



Basics of Parallel Computing
2025S
Assignment 2

May 28, 2025

A2 - 3 Person Group 3

1: Veneta GRIGOROVA, 12332287

2: Luc THIERY, 51903906

3: Aya WAHBI, 01427598

1. Exercise 1

1.1. What do a and t count?

The array $a[]$ records which thread executed which iteration of the loop. For each iteration index i , $a[i]$ holds the thread ID (0, 1, or 2) of the thread that processed that iteration.

The array $t[]$ counts how many iterations each thread executed in total. Each time a thread executes an iteration, it increments $t[\text{omp_get_thread_num()}]++$. Thus, $t[0]$ gives the total number of iterations processed by thread 0, $t[1]$ by thread 1, and $t[2]$ by thread 2.

1.2. Values for all elements in a and t

The values depend on the scheduling policy defined by the `OMP_SCHEDULE` environment variable. The complete tables with the possible values for all requested cases are provided in the following section (see Table 1 and Table 2).

Table 1: Thread assignment ($a[]$) for all schedules

Case / a	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
static	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2
static,2	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1
static,3	0	0	0	1	1	1	2	2	2	0	0	0	1	1	1
dynamic,1	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
dynamic,4	0	0	0	0	1	1	1	1	2	2	2	2	0	0	0
guided,7	0	0	0	0	0	0	0	1	1	1	1	2	2	2	2

Table 2: Iteration counts (τ []) for all schedules

Case / τ	Thread 0	Thread 1	Thread 2
static	5	5	5
static,2	6	5	4
static,3	6	6	3
dynamic,1	5	5	5
dynamic,4	7	4	4
guided,7	7	4	4

2. Exercise 2

2.1. Optimal Schedule

The workload per iteration is as follows:

$\text{time[]} = [1, 2, 3, 1, 1, 1, 3, 2, 3, 2, 1, 2, 10, 2, 4, 3, 3, 12]$

We assume $p = 4$ cores and no scheduling overhead. To achieve the minimum makespan, we use Longest Processing Time First (LPT) scheduling. The largest jobs are assigned first to minimize imbalance.

The sorted job times (descending) are:

12, 10, 4, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1

The assignment is as follows:

Table 3: Gantt chart for optimal schedule (Exercise 2.1)

Core	Tasks (time units)
Core 1	12, 3, 1, 1, 1, 1
Core 2	10, 4, 1, 1, 1
Core 3	3, 3, 2, 2, 2, 2, 1, 1
Core 4	3, 3, 2, 2, 2, 2, 1, 1

The resulting workloads are:

- Core 1: 15
- Core 2: 15
- Core 3: 16
- Core 4: 16

The **minimum makespan is 16 time units**, which is the maximum of all core workloads.

2.2. Schedule `static,3`

With `OMP_SCHEDULE=static,3`, blocks of 3 iterations are cyclically assigned to each thread. The assignment is:

Table 4: Gantt chart for `static,3` schedule (Exercise 2.2)

Core	Tasks (time units)
Core 0	1, 2, 3, 10, 2, 4
Core 1	1, 1, 1, 3, 3, 12
Core 2	3, 2, 3
Core 3	2, 1, 2

The resulting workloads are:

- Core 0: 22
- Core 1: 21
- Core 2: 8
- Core 3: 5

The **makespan is 22 time units**, which is the maximum load across all cores.

2.3. Schedule `dynamic,2`

With `OMP_SCHEDULE=dynamic,2`, threads pull chunks of 2 iterations dynamically as they become available. One possible example assignment is shown below:

Table 5: Gantt chart for `dynamic,2` schedule (Exercise 2.3)

Core	Tasks (time units)
Core 0	1, 2, 3, 2, 3, 12
Core 1	3, 1, 1, 2
Core 2	1, 1, 10, 2
Core 3	3, 2, 4, 3

The resulting workloads are:

- Core 0: 23
- Core 1: 7
- Core 2: 14
- Core 3: 12

The **makespan is 23 time units**, which is the maximum load across all cores.

3. Exercise 3

3.1. What is the output of the three different versions?

We experimentally tested all three versions of the program on Hydra with `OMP_NUM_THREADS=3`.

- **Version A:** `res=1`
- **Version B:** `res=5`
- **Version C:** `res=5`

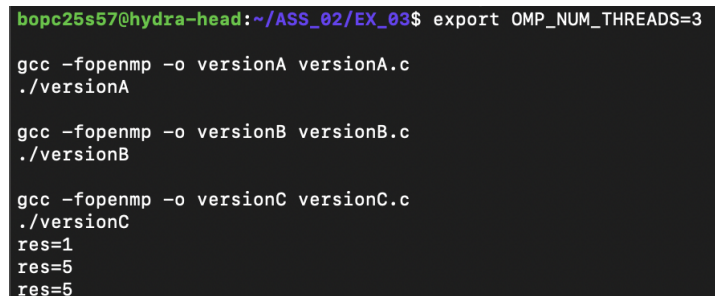
Explanation:

Version A runs `omp_tasks(5)` inside an `#pragma omp master` region, which allows only the master thread (typically thread 0) to execute the code. The tasks are not executed by any other threads, leading to incomplete recursion and returning only the base case value of 1.

Versions B and C allow all threads to participate in task execution:

- In Version B, the use of `#pragma omp single nowait` allows all other threads to pick up tasks immediately.
- In Version C, the `#pragma omp critical` does not restrict task execution since all threads remain active inside the parallel region.

Thus, Versions B and C correctly compute `omp_tasks(5) = Fibonacci(5) = 5`.



```
bopc25s57@hydra-head:~/ASS_02/EX_03$ export OMP_NUM_THREADS=3
gcc -fopenmp -o versionA versionA.c
./versionA
res=1

gcc -fopenmp -o versionB versionB.c
./versionB
res=5

gcc -fopenmp -o versionC versionC.c
./versionC
res=5
```

Figure 1: Experimental output of Exercise 3.1 tests on Hydra.

3.2. How often is the function `omp_tasks` called?

The number of calls to `omp_tasks()` depends on the version of the program and how OpenMP tasks are handled:

- **Version A:** The function is called only once. Due to the use of `#pragma omp master`, only the master thread participates. Tasks are not executed by other threads, so no recursive expansion occurs, and the result is 1.

- **Version B:** The function is called 11 times. The use of `#pragma omp single nowait` allows all threads to execute tasks. The recursion proceeds as expected, corresponding to the Fibonacci(5) call tree.
- **Version C:** The function is called 11 times. Similar to Version B, threads remain active inside the parallel region and execute all spawned tasks, fully expanding the recursive call tree.

In Versions B and C, the call structure corresponds to the Fibonacci expansion:

$$\text{omp_tasks}(5) = \text{omp_tasks}(4) + \text{omp_tasks}(3) = \dots$$

which results in a total of 11 function calls.

4. Exercise 4

4.1. Parallelize the pixel computation

We parallelized the function `compute_julia_set` using OpenMP to speed up the computation of the Julia set fractal. The main computation is done over a 2D image grid, and originally used two nested for loops to compute each pixel value.

Below is the parallelized version of the code:

Listing 1: Parallelized Julia Set Function

```
void compute_julia_set(double xmin, double xmax, double ymin, double ymax,
    double **image, int im_width, int im_height)
{
    double zabs_max = 10;
    double complex c = -0.1 + 0.65 * I;
    int nit_max = 1000;

    double xwidth = xmax - xmin;
    double yheight = ymax - ymin;

    // parallelize the two loops with openmp
    #pragma omp parallel for collapse(2) schedule(runtime)
    for(int i = 0; i < im_width; i++) {
        for(int j = 0; j < im_height; j++) {
            int nit = 0;
            double complex z;

            // map pixel (i,j) to complex plane
            z = (double)i / (double)im_width * xwidth + xmin
                + ((double)j / (double)im_height * yheight + ymin) * I;

            // iterate z = z^2 + c
            while( cabs(z) <= zabs_max && nit < nit_max ) {
                z = cpow(z, 2) + c;
                nit += 1;
            }

            // store normalized iteration count
            image[i][j] = (double)nit / (double)nit_max;
        }
    }
}
```

The OpenMP directive used here is:

```
#pragma omp parallel for collapse(2) schedule(runtime)
```

- **parallel for:** Tells the compiler to run the outer loop in parallel using multiple threads.
- **collapse(2):** Combines the two nested loops into a single iteration space. This improves load balancing by distributing all pixel computations more evenly across threads.
- **schedule(runtime):** Allows the scheduling strategy (e.g., static, dynamic, guided) to be set at runtime using the OMP_SCHEDULE environment variable. This makes it easier to compare different strategies during performance tests.

4.2. Running time analysis

We measured the runtime of the parallelized Julia set computation for two problem sizes: $n = 85$ and $n = 1150$, using different thread counts $p \in \{1, 2, 4, 8, 16, 24, 32\}$. Each experiment was repeated three times, and we recorded the minimum runtime for each case. No specific scheduling strategy was set (OMP_SCHEDULE was left unset), which means the default OpenMP schedule was used.

Table 6: Julia set strong scaling results for $n=85$ and $n=1150$

n	p	Run 1	Run 2	Run 3
85	1	0.101941	0.101826	0.101940
85	2	0.051824	0.051176	0.051211
85	4	0.025870	0.025814	0.025834
85	8	0.013258	0.013244	0.013235
85	16	0.007074	0.007080	0.007148
85	24	0.007368	0.007284	0.007434
85	32	0.007713	0.007611	0.007595
1150	1	17.984472	17.878764	17.835612
1150	2	9.008541	9.006459	9.011777
1150	4	4.519853	4.522764	4.518850
1150	8	2.272444	2.271688	2.272262
1150	16	1.155667	1.155936	1.155455
1150	24	1.155251	1.159520	1.155940
1150	32	1.157411	1.154194	1.155011

For the smaller input size ($n = 85$), we observe that the runtime decreases when increasing the thread count up to $p = 4$. After that, performance improvements become negligible or inconsistent. This behavior is expected because the problem is too small to fully utilize many cores. The overhead of thread management begins to dominate.

For the larger input size ($n = 1150$), we see a clear and consistent decrease in runtime as the number of threads increases. This shows effective strong scaling: the larger problem size provides enough work to keep all threads busy, resulting in better parallel performance.

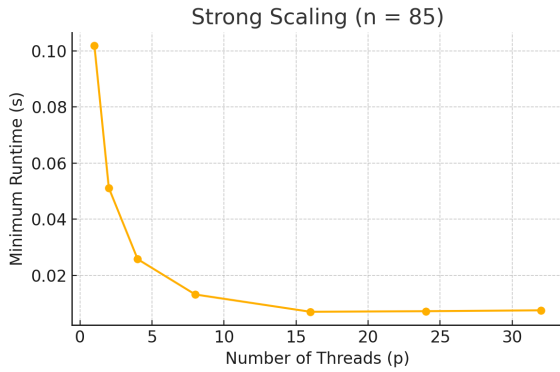
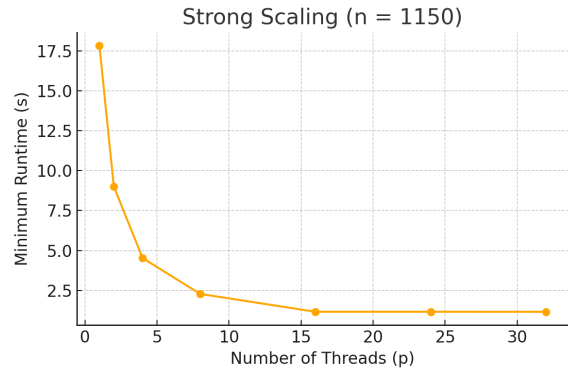
Figure 2: (a) $n = 85$ Figure 3: (b) $n = 1150$

Figure 4: Minimum runtime vs. number of threads for two problem sizes.

In both cases, we observe a point of diminishing returns. For $n = 1150$, using 32 threads still improves performance, but the gains are smaller compared to earlier increases (e.g., from 2 to 4 or 4 to 8 threads). This is likely due to memory bandwidth limitations and thread management overhead.

The experiment confirms that strong scaling is effective for large inputs. For small inputs, parallelization provides little benefit beyond a few threads. These results highlight the importance of matching problem size to parallelism granularity.

4.3. Influence of schedule parameter

To evaluate how the OpenMP scheduling strategy affects performance, we fixed the input size to $n = 1150$ and used $p = 16$ threads. We tested the following OMP_SCHEDULE settings:

- `static`
- `static,1`
- `dynamic,5`
- `guided,10`

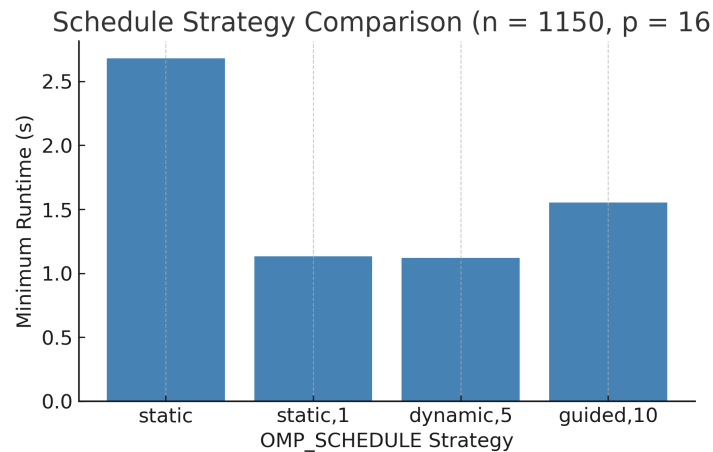
Each strategy was executed 3 times. The plot below shows the **minimum runtime** observed for each case.

The results reveal clear performance differences between the scheduling strategies:

- `static` performed the worst. Since it evenly splits the iteration space at compile time, it can cause threads to be idle if some chunks are heavier than others. This leads to load imbalance and wasted parallel potential.

Table 7: Julia Set schedule comparison results ($n=1150$, $p=16$)

Schedule	Run 1	Run 2	Run 3
static	1.158064	1.154984	1.155272
static 1	1.157391	1.157702	1.154595
dynamic 5	1.165659	1.165530	1.166533
guided 10	1.156678	1.158215	1.159699

Figure 5: Minimum runtime for each scheduling strategy ($n = 1150$, $p = 16$).

- `static,1` performed significantly better. With a chunk size of 1, each thread receives a small amount of work at a time. This improves load balancing and keeps all threads busy.
- `dynamic,5` achieved the best performance. Dynamic scheduling assigns chunks on-the-fly as threads finish their work, allowing very efficient balancing of irregular workloads.
- `guided,10` also improved over static, but not as much as dynamic. Guided scheduling starts with large chunks and then decreases chunk size. This works well in some cases, but can still leave some threads idle early on if the work per chunk is uneven.

For this computation, small chunk sizes and dynamic work assignment clearly lead to better thread utilization and faster runtimes.

5. Exercise 5

5.1. Parallelize the filter computation

To parallelize the filtering process, we modified the `apply_filter()` function in `filter.c` using OpenMP task-based parallelism.

Instead of parallelizing the nested for loops with `#pragma omp for`, we opted for `#pragma omp task` to parallelize the application of the filter over independent rows. Each iteration of the outer pixel row loop (`i`) is wrapped as an OpenMP task, which allows the runtime to dynamically schedule the row-wise work across threads.

Below is the updated function:

```
void apply_filter(png_bytep *row_pointers, png_bytep *buf, int width, int height,
                 double filter[3][3], int rounds) {
    for (int r = 0; r < rounds; r++) {
        #pragma omp parallel
        {
            #pragma omp single
            {
                for (int i = fwidth2; i < width - fwidth2; i++) {
                    #pragma omp task firstprivate(i)
                    {
                        for (int j = fheight2; j < height - fheight2; j++) {
                            filter_on_pixel(row_pointers, buf, i, j, filter);
                        }
                    }
                }
            }
        }

        // Swap buffers
        png_bytep *tmp = buf;
        buf = row_pointers;
        row_pointers = tmp;
    }
}
```

This task-based implementation provides fine grained parallelism and is effective for dynamic scheduling of pixel row computations.

5.2. Strong scaling analysis

To evaluate the strong scaling of our parallel image filter, we fixed the number of filtering rounds to $r = 1$ and varied the number of threads $p \in \{1, 2, 4, 8, 16, 24, 32\}$. For each configuration, the experiment was repeated three times and the minimum runtime was recorded. Figure 6 shows a clear decrease in execution time as the number of threads increases, demonstrating effective strong scaling up to 16 threads. Beyond that point, the performance gain flattens out, likely due to increased task scheduling overhead, memory bandwidth contention, and diminishing parallel workload per thread.

Table 8: Strong Scaling Results

Threads	Run1	Run2	Run3	Min Time	Avg Time
1	0.07654	0.06999	0.07501	0.06999	0.07385
2	0.04679	0.04587	0.05388	0.04587	0.04884
4	0.03305	0.03477	0.03408	0.03305	0.03397
8	0.02367	0.02291	0.02245	0.02245	0.02301
16	0.01529	0.01522	0.01539	0.01522	0.01530
24	0.01575	0.01552	0.01609	0.01552	0.01579
32	0.01563	0.01553	0.01651	0.01553	0.01589

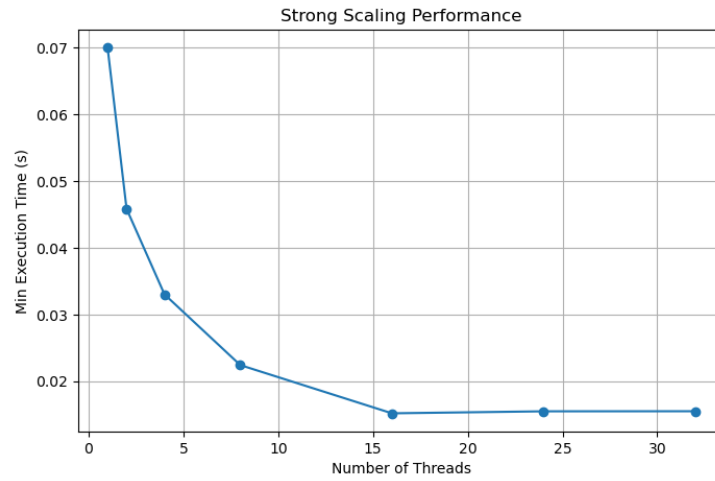


Figure 6: Strong scaling performance: minimum runtime vs. number of threads.

The largest gains are observed between 1 and 4 threads, indicating that the task-based approach effectively utilizes additional cores up to a moderate level of concurrency. However, at high thread counts, overhead dominates and the speedup saturates.

5.3. Weak scaling analysis

To evaluate the weak scaling behavior, we proportionally increased the number of filtering rounds with the number of threads, i.e., $p = r$ for $p \in \{1, 2, 4, 8, 16, 24, 32\}$. Each configuration was executed three times, and the minimum runtime was recorded.

Table 9: Weak Scaling Results

Threads	Rounds	Run1	Run2	Run3	Min Time	Avg Time
1	1	0.067904	0.068320	0.068791	0.067904	0.068338
2	2	0.083288	0.082935	0.082866	0.082866	0.083030
4	4	0.090475	0.089510	0.092228	0.089510	0.090738
8	8	0.097943	0.098335	0.098019	0.097943	0.098099
16	16	0.12846	0.13822	0.11255	0.11255	0.126415
24	24	0.16561	0.16719	0.16685	0.16561	0.166558
32	32	0.22178	0.21764	0.2207	0.21764	0.220065

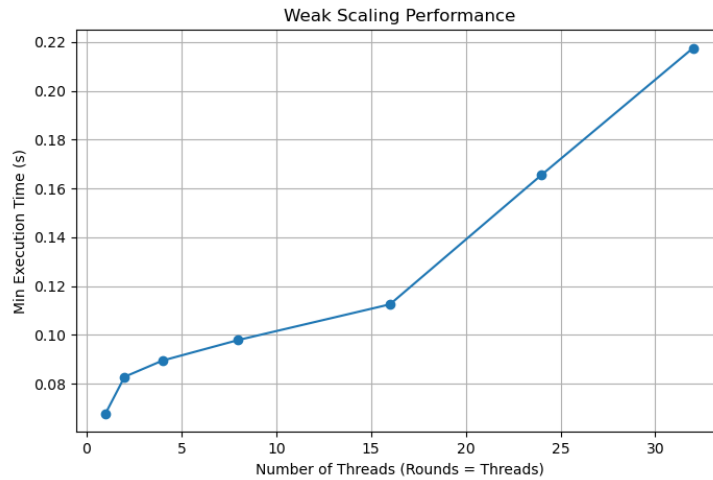


Figure 7: Weak scaling performance: minimum runtime vs. number of threads (rounds = threads).

Figure 7 shows how runtime increases with the number of threads when the workload is scaled proportionally. Ideally, weak scaling would result in a constant runtime. However, we observe a gradual increase, especially beyond 8 threads.

This behavior reflects common performance limitations due to:

- Increasing scheduling and synchronization overhead.

- Memory bandwidth contention at high thread counts.
- Sublinear parallel efficiency with large workloads.

The implementation still scales well up to 8–16 threads, after which overheads dominate.

A. Task 4 - Raw Data**A.1. Scaling Results**

n	p	run	time
85	1	1	0.101941
85	1	2	0.101826
85	1	3	0.101940
85	2	1	0.051824
85	2	2	0.051176
85	2	3	0.051211
85	4	1	0.025870
85	4	2	0.025814
85	4	3	0.025834
85	8	1	0.013258
85	8	2	0.013244
85	8	3	0.013235
85	16	1	0.007074
85	16	2	0.007080
85	16	3	0.007148
85	24	1	0.007368
85	24	2	0.007284
85	24	3	0.007434
85	32	1	0.007713
85	32	2	0.007611
85	32	3	0.007595
1150	1	1	17.984472
1150	1	2	17.878764
1150	1	3	17.835612
1150	2	1	9.008541
1150	2	2	9.006459
1150	2	3	9.011777
1150	4	1	4.519853
1150	4	2	4.522764
1150	4	3	4.518850
1150	8	1	2.272444
1150	8	2	2.271688
1150	8	3	2.272262
1150	16	1	1.155667
1150	16	2	1.155936
1150	16	3	1.155455
1150	24	1	1.155251
1150	24	2	1.159520
1150	24	3	1.155940
1150	32	1	1.157411
1150	32	2	1.154194
1150	32	3	1.155011

A.2. Scheduling Results

schedule	run	time
static	1	2.683538
static	2	2.683155
static	3	2.680293
static_1	1	1.132370
static_1	2	1.132771
static_1	3	1.134270
dynamic_5	1	1.121596
dynamic_5	2	1.121727
dynamic_5	3	1.121801
guided_10	1	1.553587
guided_10	2	1.554751
guided_10	3	1.554237

B. Task 5 - Raw Data**B.1. Strong Scaling Results**

Threads	Run1	Run2	Run3	Min Time	Avg Time
1	0.0765401	0.0699914	0.0750116	0.0699914	0.0738477
2	0.0467858	0.045865	0.053879	0.045865	0.04884326666666667
4	0.0330477	0.0347701	0.0340819	0.0330477	0.03396656666666666
8	0.0236741	0.0229067	0.0224481	0.0224481	0.02300963333333333
16	0.0152943	0.0152223	0.0153936	0.0152223	0.0153034
24	0.0157466	0.0155221	0.0160895	0.0155221	0.015786066666666668
32	0.0156315	0.0155334	0.0165146	0.0155334	0.015893166666666667

B.2. Weak Scaling Results

Threads	Rounds	Run1	Run2	Run3	Min Time	Avg Time
1	1	0.0679049	0.0683204	0.0687911	0.0679049	0.0683388
2	2	0.0832885	0.0829358	0.0828668	0.0828668	0.08303036666666667
4	4	0.0904756	0.0895107	0.0922283	0.0895107	0.0907382
8	8	0.0979439	0.0983353	0.0980195	0.0979439	0.09809956666666667
16	16	0.128465	0.138223	0.112559	0.112559	0.12641566666666668
24	24	0.165618	0.167199	0.166858	0.165618	0.16655833333333334
32	32	0.221783	0.217643	0.22077	0.217643	0.22006533333333334