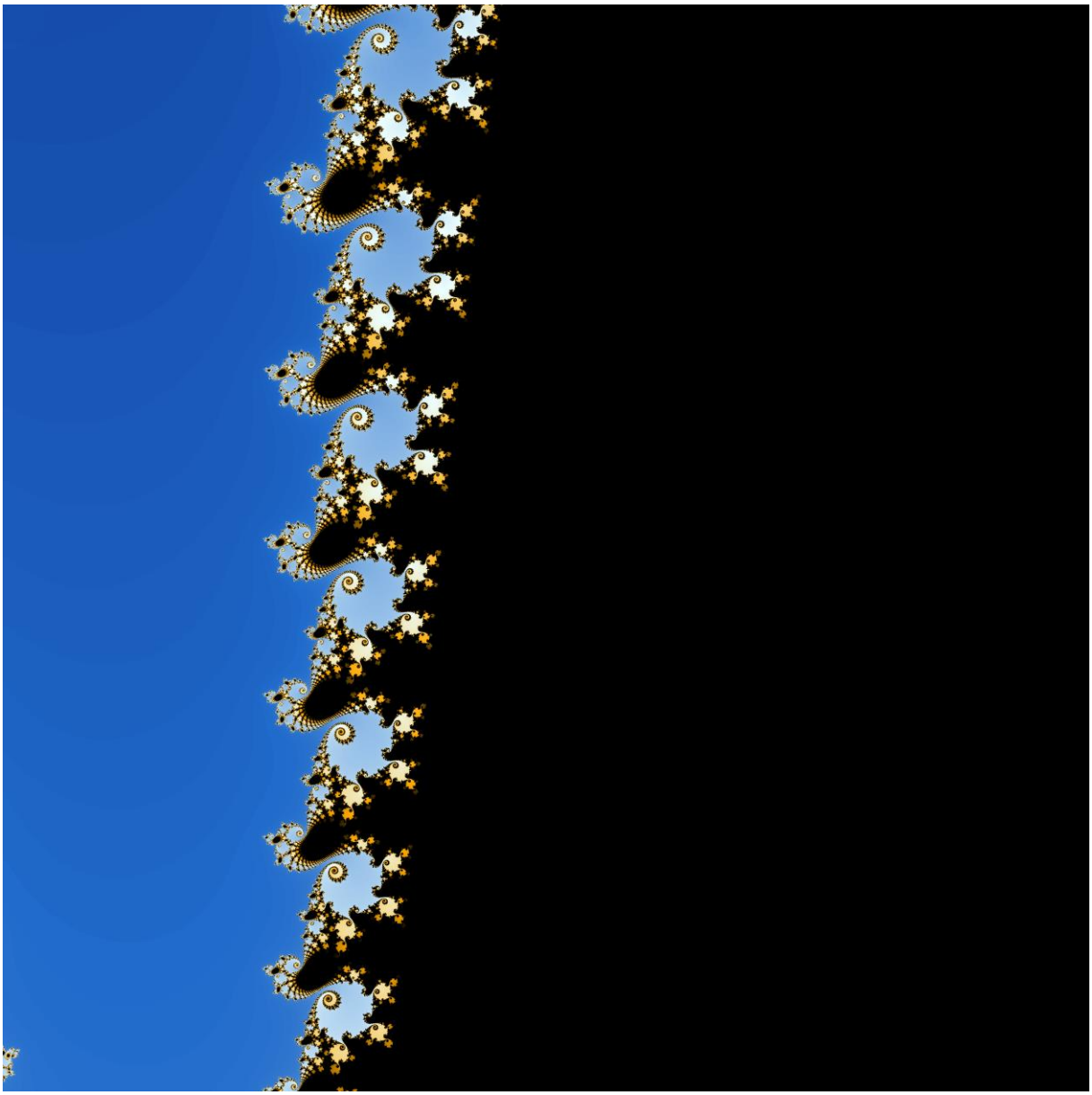
Our mandel segment:



Task 1

Task 1a)

Render the Mandelbrot set using the provided single thread version and your implemented multithreaded approach, and measure the speedup.

command:

mandel/mandel -q -r 800 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 128 -c 1 -b 16 -d 4 -o output/mandelnav.png -t

without threads:

real 0m0.665s
user 0m0.653s
sys 0m0.005s

with threads:

real 0m0.261s
user 0m0.738s
sys 0m0.000s

speedup: 2.54x

Task 1b)

Compare the speedup to the original implementation:

command:

mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 -b 16 -d 4 -o output/mandelnav.png

original:

real 0m1.096s
user 0m1.069s
sys 0m0.008s

With threads:

real 0m6.399s
user 0m6.165s
sys 0m1.349s

speedup: 0.17x → way slower!

Explain why it is faster or slower:

It is slower, because of the recursion calls of the mariani silver method. Every mariani call creates four threads computing one border each. In the recursion case of the mariani function, it calls itself 16 times. Thus, many threads are created and destroyed (thread.join()). This overhead of creating and destroying threads slows down the speed of the algorithm.

How effect the parameter for block dimension, subdivision factor and resolution the rendering time?

block dimension: A bigger block dim results in less border comparisons but pixelDwell has to be executed for more pixels. In our case, the run time increases if the block size becomes smaller, because we have more overhead by creating/destroying threads, because the threadCommonBorder is called more often.

E.g:
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 **-b 64** -d 4 -o output/mandelnav.png
real 0m5.252s
user 0m5.223s
sys 0m0.130s
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 **-b 32** -d 4 -o output/mandelnav.png
real 0m5.398s
user 0m5.355s
sys 0m0.319s
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 **-b 16** -d 4 -o output/mandelnav.png
real 0m6.390s
user 0m6.066s
sys 0m1.418s
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 **-b 8** -d 4 -o output/mandelnav.png
real 0m8.677s
user 0m7.773s
sys 0m3.817s
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 **-b 4** -d 4 -o output/mandelnav.png
real 0m19.826s
user 0m10.825s
sys 0m19.735s
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 **-b 2** -d 4 -o output/mandelnav.png
real 0m18.728s
user 0m10.566s
sys 0m19.684s

subdivision factor: A higher subdivision factor increases the runtime when blockSize and resolution stay constant.
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 -b 16 **-d 2** -o output/mandelnav.png
real 0m6.124s
user 0m6.011s
sys 0m0.959s
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 -b 16 **-d 4** -o output/mandelnav.png
real 0m6.312s
user 0m5.788s
sys 0m1.558s
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 -b 16 **-d 8** -o output/mandelnav.png
real 0m7.184s
user 0m6.096s

sys 0m2.688s
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 -b 16 **-d 16** -o output/mandelnav.png
real 0m6.349s
user 0m5.879s
sys 0m1.520s
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 -b 16 **-d 32** -o output/mandelnav.png
real 0m9.466s
user 0m6.553s
sys 0m6.096s
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 -b 16 **-d 64** -o output/mandelnav.png
real 0m20.739s
user 0m8.933s
sys 0m23.431s

resolution: The resolution is proportional to the run time. E.g. resolution * 2 --> run time * 4
E.g.:
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 -b 16 -d 4 -o output/mandelnav.png
real 0m6.439s
user 0m5.932s
sys 0m1.611s

time mandel/mandel -q -r 3200 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 -b 16 -d 4 -o output/mandelnav.png
real 0m23.685s
user 0m23.678s
sys 0m2.974s

c)
How many threads are running at most in parallel?
unsigned int const numDiv = std::ceil(std::log((double) res/blockDim)/std::log((double) subDiv));
unsigned int const correctedBlockSize = std::pow(subDiv,numDiv) * blockDim;
unsigned int amount_recursions = log(correctedBlocksize) / log(subDiv)
unsigned int amount_threads = (subDiv^2)^amount_recursions
Describe briefly how the thread branching ramps up:
Every mariani call, creates 16 new threads if it runs into the last else block. Thus, the number of threads explodes very fast.
why this is not an ideal solution:
There exist too many threads which all ask for CPU time.
Is it dependent on some application parameters?
Yes: The recursion depth depends on the -d (subDiv) and -b (blockSize) parameters.

How does the operating system handle these kinds of cases where too many threads ask for CPU time?
The OS terminates some threads to decrease the number of threads asking 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?
See task 1b)
Does the current position and zoom level in the Mandelbrot set play a role here?
Yes, because for example at some positions the picture is completely black which means the computation exceeeded the maximum iterations.
Give an idea whether it would be possible to find optimal parameters (blockDim, subDiv) which fit for all use cases
It is not possible because the number of cases are too huge. It depends on which part of the mandelbrot is rendered!

Task 2

a)
Briefly describe the changed execution behaviour with the work queue. When is the work created and when is it processed?
In the main method, the first job instance with initial parameters is created and added to the queue. Thereafter the queue is processed until
no job instances left. When a job is processed within a while loop, the mariani algorithm is called for this job instance, which uses new calculated parameters
to create additional job instances to fill the queue.
b)
Why is this step important for parallisation?
Techniques like mutex coordinate the timing of concurrent threads which other threads are forbidden to access the critical sections, when a thread is already in this section.
What are the risks of a race condition and how can inproper usage of mutexes lead to dead locks?
The risk of a race condition is that multiple threads are trying to access and manipulate a datastructure, which can be lead to inconsistent state of this datastructure. Using mutexes not correctly, it will cause dead locks because threads try to access the critical section. If this section will not be unlocked, then threads will wait as long as this section is unlocked again.
c)
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
time mandel/mandel -q -r 4000 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 -b 16 -d 4 -o output/mandelnav.png
initial:
real 0m9,099s
user 0m9,063s
sys 0m0,036s
optimized:
real 0m6,175s
user 0m17,415s
sys 0m0,065s

speedup: 1.47x

Task 3)

a) Parallel for loops

Explain briefly why adding this pragma to the recursive branching would not help parallelisation (of course, you are free to try it out!).

It does not effect the runtime, because the functions the marianiSilver function calls, are already running in parallel. Running the code of the marianiSilver function (without the functions which are called) does not take a long time and can be done in a single thread. In addition, openMP would spawn way too many threads, compared to the available CPU cores.

What actions could be taken to successfully parallelise a recursive algorithm with OpenMP?

Transfer the recursive algorithm into an iterative one. Afterwards, apply loop pragmas to the created loops in the iterative algorithm.

You should come across one compiler error. Briefly explain why OpenMP cannot be applied to the reported code block.

compile error:

src/main.cpp: In function ‘int commonBorder(std::vector<std::vector<int> >&, const std::complex<double>&, const std::complex<double>&, unsigned int, unsigned int, unsigned int)’:

src/main.cpp:126:14: error: invalid exit from OpenMP structured block

    return -1;

explanation: the return would end the function commonBorder in a single thread. But this is not possible when parts of the function are ran in parallel as one thread can not end all others!

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.

This happens because of the two nested for loops at the end of the main function.

pixel = colour.putFramebuffer(pixel); increments the address by 4 * sizeof(char).

When run in parallel, multiple threads write to the same locations in the frame buffer.

with

colour.putFramebuffer(&(frameBuffer.at((res*y + x) * 4)));

it is made sure that each thread fills the correct pixel of the frame buffer.

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

ran on i7 4500U

command:

time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 -b 16 -d 4 -o output/mandelnav.png

initial:

real 0m1,757s

user 0m1,744s

sys 0m0,012s

task1:

real 0m1,918s

user 0m3,511s

sys 0m1,710s

task2:

real 0m1,220s

user 0m3,553s

sys 0m0,028s

task3: openMP implementation

real 0m1,439s
user 0m4,813s
sys 0m0,336s

command with other parameters:
time mandel/mandel -q -r 1600 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 -b 128 -d 2 -o output/mandelnav.png

initial:
real 0m2,477s
user 0m2,461s
sys 0m0,016s
task1:
real 0m1,620s
user 0m5,130s
sys 0m0,028s
task2:
real 0m1,628s
user 0m5,116s
sys 0m0,016s
task3: openMP implementation
real 0m1,562s
user 0m5,318s
sys 0m0,028s

command with other parameters:
time mandel/mandel -q -r 4000 -x .43886020487019384908 -y .48270435953916761800 -s .01030744704819577509 -i 300 -c 1 -b 16 -d 4 -o output/mandelnav.png
initial:
real 0m9,099s
user 0m9,063s
sys 0m0,036s
task1:
real 0m6,733s
user 0m16,970s
sys 0m1,051s
task2:
real 0m6,175s
user 0m17,415s
sys 0m0,065s
task3: openMP implementation
real 0m6,127s
user 0m21,046s
sys 0m0,344s

Depending on the used parameters, there is a speedup up to 1.48x compared to the initial code. Compared to the implementation in task2, there is no speedup.