

Parallel Architectures and Programming Models.

Assignment 2.

Code structure

The project contains the following major files:

- **a2-helpers.hpp** – helper functions
- **a2-sequential.cpp** – the original sequential version of the algorithm
- **task.cpp** – parallel version of the algorithm using OpenMP tasks
- **taskloop.cpp** – parallel version of the algorithm using OpenMP task loops
- **Makefile** – base commands needed to run and test the program
- **mandelbrot.ppm** – the PPM file generated by the sequential version of the algorithm
- **mandelbrot-taskloop.ppm** – the PPM file generated by the parallel using task loops version of the algorithm
- **mandelbrot-task.ppm** – the PPM file generated by the parallel using tasks version of the algorithm
- **out_m.txt** – temporary file which is used to store the execution time of Mandelbrot part
- **out_c.txt** – temporary file which is used to store the execution time of Convolution part

How to run the program?

To run the program and test performance of the algorithms use commands from the Makefile listed in the Table 1.

Command	Description
make compile_seq	compiles the sequential version of the algorithm (<i>a2-sequential.cpp</i>). The command generates the output file <i>a.out</i> .
make compile_par_taskloop	compiles the parallel using task loops version of the algorithm (<i>taskloop.cpp</i>). The command generates the output file <i>a.out</i> .
make compile_par_task	compiles the parallel using tasks version of the algorithm (<i>taskloop.cpp</i>). The command generates the output file <i>a.out</i> .
make run	Runs the currently generated output file <i>a.out</i> .
make test_run	Runs the currently generated output file <i>a.out</i> 10 times, after that reads Mandelbrot and Convolutional parts execution times, computes averages and prints it to the console.
make cmp_taskloop	Compares PPM file generated by sequential algorithm (<i>mandelbrot.ppm</i>) to a PPM file generated by the parallel using task loops algorithm (<i>mandelbrot-taskloop.ppm</i>) using cmp bash command. If the output is empty – files are equal.
make cmp_task	Compares PPM file generated by sequential algorithm (<i>mandelbrot.ppm</i>) to a PPM file generated by the parallel using tasks algorithm (<i>mandelbrot-task.ppm</i>) using cmp bash command. If the output is empty – files are equal.

Table 1 – Makefile commands

To run the program from scratch, first you need to compile the necessary version of the algorithm using one of the following commands:

- **make compile_seq**
- **make compile_par_taskloop**
- **make compile_par_task**

After the program is compiled the output file *a.out* should appear in the root folder. Then you can run the program using the commands:

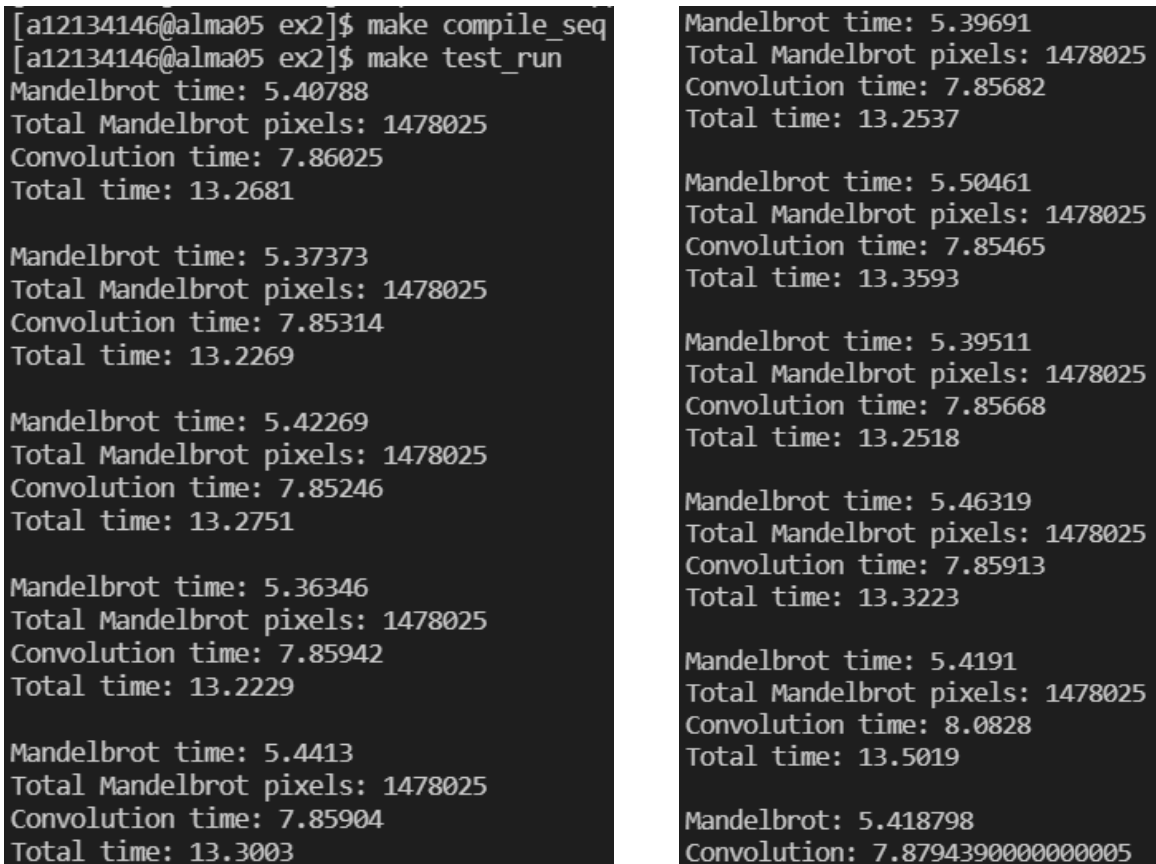
- **make run** – for running the program one time
- **make test_run** – for running the program 10 times and getting the average in the end

The necessary PPM file will be generated or rewritten after running the program. You can compare generated file to the original by running:

- **make cmp_taskloop** – to compare the file generated by algorithm using task loops (*taskloop.cpp*) to the original
- **make cmp_task** – to compare the file generated by algorithm using tasks (*task.cpp*) to the original

Testing the sequential version of the algorithm

First, to get initial values of the execution times of two parts of the algorithm, we do a test run on the original algorithm. All the commands used to perform this, and their output is shown on the Figure 1.



```
[a12134146@alma05 ex2]$ make compile_seq
[a12134146@alma05 ex2]$ make test_run
Mandelbrot time: 5.40788
Total Mandelbrot pixels: 1478025
Convolution time: 7.86025
Total time: 13.2681

Mandelbrot time: 5.37373
Total Mandelbrot pixels: 1478025
Convolution time: 7.85314
Total time: 13.2269

Mandelbrot time: 5.42269
Total Mandelbrot pixels: 1478025
Convolution time: 7.85246
Total time: 13.2751

Mandelbrot time: 5.36346
Total Mandelbrot pixels: 1478025
Convolution time: 7.85942
Total time: 13.2229

Mandelbrot time: 5.4413
Total Mandelbrot pixels: 1478025
Convolution time: 7.85904
Total time: 13.3003

Mandelbrot time: 5.39691
Total Mandelbrot pixels: 1478025
Convolution time: 7.85682
Total time: 13.2537

Mandelbrot time: 5.50461
Total Mandelbrot pixels: 1478025
Convolution time: 7.85465
Total time: 13.3593

Mandelbrot time: 5.39511
Total Mandelbrot pixels: 1478025
Convolution time: 7.85668
Total time: 13.2518

Mandelbrot time: 5.46319
Total Mandelbrot pixels: 1478025
Convolution time: 7.85913
Total time: 13.3223

Mandelbrot time: 5.4191
Total Mandelbrot pixels: 1478025
Convolution time: 8.0828
Total time: 13.5019

Mandelbrot: 5.418798
Convolution: 7.8794390000000005
```

Figure 1 – Test run of the sequential algorithm

The execution time of the Mandelbrot part varies from 5.36 to 5.51 seconds. The average value is 5.42 seconds.

The execution time of the Convolutional part varies from 7.85 to 8.08 seconds. The average value is 7.88 seconds.

General optimizations

To prevent cache misses and false sharing, to speed up the algorithm and decrease the load some general changes were applied to the original version of the algorithm.

Increment

The post-increment is replaced with pre-increment in all the parts because it is more efficient. It increments the value in one operation, while post-increment first moves the value to the register.

Constants

All the variables that do not change their value were replaced with constants because they don't need any additional synchronization and are shared by default. Also, there are a lot of compiler optimizers for constant values which help sometimes to get better performance.

Complex numbers

The call to the **abs()** function in the **while** loop was replaced by taking the square of a complex number, because the **sqrt** operation involved in **abs()** is costly. The loop is presented in the code fragment on Figure 2.

```
// Replace abs(z) which is calculated like
// sqrt(z.real()*z.real() + z.imag()*z.imag())
// with just square of complex number because sqrt operation is costly
while (z.real()*z.real() + z.imag()*z.imag() <= 16 && (iteration < max_iterations))
{
    z = z * z + c;
    ++iteration;
}
```

Figure 2 – Mandelbrot part code fragment

Convolution part

The convolution part applies a filter to an image by iterating over image pixels and applying a kernel transformation to each pixel. The iteration process may be optimized by reducing the number of iterations while processing the pixel. We can skip the check for the cases when current position goes outside the image bounds by determining the range of **displ** for **k** and **l** once before the iteration starts. The illustration of **k** and **l** bounds depending on the position of the pixel is illustrated on the Figure 3. For this example, the image size is 10x10 and **displ** is equal to 3.

	0	1	2	3	4	5	6	7	8	9	j
0	i < displ && j < displ			i < displ			i < displ && j >= w - displ				
1	k = -i ... displ			k = -i ... displ			k = -i ... displ				
2	l = -j ... displ			l = -displ ... displ			l = -displ ... w - j - 1				
3	j < displ			k = -displ ... displ			j >= w - displ				
4	k = -displ ... displ			l = -displ .. displ			k = -displ ... displ				
5	l = -j ... displ						l = -displ ... w - j - 1				
6											
7	i >= h - displ && j < displ			i >= h - displ			i >= h - displ && j >= w - displ				
8	k = -displ ... h - i - 1			k = -displ ... h - i - 1			k = -displ ... h - i - 1				
9	l = -j ... displ			l = -displ ... displ			l = -displ ... w - j - 1				
i											

Figure 3 – The illustration of the Convolution algorithm

The implementation of the described way of iterating is shown in the code fragment on Figure 4. The bounds (**k_start**, **k_end**, **l_start**, **l_end**) are calculated before going into the loops. This way of iterating through the surrounding pixels helped very much in Convolution part optimization.

```

for (int i = 0; i < h; ++i)
{
    for (int j = 0; j < w; ++j)
    {
        double val = 0.0;
        // Reduce the range from -displ to +displ to remove unnecessary if clauses
        int k_start = i < displ ? -i : -displ;
        int k_end = i >= h - displ ? h - i - 1 : displ;
        for (int k = k_start; k <= k_end; ++k)
        {
            int l_start = j < displ ? -j : -displ;
            int l_end = j >= w - displ ? w - j - 1 : displ;
            for (int l = l_start; l <= l_end; ++l)
            {
                val += kernel[k + displ][l + displ] * src(ch, i + k, j + l);
            }
            dst(ch, i, j) = (int)(val > 255 ? 255 : (val < 0 ? 0 : val));
        }
    }
}

```

Figure 4 – Convolution part code fragment

Parallel algorithm using task loops

Mandelbrot part

```

// array to use for marking pexels_inside to prevent race condition
int pixels_inside_arr[w][h] = {0};

#pragma omp parallel num_threads(16) shared(ratio, image, pixels_inside_arr)
#pragma omp single nowait
#pragma omp taskloop num_tasks(256) collapse(2) shared(ratio, image, pixels_inside_arr) firstprivate(pixel)
for (int j = 0; j < h; ++j)
{
    for (int i = 0; i < w; ++i)
    {
        double dx = (double) i / w * ratio - 1.10;
        double dy = (double) j / h * 0.1 - 0.35;

        complex<double> c = complex<double>(dx, dy);

        if (mandelbrot_kernel(c, pixel))
            // Mark the pixel in the array
            ++pixels_inside_arr[i][j];

        for (int ch = 0; ch < channels; ++ch)
            image(ch, j, i) = pixel[ch];
    }
}

// Compute pixels_inside from array of flags pixels_inside_arr
int pixels_inside = 0;
for (int i = 0; i < w; ++i)
    for (int j = 0; j < h; ++j)
        pixels_inside += pixels_inside_arr[i][j];

```

Figure 5 – Parallel using task loop Mandelbrot part

First, we start the parallel section and create threads using **omp parallel**, where the number of threads and the scope of the variables is defined. We can skip defining constants because they are shared by default.

The **omp taskloop** distributes the iterations of the loops across tasks.

The collapse is set to two so that both loops are associated with the **omp taskloop** construct. The number of tasks is set to 256. This number was chosen after trying different values and determining the one that optimizes algorithm the most. Tasks number less than 128 or more than 512 caused decrease in performance.

To prevent the race condition while accessing the **pixels_inside** variable the array **pixels_inside_arr** was created to store the flags (one or zero) in it for each pixel. The final number of pixels is computed after the parallel region is over by summing up the values inside the array. The parallel version of the algorithm is presented above on Figure 5.

Convolutional Part

The parallel section is started using **omp parallel** construct the same way as described in the previous section.

The **omp single** is used to make the part before **omp taskloop** sequential (executed by only one thread).

The task loop is applied on the loop by **i**. The number and size of tasks turned out to be optimal when using the parallelism on this loop. Moreover it is the only one which works correct with the least changes.

The collapse value is set to two to optimally split the work among the tasks considering both loops. The number of tasks was determined by testing different values. Taking less than 128 or more than 450 tasks lead to a performance slow down.

The **omp taskwait** is used to wait for all tasks to be completed before entering this region. This way access to the **src** and **dst** is synchronized and the correct work of the algorithm is guaranteed.

The part of implementation of the Convolution algorithm using task loop is presented on Figure 6.

```
#pragma omp parallel num_threads(16) shared(nsteps, src, dst)
#pragma omp single nowait
for (int step = 0; step < nsteps; ++step)
{
    for (int ch = 0; ch < channels; ++ch)
    {
        #pragma omp taskloop num_tasks(384) collapse(2) shared(nsteps, src, dst) firstprivate(step, ch)
        for (int i = 0; i < h; ++i)
        {
            for (int j = 0; j < w; ++j)
            {
                double val = 0.0;
                ...
                dst(ch, i, j) = (int)(val > 255 ? 255 : (val < 0 ? 0 : val));
            }
        }
    }

    if ( step < nsteps - 1 ) {
        // Wait for all the tasks to finish the step
        #pragma omp taskwait
        Image tmp = src; src = dst; dst = tmp;
    }
}
```

Figure 6 – Parallel using task loop Convolutional part

Results

The test runs of the parallel using task loops algorithm are shown on Figure 7.

The execution time of Mandelbrot part is varying from 0.23 to 0.35 seconds. The average time is 0.31 seconds. The algorithm is 17.48 times faster than its sequential version.

The execution time of Convolution part is varying from 0.65 to 0.87 seconds. The average time is 0.76 seconds. The algorithm is 10.37 times faster than its sequential version.

```
[a12134146@alma05 ex2]$ make compile_par_taskloop
[a12134146@alma05 ex2]$ make test_run
Mandelbrot time: 0.314558
Total Mandelbrot pixels: 1478025
Convolution time: 0.863699
Total time: 1.17826

Mandelbrot time: 0.341337
Total Mandelbrot pixels: 1478025
Convolution time: 0.768753
Total time: 1.11009

Mandelbrot time: 0.338815
Total Mandelbrot pixels: 1478025
Convolution time: 0.751121
Total time: 1.08994

Mandelbrot time: 0.319889
Total Mandelbrot pixels: 1478025
Convolution time: 0.726511
Total time: 1.0464

Mandelbrot time: 0.320253
Total Mandelbrot pixels: 1478025
Convolution time: 0.752042
Total time: 1.07229

Mandelbrot time: 0.346992
Total Mandelbrot pixels: 1478025
Convolution time: 0.849857
Total time: 1.19685

Mandelbrot time: 0.22842
Total Mandelbrot pixels: 1478025
Convolution time: 0.661916
Total time: 0.890336

Mandelbrot time: 0.338348
Total Mandelbrot pixels: 1478025
Convolution time: 0.731396
Total time: 1.06974

Mandelbrot time: 0.329291
Total Mandelbrot pixels: 1478025
Convolution time: 0.872132
Total time: 1.20142

Mandelbrot time: 0.226449
Total Mandelbrot pixels: 1478025
Convolution time: 0.646447
Total time: 0.872896

Mandelbrot: 0.3104352
Convolution: 0.76238739999999999
[a12134146@alma05 ex2]$ make cmp_taskloop
[a12134146@alma05 ex2]$
```

Figure 7 – Test run of the task loop algorithm

Parallel algorithm using tasks

Mandelbrot part

The parallel region is opened, and threads are created the same way as described in previous sections. Here the task is created on each iteration of the first loop (by *j*, height of the image). The number of tasks created and their size is optimal in terms of the performance. The code fragment demonstrating the use of tasks for the Mandelbrot part is shown on Figure 8.

```
#pragma omp parallel num_threads(16) shared(ratio, image, pixels_inside_arr)
#pragma omp single nowait
for (int j = 0; j < h; ++j)
{
    #pragma omp task shared(ratio, image, pixels_inside_arr) firstprivate(j, pixel)
    for (int i = 0; i < w; ++i)
    {
        double dx = (double)i / (w)*ratio - 1.10;
        double dy = (double)j / (h)*0.1 - 0.35;
```

Figure 8 – Parallel using tasks Mandelbrot part

Convolution part

For the convolution part the start of parallel region and threads creation is happening the same way as described for task loop algorithm. The tasks are created to process each pixel, thus the **omp task** construct is defined inside the loops by image height and width. This way the performance of the algorithm was the best. The code showing the part of the implementation of task-based convolution is presented on Figure 9.

```
#pragma omp parallel num_threads(16) shared(nsteps, src, dst)
#pragma omp single nowait
for (int step = 0; step < nsteps; ++step)
{
    for (int ch = 0; ch < channels; ++ch)
    {
        for (int i = 0; i < h; ++i)
        {
            #pragma omp task shared(nsteps, src, dst) firstprivate(i, ch, step)
            for (int j = 0; j < w; ++j)
            {
                double val = 0.0;
                // Reduce the range from -displ to +displ to remove unnecessary if clauses
                int k_start = i < displ ? -i : -displ;
                int k_end = i >= h - displ ? h - i - 1 : displ;
                for (int k = k_start; k <= k_end; ++k)
```

Figure 9 – Parallel using tasks Convolution part

Results

```
[a12134146@alma05 ex2]$ make compile_par_task
[a12134146@alma05 ex2]$ make test_run
Mandelbrot time: 0.321367
Total Mandelbrot pixels: 1478025
Convolution time: 0.736519
Total time: 1.05789

Mandelbrot time: 0.33393
Total Mandelbrot pixels: 1478025
Convolution time: 0.780139
Total time: 1.11407

Mandelbrot time: 0.335463
Total Mandelbrot pixels: 1478025
Convolution time: 0.724012
Total time: 1.05948

Mandelbrot time: 0.319263
Total Mandelbrot pixels: 1478025
Convolution time: 0.73089
Total time: 1.05015

Mandelbrot time: 0.325856
Total Mandelbrot pixels: 1478025
Convolution time: 0.719313
Total time: 1.04517
```

```
Mandelbrot time: 0.334533
Total Mandelbrot pixels: 1478025
Convolution time: 0.734049
Total time: 1.06858

Mandelbrot time: 0.324088
Total Mandelbrot pixels: 1478025
Convolution time: 0.74659
Total time: 1.07068

Mandelbrot time: 0.335265
Total Mandelbrot pixels: 1478025
Convolution time: 0.759258
Total time: 1.09452

Mandelbrot time: 0.327909
Total Mandelbrot pixels: 1478025
Convolution time: 0.862318
Total time: 1.19023

Mandelbrot time: 0.249414
Total Mandelbrot pixels: 1478025
Convolution time: 0.644115
Total time: 0.893529

Mandelbrot: 0.3207088
Convolution: 0.7437203000000001
[a12134146@alma05 ex2]$ make cmp_task
[a12134146@alma05 ex2]$
```

Figure 10 – Test run of the task-based algorithm

The test runs of the parallel using tasks algorithm are shown on Figure 10.

The execution time of Mandelbrot part is varying from 0.25 to 0.33 seconds. The average time is 0.32 seconds. The algorithm is 16.94 times faster than its sequential version.

The execution time of Convolution part is varying from 0.64 to 0.86 seconds. The average time is 0.74 seconds. The algorithm is 10.65 times faster than its sequential version.

Conclusion

The Table 2 shows the results on final algorithms performance.

Algorithms	Execution time	Speed Up
Task loop Mandelbrot	0.31 s	17.48
Task Loop Convolution	0.76 s	10.37
Task Mandelbrot	0.32 s	16.94
Task Convolution	0.74 s	10.65

Table 2 – Results on parallel algorithms performance

As we can see both parallel versions have good almost identical performance which fits the requirements for the assignment.

The speed up graph showing the major changes in performance of the algorithm during development is presented on Figure 11.

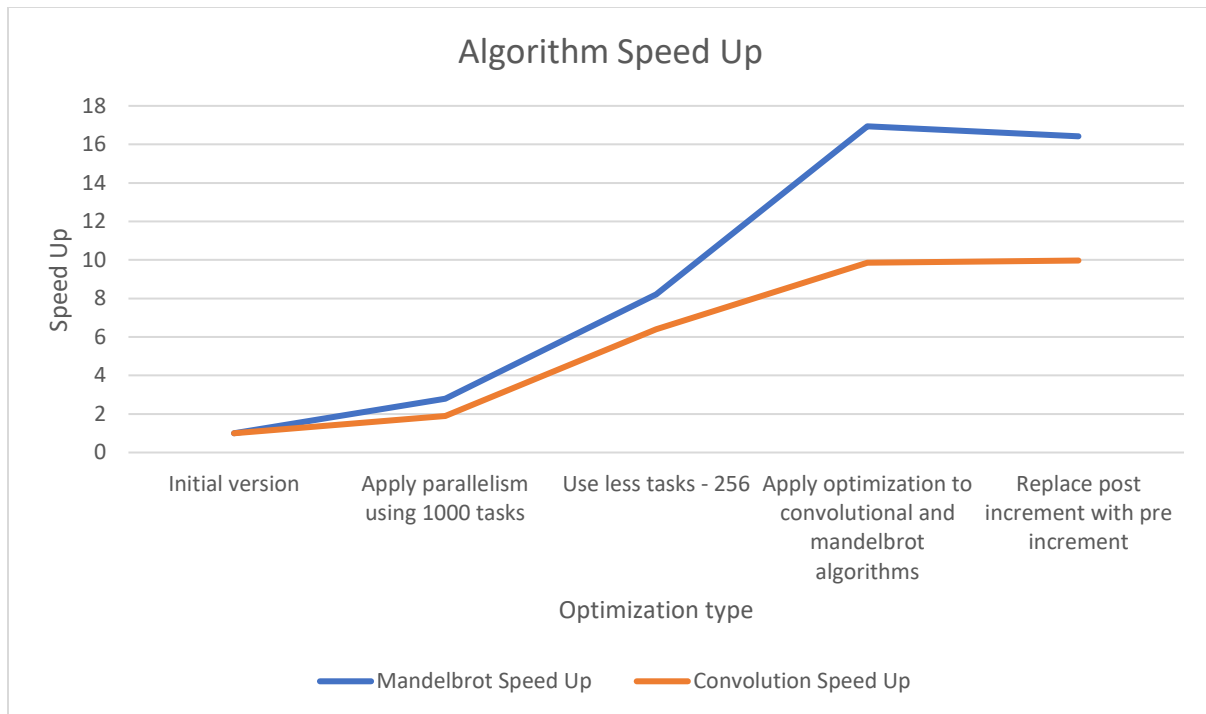


Figure 11 – Speed Up Graph

We may conclude that adding parallelism helped archive a good speed up in performance. But algorithms optimization gave almost the same gain in performance. The most powerful optimizations occurred to be Convolutional algorithm optimization in terms of filtering bounds and complex number operations optimization.

The Speed Up and execution time of the Mandelbrot and Convolution parts of the algorithm in taskloop-based program depending on the number of tasks is shown on Figure 12 and Table 3.

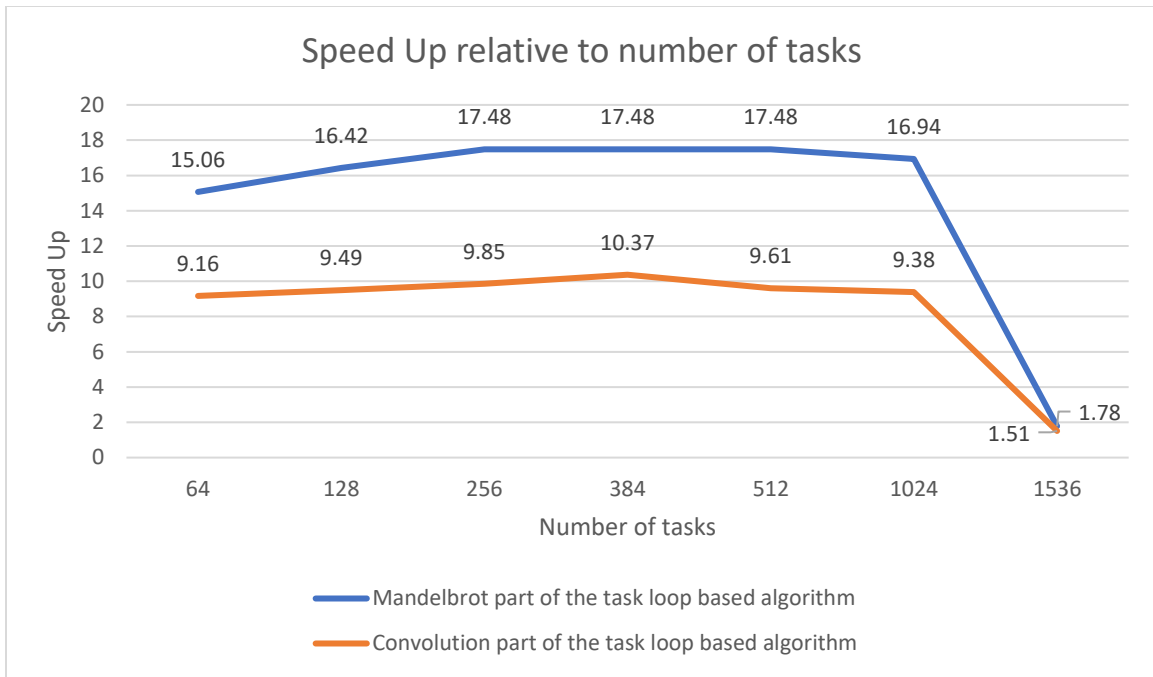


Figure 12 – Speed Up relative to the number of tasks

Number of tasks	Taskloop-based Mandelbrot		Taskloop-based Convolution	
	Time	Speed Up	Time	Speed Up
64	0.36 s	15.06	0.86 s	9.16
128	0.33 s	16.42	0.83 s	9.49
256	0.31 s	17.48	0.80 s	9.85
384	0.31 s	17.48	0.76 s	10.37
512	0.31 s	17.48	0.82 s	9.61
1024	0.32 s	16.94	0.84 s	9.38
1536	3.05 s	1.78	5.23 s	1.51

Table 3 – Speed Up and Execution time relative to the number of tasks

For the Mandelbrot part the optimal number of tasks lays between 256 and 512, the best value after multiple tests occurred to be 256 tasks. The best performance for Convolution part was met while using 384 tasks.

Taking more than 1024 tasks causes significant performance slowdown and overhead for both algorithms because tasks become too small, and more time is spent on switching between them. When tasks are larger, which means that more iterations are performed by each task, this overhead becomes smaller, and each task is busy during larger amount of time. Too small number of tasks means that the tasks become bigger, and it causes performance slow down as well because the work cannot be fairly divided between them, so smaller tasks wait for larger tasks.