c++11 -O3 -fopenmp  \
      -c src/main.cpp -o src/main.o
  g++ -Wall -Wextra -Wpedantic -std=c++11 -O3 -fopenmp \
      src/utilities/lodepng.o src/main.o -lpthread -o mandel/mandel
  ```

d) Give it a test run either through the shipped Makefile or manually:

- Make

  ```
  make call
  ```

- Manually

  ```
  mandel/mandel -o output/mandel.png
  ```

e) mandelNav.sh

Dealing with the renderer alone to zoom in and produce nice pictures can be a challenging task. Therefore, you are provided with an interactive shell script which is supposed to do all the difficult parameter calculations for you to zoom and navigate through the Mandelbrot set. It is highly recommended to use this tool to save time, as it can fully interact with the renderer. However it has some requirements to your system. To show your current rendered image you need an image viewer which updates the shown picture as so as it is changed from rendering. Such an image viewer is for example *xviewer*, *pix* or *display*. Other requirements are *tput* and *bc* which are typically preinstalled on a Linux system. You can open mandelNav from the makefile (requires xviewer) or manually from the terminal:

```
#Makefile (requires xviewer)
make mandelnav
#Manual
./mandelNav.sh
```

If you choose the manual way, you have to open now your image viewer for the *output/mandelnav.png* file. Best practise is to put your image viewer and mandelNav side by side like seen in Figure 2 and have the focus on the mandelNav window. As

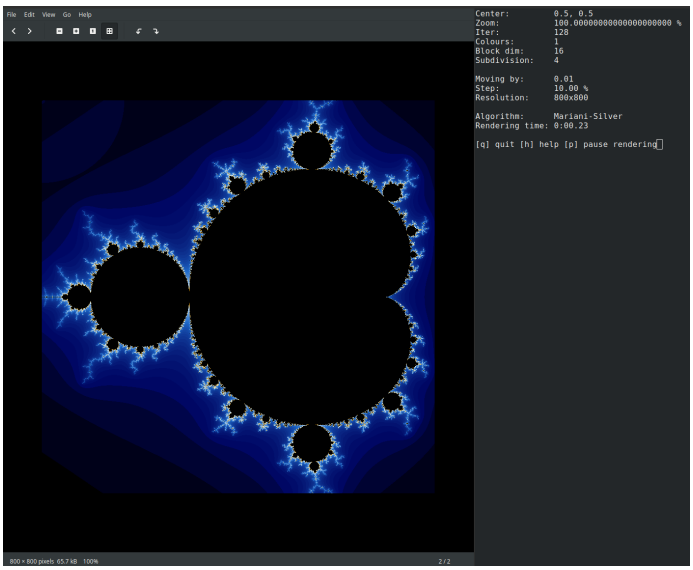

Figure 2: xviewer side by side with mandelNav

soon as you interact with mandelNav a new picture gets rendered and shown in the image viewer. In this way you can comfortably explore the Mandelbrot set and try out different parameters for the following tasks. Press *h* to see all possible interactions with mandelNav. Keep in mind that source code changes require to recompile the project, which can be done in the background without exiting mandelNav.

f) Get a feeling for which parameters impact the rendering time and maybe have already a look at the source code to reason about why. Play around with the subdivision factor, block dimensions, iterations and zooming into different parts of the set.

The renderer supports the parameters *-t* to change rendering algorithms and *-m* to show computational blocks, which are the keys *e* and *t* in mandelNav.

Task 1: Threads [2 points]

In this task we will use *std::thread* to parallelize the Mandelbrot renderer.

a) **[0.7 points][report]** Escape-Time algorithm

The renderer supports two different rendering algorithms, the Mariani-Silver and the Escape-Time algorithm. We'll look at the former later in this assignment. For now, we'll take a look at the traditional (and less complicated) Escape-Time algorithm, which calculates the dwell value for each pixel of the output image. The renderer chooses the traditional algorithm by using the parameter *-t* or by pressing *e* in mandelNav.

As the function `computeBlock` is also used in the Mariani-Silver algorithm, you should create a copy of it e.g. `threadedComputeBlock` not breaking any other functionality of the renderer by putting it in the else branch for the traditional computing.

Use `std::thread` to start multiple threads (minimum 2) for computing the dwellBuffer in parallel for the final image. Split up the work by dividing the image into rectangles (one per thread), and have each thread process one of them. You are not required to use any mutexes or atomics yet. You can ignore the `markBorders` function, which is only for visualization of compute blocks.

Render the Mandelbrot set using the provided single thread version and your implemented multithreaded approach, and measure the speedup. Take care that the rendering time depends on the resolution, position, zoom level and maximum number of iterations. Always find a good position inside the set and keep it to get comparable numbers.

HINT: Consider rounding errors by splitting the work along the resolution, as is should work for arbitrary resolutions.

HINT: `std::thread::hardware_concurrency` is the number of possible concurrent threads. This count is likely the optimal number for your system.

b) **[0.7 points][report]** Mariani-Silver first touch

The Mariani-Silver algorithm itself is a huge optimization in rendering the Mandelbrot set. It only works through the fact that the Mandelbrot set is a connected set, means complex numbers of the set are connected. Derived from this property we can define a rule: Any shape in the rendering area having the same dwell value on all border pixels, can be filled using this dwell value.

The Mariani-Silver algorithm is a typical subdividing algorithm working after the following principle: If the current block has a common border, fill it with the borders dwell value. If the block dimension is below a certain threshold, evaluate the block pixel wise. Otherwise divide the block in equal parts and start over again for each new block. The threshold is the block dimension defined through the parameter *-b* or *u, U*

in mandelNav and the block division factor (subdivision factor) is defined through the parameter *-d* or *j,J* in mandelNav. You can mark the subdivision blocks using the *-m* parameter or *t* in mandelNav. White squares have been evaluated through the common border, red suqares are fully computed.

The function `commonBorder` evaluates all four sides of a block and returns the common dwell value if one is found. Use `std::thread` (maybe `std::atomic` too) to implement a multi threaded common border evaluation function, in which each thread is responsible for one border.

Compare the speedup to the original implementation. Explain why it is faster or slower. How effect the parameter for block dimension, subdivision factor and resolution the rendering time?

HINT: Increase the iterations and resolution to scale up rendering time and get good comparable numbers. Always document which parameters were used or changed.

c) **[0.6 points] [report]** Mariani-Silver

Extend your efforts of parallizing the Mariani-Silver algorithm and start each instance of the `marianiSilver` algorithm in a separate thread. Make sure to wait at the correct positions for thread executions to finish by joining them back to the parent thread.

How many threads are running at most in parallel? Describe briefly how the thread branching ramps up and why this is not an ideal solution. Is it dependent on some application parameters? How does the operating system handle these kinds of cases where too many threads ask for CPU time?

Independently from the added parallelism: How does the block dimension and subdivision factor impacts the rendering time and how is it (or not) connected to the output resolution? Does the current position and zoom level in the Mandelbrot set play a role here? Give an idea whether it would be possible to find optimal parameters (blockDim, subDiv) which fit for all use cases.

Task 2: Producer & Consumer [2.5 point]

A very common approach to implement multi-threaded application is the producer and consumer pattern, in which a queue is filled (producer) with work and one or more workers (threads, consumers) process it. In this task we will fully parallelize the Mariani-Silver algorithm using this pattern.

a) **[0.75 point] [report]** Create a queue

You can find the data structure `job` defined above the marianiSilver function, which holds all information about a block to process, such as required computational parameters (cmin,dc), a reference to the dwellBuffer, position and size of the block.

Find the comment line above the main function and add a `std::deque` as a global variable, using the job data type. Populate the `addWork` function adding a new entry to this deque.

Populate the `worker` function with a while loop, which pops out one job from the deque and feeds it to the `marianiSilver` function. Replace the recursive call inside marianiSilver and the call from the main function with an proper `addWork` call adding the jobs to the deque.

Add a call to the `worker` in the main function after adding the first job and before mapping the frameBuffer. Compile and execute your application.

Briefly describe the changed execution behaviour with the work queue. When is the work created and when is it processed?

b) **[0.75 point] [report]** Condition variable & Mutex

Define a `std::condition_variable` and `std::mutex` as global variables besides the already defined `std::deque` and modify the `addWork` and `worker` function using the solely the mutex to safely insert and take out elements from the deque in a multi-threaded environment.

Why is this step important for parallisation? What are the risks of a race condition and how can inproper usage of mutexes lead to dead locks?

HINT: `std::unique_lock` can help you handling mutexes

c) **[1.0 point] [report]** Worker threads

Add a `std::vector<std::thread>` to the main function and initialize it with `std::thread::hardware_concurrency()` threads executing the `worker` function.

Use the already defined `std::condition_variable` to let the threads waiting in the worker if the deque is empty and notifiy the condition variable in the `addWorker` function after a job was added to the deque. Pay attention that a conditional variable must always be used with a mutex and it must be always held during interaction with

it! You can use the already defined mutex for this purpose. As the threads are now processing the worker function, the call in the main function from the previous sub task can be removed.

Find a proper way using `std::atomic` variables to determine the end of the Mandelbrot set computation done by the threads. Then, let them exit the worker function and join all threads back to the main thread before mapping the frameBuffer.

Document your speedup compared to the provided single thread computation of the Mariani-Silver algorithm. Compare always the same application parameters, like zoom level and current position.

HINT: valgrind provides a tool called helgrind, which can detect inproper usage of mutexes, data races and possible deadlocks

Task 3: OpenMP [0.5 point]

OpenMP is a very handy library for parallel loops and sections of applications. It is very easy to implement and shows strong performance improvements.

Use the original shipped source files for this task! Do not work with any sources from the previous tasks.

a) **[0.5 point]** **[report]** Parallel For Loops

Use the following OpenMP pragma to parallelise **all** for loops in the application, except those responsible for the recursion in function `marianiSilver` (ignore createColourMap, as this function is not time critical):

```
#pragma omp parallel for
```

Explain briefly why adding this pragma to the recursive branching would not help parallelisation (of course, you are free to try it out!). What actions could be taken to successfully parallelise a recursive algorithm with OpenMP?

You should come across one compiler error. Briefly explain why OpenMP cannot be applied to the reported code block. Solve this problem by changing the implementation and removing the incompatibility, without removing the OpenMP pragmas.

After resolving this issue the code compiles and executes without any issue, but the output image is completely messed up. Find the reason for this and explain what happened here. Implement a solution to output the correct image, while keeping the OpenMP pragmas in place.

Document the achieved speed up from the OpenMP implementation and compare it to the original and your parallel implementation from task 2.