Behrad Hemati
11849389

Assignment 2: OpenMP

In this assignment, we need to parallelize an already existing sequential code which produces a Mandelbrot image and blurs it. We need to produce at least two different versions of the code using **task** and **taskloop**.

Part1: Task

Code Explanation

In this part, in order to parallelize, I took the following measures.

Mandelbrot:

```
auto t1 = omp_get_wtime();
// Generate the mandelbrot set
// Use OpenMP tasking to implement a parallel version

int task_size = 256;
vector<int> pixels_inside_local(task_size, 0);
#pragma omp parallel default(none) firstprivate(task_size,ratio) shared(image,pixels_inside_local) num_threads(16)
{
    int nthreads = omp_get_num_threads();
    #pragma omp single
    {
        for (int i = 0; i<task_size; i++){
            #pragma omp task default(none) firstprivate(task_size,ratio,i) shared(image,pixels_inside_local)
            pixels_inside_local[i]=(mandelbrot(image, ratio, i, task_size));
        }
    }
}
pixels_inside = std::accumulate(pixels_inside_local.begin(), pixels_inside_local.end(), 0);
auto t2 = omp_get_wtime();

std::cout << "Mandelbrot time: " << chrono::duration<double>(t2 - t1).count() << endl;
std::cout << "Total Mandelbrot pixels(1478025): " << pixels_inside << endl;
```

In the main() function I have put the mandelbrot function in an openmp task to generate the tasks necessary, and to handle the pixels_inside variable, I have made a local vector for each task and and in the end sum the vector to get the total pixels_inside variable. The variable that had to be made shared are Image and pixels_inside_local

Behrad Hemati
11849389

```
int mandelbrot(Image &image, double ratio = 0.15, int task_num=1, int task_size=1)
{
    int i, j;
    int h = image.height;
    int w = image.width;
    int channels = image.channels;
    ratio /= 10.0;
    int pixels_inside=0;

    // pixel to be passed to the mandelbrot function
    vector<int> pixel = {0, 0, 0}; // red, green, blue (each range 0-255)
    complex<double> c;

    int w_size = w/task_size;
    int h_size = h/task_size;
    for (i = task_num * w_size; i < (task_num+1) * w_size; i++)
    {
        for (j = 0; j < h; j++)
        {
            double dx = (double)i / (w)*ratio - 1.10;
            double dy = (double)j / (h)*0.1 - 0.35;

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

            if (mandelbrot_kernel(c, pixel)) // the actual mandelbrot kernel
                pixels_inside++;

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

    return pixels_inside;
}
```

And the only change to the mandelbrot function was to pass the task_num and task_size variables in order to calculate the chunk that each task should calculate, which I have used for the width of the picture since it is a bigger value, and it might help to parallelize on width as opposed to height.

```
[a11849389@alma06 a2]$ diff mandelbrot-task.ppm mandelbrot-original.ppm
[a11849389@alma06 a2]$
```

In the end, using the diff command on linux, we can verify the the output in indeed correct, the same comparison is also done for the pixels_inside.

Behrad Hemati
11849389

Convolution:

```
void convolution_2d(Image &src, Image &dst, int kernel_width, double sigma, int nsteps=1)
{
    int task_size = 256;
    for (int step = 0; step < nsteps; step++)
    {
        #pragma omp parallel num_threads(16) default(none) firstprivate(task_size,nsteps,sigma,kernel_width) shared(dst,src)
        {
            #pragma omp single
                {
                for (int i = 0; i<task_size; i++)
                {
                    #pragma omp task default(none) firstprivate(i,task_size,nsteps,sigma,kernel_width) shared(dst,src)
                    convolution_2d_helper(src, dst, kernel_width, sigma, nsteps, i, task_size);
                }
            }
        }
        if ( step < nsteps-1 ) {
            // swap references
            // we can reuse the src buffer for this example
            Image tmp = src; src = dst; dst = tmp;
        }
    }
}
```

For the convolution part, I made a new function called convolution_2d_helper in order to make the parallelism easier and more readable. I have made each step parallel and then swapping the references is done outside the parallel region since each step needs to be completed before the references are swapped, and since this is outside the parallel region there is no need for synchronization since this is done by the implicit barrier at the end of the parallel region. Inside the helper function, parallelization is done the same way is the mandelbrot part, in which the image is parallelized along the width. Src and dst need to be shared since in each step, the tasks need to have access to the already convoluted picture since convolution needs to be done 20 times.

Speed-up comparisons

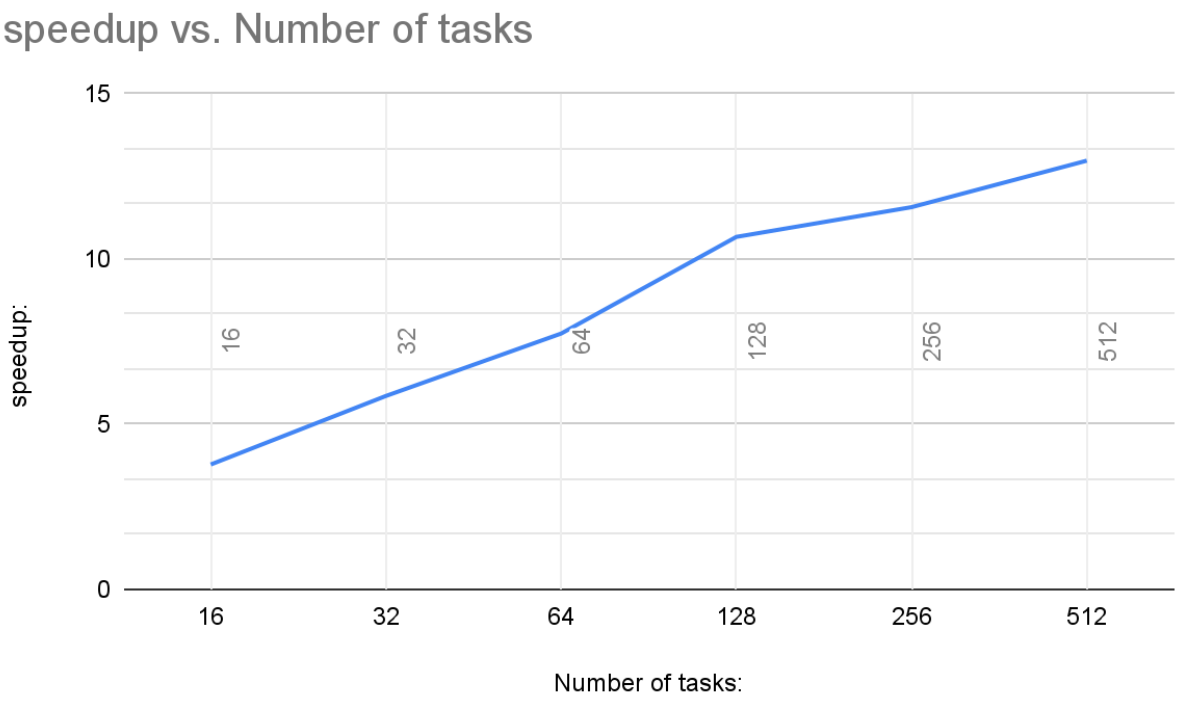
For the comparison of speed-ups in this task I have calculated the results using different task sizes and after running the code multiple times I have included the best result I have seen here, as a result these results are not very accurate because ALMA is a shared environment and it having exclusive access to it is almost impossible since there is always some activity on it and since it takes less than a second to run this program, any amount of activity will affect the output to a large degree. The codes are run with **16 Threads** for both of the functions.

Mandelbrot:

Number of tasks:	Absolute Tims in s:	speedup:
16	1.27	3.779527559
32	0.82	5.853658537
64	0.62	7.741935484
128	0.45	10.66666667
256	0.415	11.56626506

Behrad Hemati
11849389

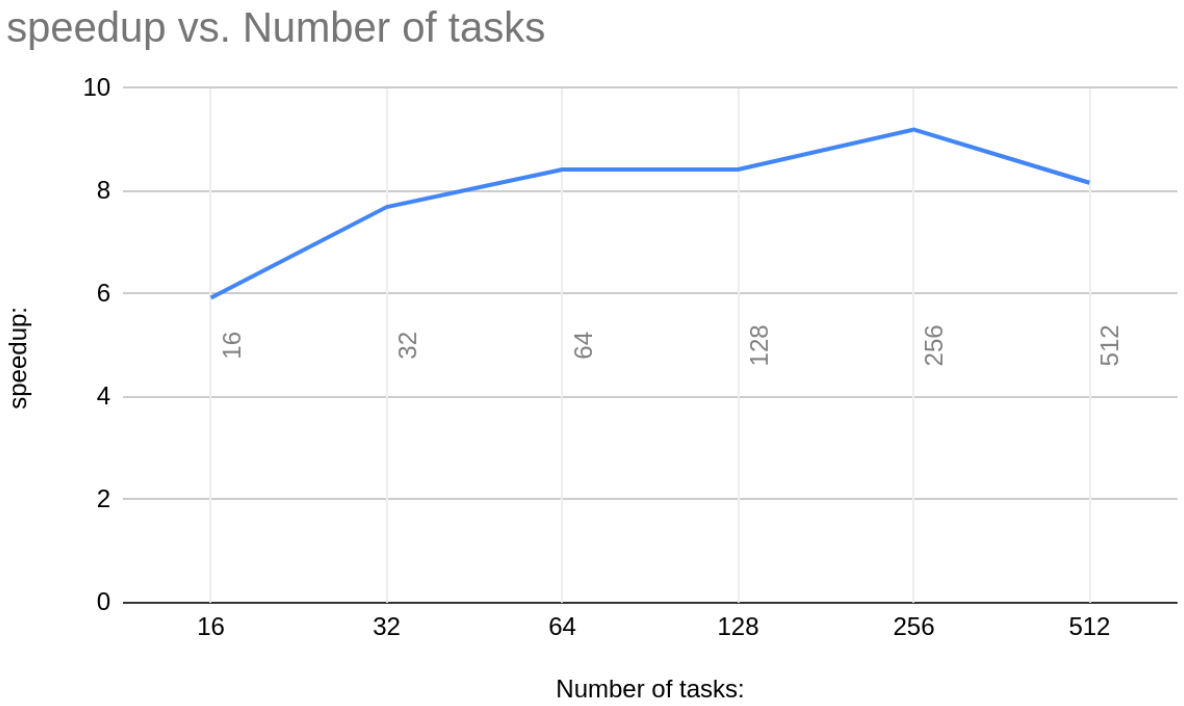
512	0.38	12.63157895



Convolution:

Number of tasks:	Absolute Tims in s:	speedup:
16	1.35	5.925925926
32	1.04	7.692307692
64	0.95	8.421052632
128	0.95	8.421052632
256	0.87	9.195402299
512	0.98	8.163265306

Behrad Hemati
11849389



We can see in both convolution and Mandelbrot, when the number of tasks are between 128 and 512, the times are pretty similar. Even though in these carts there maybe some difference, it comes down to the aforementioned problems of not having exclusive access to the system and run-to-run variants are to be expected here.

Task 2: Taskloop

The second challenge was to parallelize the same code using taskloop instead of task, which is easier to some extent since the creation of tasks is no longer by the programmer and the openmp API handles the creation of tasks depending on two variables we can control, **grainsize** and **num_tasks**. **Grainsize** controls the size of each chunk, while **num_tasks** would be very similar to the last challenge and handles the number of tasks created.

Mandelbrot:

```
#pragma omp parallel num_threads(16) default(none) firstprivate(ratio) shared(image, pixels_inside)
{
    #pragma omp single
    pixels_inside = mandelbrot(image, ratio);
}
auto t2 = omp_get_wtime();
```

Behrad Hemati
11849389

```
#pragma omp taskloop num_tasks(512) default(none) firstprivate(i,w,ratio,h,c,pixel,channels) shared(image) reduction(+: pixels_inside)
    for (j = 0; j < h; j++)
    {
        for (i = 0; i < w; i++)
        {
            double dx = (double)i / (w)*ratio - 1.10;
            double dy = (double)j / (h)*0.1 - 0.35;

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

            if (mandelbrot_kernel(c, pixel)) // the actual mandelbrot kernel
                pixels_inside++;

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

return pixels_inside;
```

Taskloop is very similar to parallel for, therefore the codes would look very similar, with this directive the creation and propagation of the tasks is done by the compiler and therefor it would also handle arbitrary task numbers and resolution better. The pixels_inside variable is handled by a reduction and at the end of taskloop scope the result is returned. The variable image needs to be shared but since it is an array, the same cells are never accessed by the same tasks and therefor there is no data race.

Convolution:

```
auto t3 = omp_get_wtime();
#pragma omp parallel default(none) shared(filtered_image,image) num_threads(16)
{
    #pragma omp single
    convolution_2d(image, filtered_image, 5, 0.37, 20);

}
auto t4 = omp_get_wtime();
```

Behrad Hemati
11849389

```
int displ = (kernel.size() / 2); // height==width!

for (int step = 0; step < nsteps; step++)
{
    #pragma omp taskloop grainsize(8) default(none) firstprivate(displ,w,h,kernel,channels) shared(src,dst)
        for (int i = 0; i < h; i++)
        {
            for (int j = 0; j < w; j++)
            {
                for (int ch = 0; ch < channels; ch++)
                {
                double val = 0.0;

                for (int k = -displ; k <= displ; k++)
                {
                    for (int l = -displ; l <= displ; l++)
                    {
                        int cy = i + k;
                        int cx = j + l;
                        int src_val = 0;

                        // if it goes outside we disregard that value
                        if (cx < 0 || cx > w - 1 || cy < 0 || cy > h - 1)
                        {
                            continue;
                        }
                        else
                        {
                            src_val = src(ch, cy, cx);
                        }

                        val += kernel[k + displ][l + displ] * src_val;
                    }
                }
                dst(ch, i, j) = (int)(val > 255 ? 255 : (val < 0 ? 0 : val));
            }
        }
    }
    if (step < nsteps - 1)
    {
        // swap references
        // we can reuse the src buffer for this example
        Image tmp = src;
        src = dst;
        dst = tmp;
    }
}
```

Similarly for each step I have made taskloops and since the scope of the taskloop is the same as the first for it parallelizes, the swap reference is done by a single thread and therefore there is no data race, the shared variables here are **src** and **dst** which are of type **Image** and similar to the last section, since each task only accesses certain addresses, there is no data race.

Speed-up Comparisons:

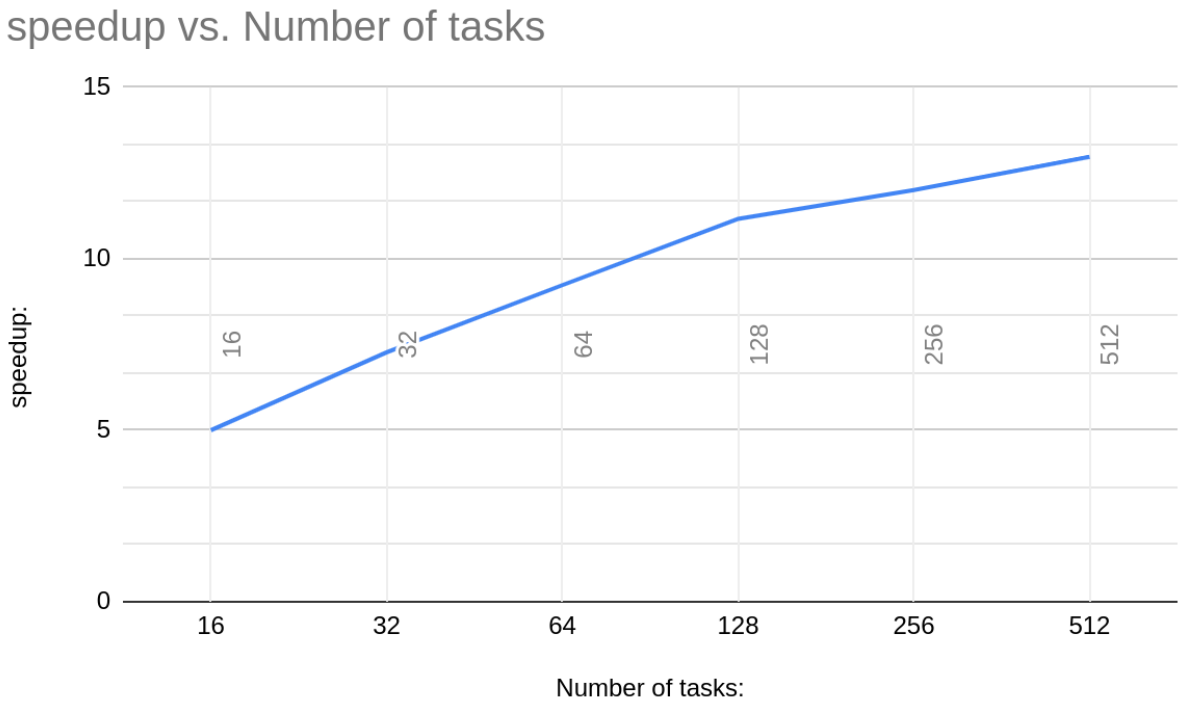
For this part, for mandelbrot I have used **num_tasks** which are the same as specifying a task size in the previous part, and for the sake of comparison, for the convolution part I have used **grainsize**.

Mandelbrot:

Number of tasks:	Absolute Tims in s:	speedup:
16	0.96	5

Behrad Hemati
11849389

32	0.66	7.272727273
64	0.52	9.230769231
128	0.43	11.1627907
256	0.4	12
512	0.37	12.97297297

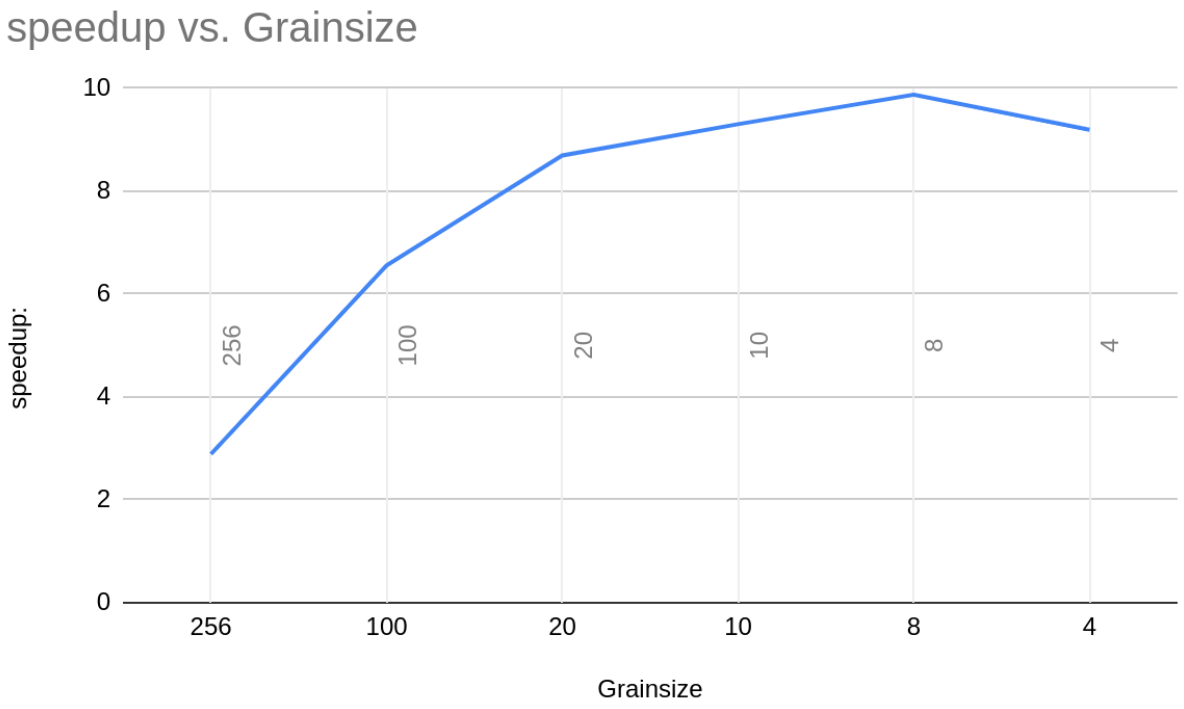


The results for mandelbrot is very similar in this part and the previous one. It seems that the number of tasks need to be higher in order to achieve a better speed up and with lower number of tasks the speed up achieved is significantly lower. The best results are identical in these two parts and eliminating the run-to-run variant I think task size of 256 and 512 would have negligible difference between them.

Convolution

Grainsize	**Absolute Tims in s:**	**speedup:**
256	2.77	2.888086643
100	1.22	6.557377049
20	0.92	8.695652174
10	0.86	9.302325581
8	0.81	9.87654321
4	0.87	9.195402299

Behrad Hemati
11849389



Using **grainsize** here shows the difference between these 2 metrics, they can be seen as opposite to eachother and therefore the lower the **grainsize** the higher the number of tasks. So as an example a grainsize of size 4 is equivalent in this case (width=1536) to a task size of 384, so it is expected to seen the the lower grain sizes would result in better speed-up. For this chart I have reversed the order of grainsizes from high to low.