Contents

OpenMP exercise set

Fotis Bistas
AM: 3190135
uni_email: p3190135@aueb.gr
personal_email: fotis.bistas@gmail.com

May 2023

1 Introduction

The first set of exercises in this course focuses on parallel programming using OpenMP. Specifically, I will be parallelizing two programs: one that finds prime numbers and another that performs a Gaussian blur on an image. Parallel programming allows us to speed up the execution time of these programs by dividing the work among multiple processors or cores. Through these exercises, I will learn how to use OpenMP directives and features to write efficient parallel code and gain experience in optimizing performance.

1.1 Setup

Metrics were conducted on the following system:

Computer	Processor	Cores	Compiler
DESKTOP-SIL3NU4	AMD Ryzen 5 3600	6	gcc 11.3.0

and more specifically in the WSL(windows subsystem for linux) virtual machine using Ubuntu 22.04.2 LTS.

2 Exercise 1

2.1 The problem

The program I will be parallelizing using OpenMP is designed to count all prime numbers up to a given value n and find the largest prime number up to n. The program uses a mathematical algorithm that iteratively checks if each odd number is divisible by any of its previous odd numbers. If an odd number is not divisible, it is considered a prime number, and the program increments a counter for the number of primes found so far and stores the largest prime number found. The serial implementation of this program can be computationally

expensive for large values of n, which makes it a good candidate for parallelization. Through this exercise, I will learn how to use OpenMP directives to divide the loop among multiple threads and optimize the performance of the program for efficient computation of prime numbers.

2.2 Methodology

The for loop of the primes program was parallelized using the following command:

```
//static scheduling is best here since I know how to best divide
    the work betIen threads
#pragma omp parallel for schedule(static,chunk) default(none)
    firstprivate(n) private(num, divisor,remainder,quotient)
    reduction(+:count) lastprivate(lastprime)
for (i = 0; i < (n-1)/2; ++i) {     /* For every odd number */
    num = 2*i + 3;

    divisor = 1;
    do
...
```

Listing 1: Primes

My thought process for parallelizing the primes algorithm was the following:

Some variables such as num, divisor, remainder, and quotient are iteration independent. Since the default mode for **OpenMP** variables is shared, this introduces a lot of overhead for the parallel algorithm. Declaring them as private, using the private clause, speeds up the process by a significant amount.

This is not enough. The variable count must be shared among threads and the threads must synchronize to update it properly. I figured out the most efficient way to go about this is to use a reduction which: creates a private instance of count for each thread and proceeds to combine it from all the threads to obtain the correct value.

Furthermore, the lastprime variable must receive the value of the last iteration of the loop. This is easily done using the **lastprivate** directive, and reduces the overhead of having the threads share the lastprime variable.

A thing to note is that n comes from the preprocessor variable UPTO, which is set here in most cases at 1000000. If it is changed I will mention it.

2.2.1 Scheduling policy

I chose to use static scheduling since I knew how to best divide the work between threads, which is best in this case since the workload is known in advance. Overall, these choices helped to speed up the algorithm and increase its efficiency.

Additionally and most importantly: with static scheduling I can ensure what thread must be responsible for assigning a value to the lastprime variable. The thread that executes the last part of the iteration. This can be done with the **lastprivate** directive.

Using dynamic/guided scheduling produces better time results but there is no way to tell, what is the correct thread to assign a value to lastprime. The better time results can attributed to the fact that openMP chosees the best courses of action at runtime.
Dynamic Scheduling:

Threads	first run	second run	third run	fourth run	average
4	2.791431	2.868791	2.651643	2.918657	2.8076305

Guided Scheduling:

Threads	first run	second run	third run	fourth run	average
4	2.908008	3.308528	2.713370	2.650496	2.8951005

However both produce results where the last is wrong(UPTO=10000000):

Count	Last	Time
664579	9999433	2.713370
664579	9999973	2.650496
664579	9999943	3.308528

2.2.2 Chunking using static scheduling

Experimenting with different chunk sizes using static scheduling does seem to have an effect on performance. This is for UPTO=10000000.
For example using a chunk size of 100:

Threads	first run	second run	third run	fourth run	average
4	3.263508	3.529811	3.049428	3.131362	3.24352725

Or using a chunk size of 1000:

Threads	first run	second run	third run	fourth run	average
4	3.114033	2.972887	3.293254	3.731384	3.2778895

Or using a chunk size of 10000:

Threads	first run	second run	third run	fourth run	average
4	3.279579	3.075843	3.049428	2.749036	3.0384715

Or using a chunk size of 100000:

Threads	first run	second run	third run	fourth run	average
4	2.515437	3.208725	2.456637	2.850294	2.75777325

Finally using a chunk size of 1000000(which is around the same as the default one):

Threads	first run	second run	third run	fourth run	average
4	3.756118	3.267465	3.632437	3.725072	3.595273

There seems to be an increase in performance with the increase in chunk size up to a certain point:

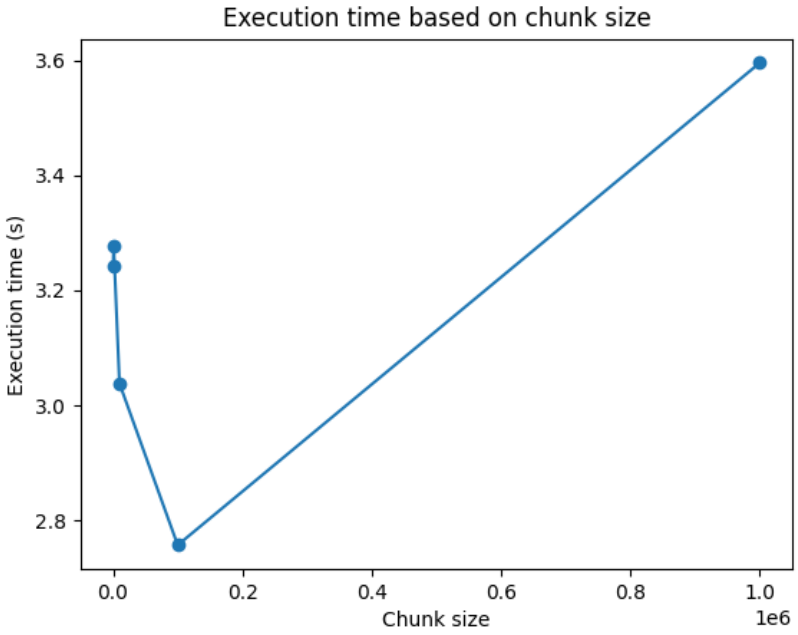

Figure 1: Execution time of prime number program with varying chunk sizes, given UPTO=1000000 and threads 4.

Since the chunk size efficiency seems to depend on the value of UPTO we modify the code to accommodate for that fact:

```
#define UPTO 10000000
//since we are using a chunk size we use the following empirical
    rule
#define CHUNK_SIZE (int)(UPTO * 0.01)
```

Listing 2: Modified code for using chunks

2.3 Results

To measure the time it took for the program to complete, I used the OpenMP library function **omp_get_wtime()**. The function returns a double precision value equal to the number of seconds since the initial value of the operating system real-time clock.

To specify the number of threads to be used, I set the environment variable **OMP_NUM_THREADS** before running the binary. For example, to use 4 threads, I set export **OMP_NUM_THREADS=4** in the command line.

Finally, I compiled the program using the following command: **gcc -o [name] -fopenmp primes.c**, where [name] is the desired name of the output binary file. The below is the output of the serial program:

Serial	first run	second run	third run	fourth run	average
	8.552580	8.522516	8.537302	8.522466	8.55820875

2.3.1 No chunk size threaded program

The below is the output of the threaded program:

Threads	first run	second run	third run	fourth run	average
1	8.586801	8.567877	8.548594	8.529563	8.55820875
2	5.408603	6.428148	7.142996	5.379335	6.0897705
3	5.018446	3.773073	4.749844	5.012645	4.638502
4	4.018007	3.549024	3.393004	2.937119	3.4742885

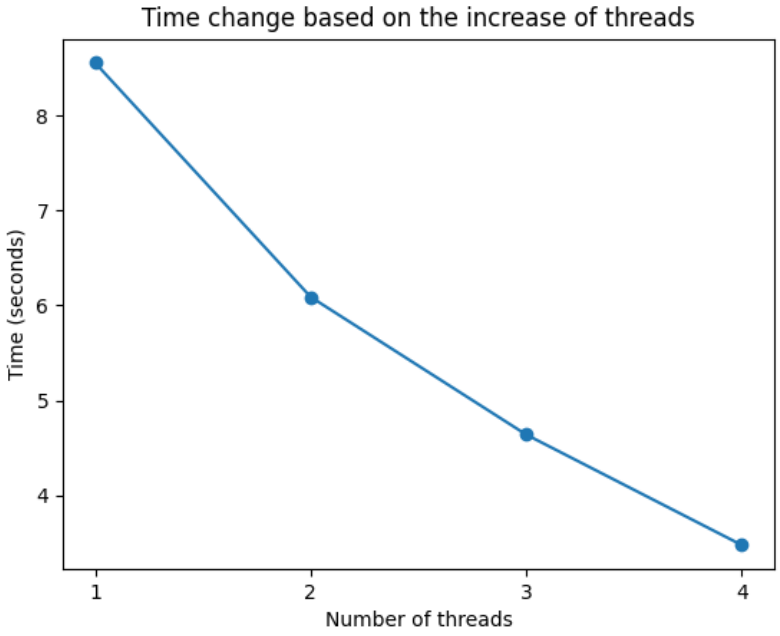

Figure 2: Execution time of prime number program with varying number of threads.

2.3.2 Using chunk size

The below is the output of the threaded program:

Threads	first run	second run	third run	fourth run	average
1	8.540301	8.692581	8.628131	8.583465	8.6111195
2	4.378742	4.368210	4.645062	4.491790	4.470951
3	3.430615	3.991035	4.167950	3.511062	3.7751655
4	3.376941	3.242566	3.352307	3.255134	3.306737

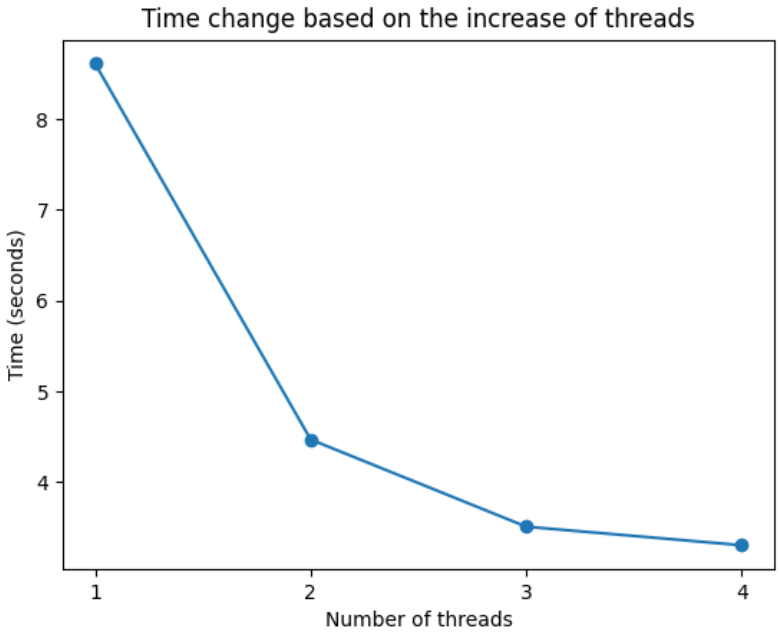

Figure 3: Execution time of prime number program with varying number of threads.

2.3.3 Comparison chunk size and non

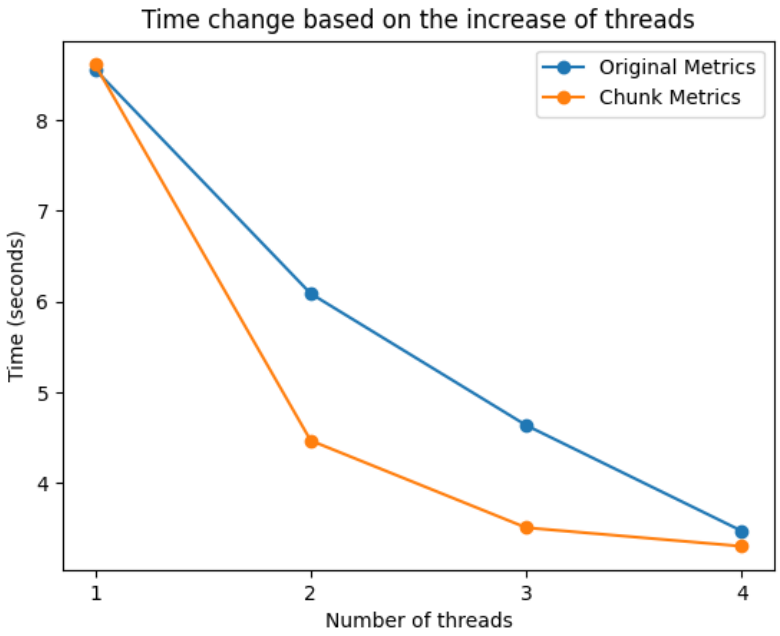

Figure 4: Comparison between the execution time using the a custom chunk scheduling and using the automatic chunk scheduling.

2.3.4 Observations

It can observed that as the number of threads increases, the execution time decreases. In fact the increasing of threads (up to a certain number as to not overload the system), the execution time $N \to \frac{1}{K}$ where K is the number of threads. This essentially represents how well the threads distribute the workload of the sequential program.

The above is achieved using the static scheduling feature of OpenMP, which divides the iterations of the loop equally among the threads. This ensures that each thread receives the same amount of work, resulting in a balanced workload distribution. However, using the default chunk size may not always yield optimal performance. Taking advantage of the empirical rule I found for custom chunk sizes, which suggests using medium-sized chunks, the performance is improved. This is actually the case for smaller and larger UPTOs (for very small UPTOs the serial algorithm is better).

I think this can attributed to the fact that a medium sized chunk combines the benefits of a smaller chunk and a larger chunk. On one hand, a medium-sized chunk can help balance the workload among threads, similar to a small

chunk. This can reduce idle time for threads, leading to better efficiency. A medium-sized chunk can also help improve cache locality, similar to a small chunk. This means that the data accessed by each thread is more likely to be stored in the cache, which can lead to faster access times and better performance. On the other hand, a medium-sized chunk can also help reduce the overhead associated with parallelization, similar to a large chunk. When the chunk size is too small, the overhead per chunk can become a significant portion of the overall processing time, reducing performance gains. By using a medium-sized chunk, the overhead per chunk can be reduced while still maintaining good load balancing and cache locality.

3 Exercise 2

3.1 The problem

The program applies Gaussian blur filtering to a BMP image using OpenMP for parallel processing. It reads in the BMP file and separates the RGB channels using utility functions. The Gaussian blur filter is applied using either a serial or parallel function depending on the mode of operation. The output image is created by cloning an empty image structure and calculating each pixel using a weighted average of its surrounding pixels based on the Gaussian kernel. Finally, the RGB channels are combined and written to a BMP file.

3.2 Methodology

3.2.1 Parallel loops

The parallel loops implementation:

```
omp_set_dynamic(0);
#pragma omp parallel for schedule(dynamic) firstprivate(redSum,
    greenSum,blueSum,weightSum,radius,imgout,width,height) private(
    i,j,row,col) default(none) shared(imgin) if(height >= width)
for (i = 0; i < height; i++)
{

    #pragma omp parallel for schedule(dynamic) firstprivate(redSum,
        greenSum,blueSum,weightSum,radius,imgout,width,height,i)
        private(j,row,col) default(none) shared(imgin) if(height <
         width)
    for (j = 0; j < width ; j++)
    {
...
```

Listing 3: Parallel loops for gaussian blur

3.2.1.1 Code

I attempted to parallelize the Gaussian blur algorithm using OpenMP, in many different ways. I initially tried a simple parallelization of only one for loop. Then I tried using two parallel for loop directives to parallelize the rows and columns of the image separately. However, I did not observe significant performance gains with this approach, likely due to load imbalance and synchronization overhead between the two loops.

As an alternative, I settled on using a conditional for loop approach, where the outer directive will fire only if the height is larger or equal to the width and the inner directive will fire otherwise. This approach allows the code to check what's the best loop to parallelize, leading to improved performance compared to the single or the two-loop approach.

3.2.1.2 Scheduling policy

In order to determine the best scheduling strategy for our implementation of the Gaussian blur algorithm using OpenMP, conducted a series of experiments comparing static, dynamic, and guided scheduling. tested each scheduling strategy on a range of image sizes and blurring radii, and recorded the execution time and speedup achieved. Here are some results:
Static scheduling:

threads	first run	second run	third run	fourth run	average
4	12.903186	8.473524	12.020314	9.254618	10.6629105

Dynamic scheduling:

threads	first run	second run	third run	fourth run	average
4	10.545440	9.552152	10.862304	9.931808	10.222926

Guided scheduling:

threads	first run	second run	third run	fourth run	average
4	10.361456	11.226618	10.623737	11.732317	10.986032

The results indicate that dynamic scheduling outperformed, although not by a lot,both static and guided scheduling across all tested configurations. Dynamic scheduling seems to have provided more flexibility, adaptability and consistency in balancing the workload across threads, which helped to minimize load imbalance and improve parallel performance. Based on these results, I chose to use dynamic scheduling for parallelization of the Gaussian blur algorithm in our implementation.

3.2.2 Tasks

In the experiments with implementing a Gaussian blur algorithm using OpenMP, I tested the performance of tasks when the mathematical operation was divided into tasks per row of the image. I found that when each row was assigned to one task, the performance was almost the same as that of using parallel loops, which was slower than the current implementation:

```
#pragma omp parallel private(i,j,row,col) firstprivate(weightSum,
    redSum,greenSum,blueSum,radius,imgout,width,height) default(
    none) shared(imgin)
#pragma omp single
for (i = 0; i < height; i++)
{

    #pragma omp task firstprivate(i,weightSum,redSum,greenSum,
        blueSum,radius,imgout,width,height) private(j,row,col)
        default(none) shared(imgin)
    {
            for (j = 0; j < width ; j++)
            {
                    #pragma omp task firstprivate(i,radius,imgout,
                        width,height,j) private(row,col) default(
                        none) shared(imgin,weightSum,redSum,
                        greenSum,blueSum)
                    {
                            for (row = i-radius; row <= i + radius;
                                row++)
                            {
                                for (col = j-radius; col <= j +
                                    radius; col++)
                                {
                                    ...
                                }
                            ...
                            }
                    }
                    #pragma omp taskwait
                    ...
            }
            ...
    }

...
```

Listing 4: Task implementation for gaussian blur

With the above implementation I noticed significant improvements in performance. This is probably because tasks provide a more flexible approach to parallelism, allowing the runtime system to better manage load balancing and scheduling of work across threads. By dividing the mathematical operation (meaning these two inner for loops) into tasks, the potential for load imbalance seems to be reduced, although it probably has a lot of overhead in memory,and achieve better performance overall. Here are some metrics:

One task implementation:

threads	first run	second run	third run	fourth run	average
4	9.895203	10.562432	10.262486	10.669127	10.347312

Two tasks implementation:

threads	first run	second run	third run	fourth run	average
4	5.010607	5.167795	5.297747	5.164225	5.1600935

3.3 Results

I compile the program for Gaussian blur with OpenMP support, you can use the following command: **gcc -o ./bin/blur -fopenmp gaussian-blur.c -lm**.Before running the program, I set the number of threads to use with the following command: **export OMP_NUM_THREADS=[num_threads]**.

I run the program, you can use the following command: **./bin/blur r image_name.bmp**, where **r**, is the radius of the blur and **1500.bmp** is the name of the input image file. By convention I only run the program for **r=8** and **image_name = 1500.bmp**.The program will apply Gaussian blur with the specified radius to the input image and output the result to a new file named "1500-r8-method[serial,tasks,loops].bmp".
The serial output looks like this:

serial	first run	second run	third run	fourth run	average
	34.430838	36.922534	38.176947	38.738349	37.067167

3.3.1 Parallel loops

This is the output of the program using parallel loops:

Threads	first run	second run	third run	fourth run	average
1	36.879000	36.674827	36.537232	35.228685	36.329936
2	20.623917	18.523448	20.760018	20.007655	19.9787595
3	15.132231	12.876375	13.298183	13.658077	13.7412165
4	10.637261	9.747301	10.023800	9.244302	9.913166

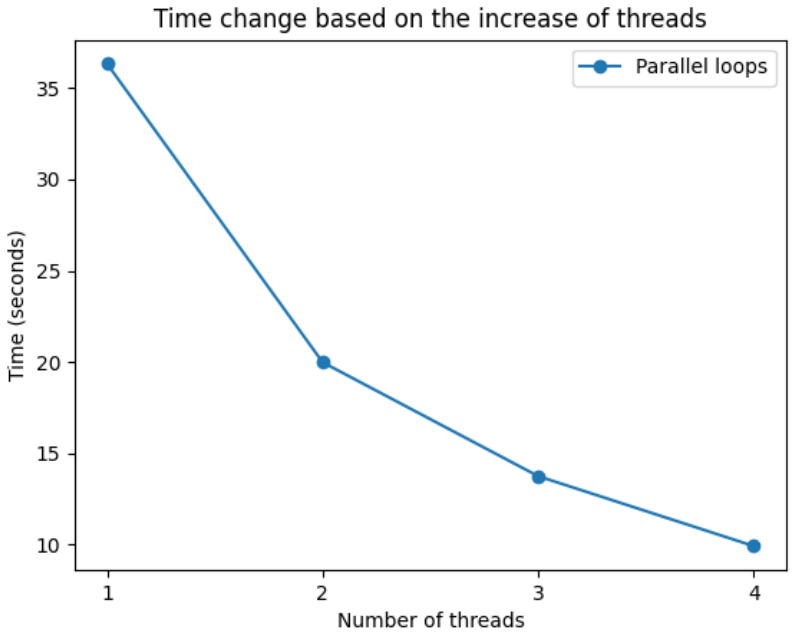

Figure 5: Performance of the algorithm using parallel loops.

3.3.2 Tasks

This is the output of the program using tasks:

Threads	first run	second run	third run	fourth run	average
1	37.477622	41.118454	41.403998	35.866943	38.96675425
2	9.773049	9.680708	9.456276	9.567520	9.61938825
3	6.913972	6.639000	6.605483	7.040099	6.7996385
4	5.063666	5.049521	5.008849	5.203934	5.0814925

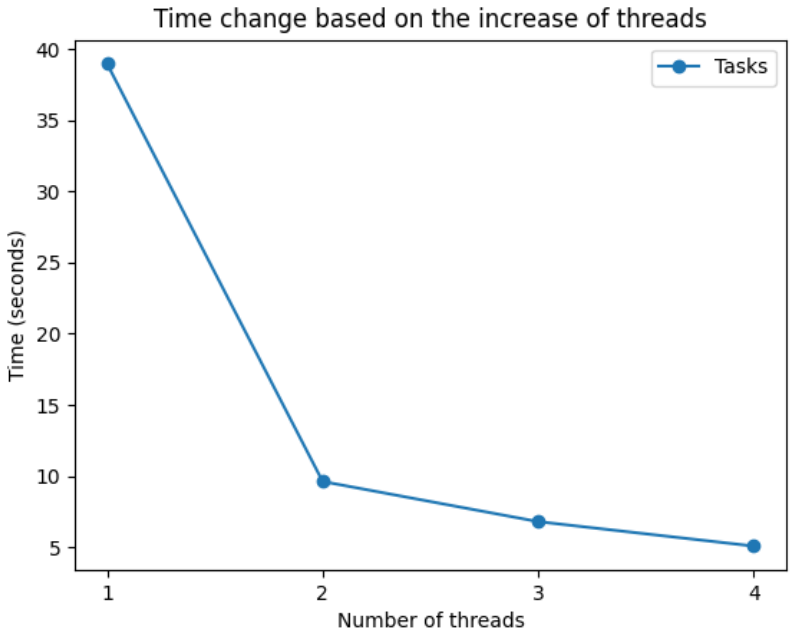

Figure 6: Performance of the algorithm using tasks.

3.3.3 Observations

Comparison between parallel loops and tasks:

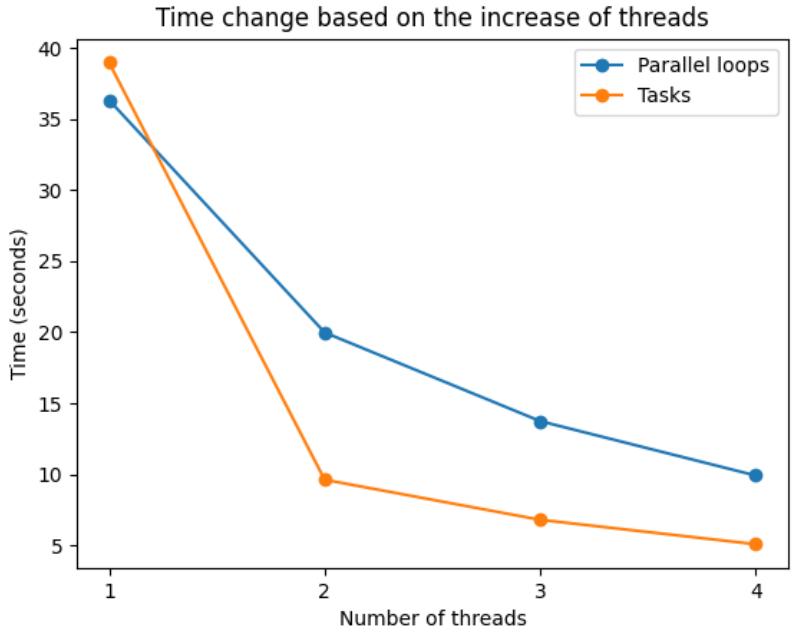

Figure 7:

It can observed that as the number of threads increases, the execution time decreases. In fact in the parallel loops implementation, increasing of threads (up to a certain number as to not overload the system), the execution time $N \to \frac{1}{K}$ where K is the number of threads. Parallel loops provide performance gains over the serial implementation.

However, when I switched to using tasks, I noticed even better improvements in performance across all image sizes and radii. This is likely because tasks provide a more flexible and dynamic approach to parallelism, allowing the runtime system to better manage load balancing and scheduling of work across threads. By contrast, parallel loops require explicit partitioning of the work, which can be less effective in managing load imbalance, or parallel loops might introduce more overhead and probably can be more difficult to schedule etc.

Despite dynamic scheduling in parallel loops being more flexible and faster than static scheduling, tasks seems to be faster in this case. Overall, my experiments suggest that using tasks is the most effective approach for parallelizing the algorithm in OpenMP.