PAP Assignment 2

Christian Lascsak – 01363742

December, 2021

1 Running the program

Build it via ./build.sh . The output directory of the build is /out. To run the program run out/openmp. Alternatively on vscode, you can build the program by clicking ctrl+shift+b and run it by pressing F5. Check out the README.md file for further information.

1.1 Testing

In the root directory is a file called assert.sh. After running the openmp binary, this shell can check if the resulting ppm images are the same. It compares the output with the original ppm image via the diff command and checks if the diff is empty. If this is the case, then the algorithm runs correctly. This test can be executed in vscode by running the test task.

2 Methodology

First, I copied the main function from a2-sequential.cpp and refactored it. With this I learned more about the code. Then I moved all the functionality to a generate function, that is called by passing an image from main. This function will then generate the mandelbrot set and applic the convolution filter, while also measuring the time.

3 Code structure

Super class ParallelGenerator that executes the mandelbrot and 2d convolution in a generate function. When we need to create a new implemenatation of a mandelbrot generator, we can simply create a new class that inherits from ParallelGenerator. We also give the constructor the number of max_tasks. This makes it easier for us to pass cli arguments to the task generation (see later).

4 Parallelization Strategies

4.1 Mandelbrot

I started with the mandelbrot generator. To get warm with the parallelization method, I first implemented the parallel for loop. I used this construct then, to refactor it to the omp tasks. Then I started to create a new Generator using Taskloops.

4.2 Tasks

My first approach was to create tasks at every loop iterating over the height of the image. The advantage of this approach is, that we can easily parallelize this code. However the disadvantage is, that we do not have a very fine control over the tasks. One task still iterates over the whole width of the image. So with this solution we basically create one task per row and put them to the queue. We also need to take care of the, access of the pixels_inside variable. For that two things were necessary:

- It needs to be shared.
- we need to be aware of race conditions.

To take care of that, I tried to different methods: declaring the code section as omp critical and omp atomic. It had no significant difference, so I decided to leave it at atomic (since incrementing is an atomic operation in the end). With this implementation, we get the following results: Speedups: around 8 - 9 for 16 threads.

4.2.1 Optimizations and Experiments

: To improve these speedups, I experimented with a different approach. With this, we would create tasks for every iteration. But the overhead was too big (one thread making x * y iterations and then x*y tasks had to be done. This was too much overhead and took very long). So this task creation clearly was not feasible. My third approach was to create 16*2 tasks before running the for loop and then splitting the image in 32 similar sized chunks. However, creating these tasks had an overhead and also did not help at improving performance.

So in the end, I decided to stay with the first approach and increase its performance by doing some tweaks. First, I added the nowait directive at the omp single pragma statement. This improves performance a little, because the created tasks can run before the single execution is done. Then, I also added the "untied" directive, because it is not importand for the process which task is calculated by which thread. Then as a last improvement, I added a final directive. The final directive lets us control when the task creation is stopped. So this is basically the omp-conform version of my third approach. With this, I got a really good speedup! Now my speedups are around 9-10 for the Mandelbrot generation. For the 2d convolution, I basically added the same directives after experimenting with the task generation. And it also had a great impact here! Although another problem is, that I am unable to set different max_tasks for 2d convolution and mandelbrot generation in my current architecture. I think I could make better improvements if I had finer controls over it.

4.3 Taskloops

4.3.1 Methodology

My first approach was to create Taskloops for every for loop in the 2d convolution and on the mandelbrot generator. We also have to consider the pixels_inside variable but in this case, I decided to use the "reduction" directive from omp, that sums up all the pixels_inside variables of every task, when they are done executing. With this, performance was not that good, I had a speedup of Around 8-9 compared to the sequential version.

4.4 Optimization

I could significantly improve the performance by simply setting the num_tasks directive. This directive caps the amount of tasks that can be created for a loop. I also set it to threads x 4, which is 16 x 4 tasks, just as in the generator with noraml omp tasks. This had such a great impact on performance, that I reached a speedup of 13 on Average for the mandelbrot generation and a speedup of 9.89 for the 2d convolution. I was very happy with that improvements, so I also decided to stick with them.

4.5 Speedups visualized

I ran my Taskloop generators for 30 times in a loop. With this I got the average results of figure 1. I captured the logs by appending them to a file and then I converted the output to a csv format. I imported this into excel and calculated the averages. This excel sheet also is included in the git repository, as well as the csv with the raw data.

5 Discussion and conclusion

In my case, the version that only uses tasks still needs some optimization. Though I could not find another point where I could increase performance unfortunately. What I could observe however, was that generally

Averages	Tasks	Taskloops
Mandelbrot	9.391369148	13.77505511
2d conv	7.975136171	9.884676236

Figure 1: Averages calculated for tasks approach and taskloop approach

the speedups of my tasks for the 2d convolution were at some point faster than on the taskloops version. Through further optimization on the taskloops, this advantage was completely eliminated. In my opinion it is clear to see, that although tasks gives one much more control, taskloops are far easier to work with and to optimize. Also, I think that the documentation of the normal omp tasks is very scarce. It was hard to find a reliable source that explains task creation in loops.

5.1 Further improvement possibilities

Another improvement, that I would make would be to create the tasks recursive. I think creating them recursive would create new possibilities for reaching a better speedup. Because the biggest disadvantage of the current solution is right now, that I only have a very low control of tasks and their size when only having two loops.