

Basics of Parallel Computing  
2024S  
Assignment 2

May 28, 2024

## 2 Person Group 13

1: Pia SCHWARZINGER, 12017370  
2: Yahya JABARY, 11912007

## 1 Exercise 1

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char *argv[]) {
    int nenv = 3;
    omp_set_num_threads(nenv); // set number of threads
    printf("nenv: %d\n", nenv);

    int chunk = 5;
    omp_set_schedule(omp_sched_static, chunk);
    // omp_set_schedule(omp_sched_dynamic, chunk);
    // omp_set_schedule(omp_sched_guided, chunk);
    printf("chunk size: %d\n", chunk);

    int i = 0;
    int n = 17;
    int a[n];
    int t[nenv];

    #pragma omp parallel for schedule(runtime)
    for (i=0; i<n; i++) {
        a[i] = omp_get_thread_num(); // chosen thread per iteration
        t[omp_get_thread_num()]++; // parallel increment
    }

    printf("a (schedule): ");
    for (i=0; i<n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");

    printf("t (counter): ");
    for (i=0; i<nenv; i++) {
        printf("%d ", t[i]);
    }
    printf("\n");
}
```

## 1.1 What do a and t count?

The variable a stores the selected thread number for each parallel iteration, while t stores a non-atomic counter that all threads with the same ID increment. Unless no two threads are assigned the same iteration, the final value of t will be non-deterministic as each var++ operation is in fact a read-modify-write operation:

```
movl  -4(%rbp), %eax    # load var into eax
addl  $1, %eax          # increment eax by 1
movl  %eax, -4(%rbp)    # store eax back into var
```

## 1.2 Values for all elements in a and t

See Tables 1 and 2 for the values of a and t for different scheduling strategies.

Table 1: Values of array *a* for different scheduling strategies

| case / <i>a</i> | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-----------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
| static, 0       | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1  | 1  | 2  | 2  | 2  | 2  | 2  |
| static, 1       | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1  | 2  | 0  | 1  | 2  | 0  | 1  |
| dynamic, 1      | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
| dynamic, 2      | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| guided, 5       | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

Table 2: Values of array *t* for different scheduling strategies - keep in mind that these values are not reproducible / deterministic.

| case / <i>t</i> | 0        | 1  | 2          |
|-----------------|----------|----|------------|
| static, 0       | 74307862 | 7  | 1806905557 |
| static, 1       | 8591638  | 7  | 1872621781 |
| dynamic, 1      | 6150416  | 18 | 1875062992 |
| dynamic, 2      | 40737057 | 1  | 1840476368 |
| guided, 5       | 51370273 | 1  | 1829843168 |

## 2 Exercise 2

Table 3: Duration of independent tasks we want to schedule optimally.

| Task ID       | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
| Task duration | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 1  | 2  | 1  | 2  | 4  | 3  | 3  |

### 2.1 Optimal Schedule

To minimize the total execution time with 4 workers, we can use the “Longest Processing Time” (LPT) algorithm by R. L. Graham in 1969. Here’s how it works: First, sort the tasks by duration in descending order. Then, assign the tasks to the least loaded worker. But beware that the LPT isn’t guaranteed to find the optimal solution, but just to have a provable upper bound of  $\lceil 4/3 \cdot \text{OPT} \rceil$  where OPT is the optimal solution.

Assuming that tasks can be interrupted and resumed at any time, we can calculate the OPT as follows:  $\text{OPT} = \lceil \sum_{i=0}^{16} \text{task duration}_i / 4 \rceil = \lceil 31/4 \rceil = 8$ .

Fortunately we were able to find one of the optimal solutions by using the LPT algorithm.

```
from itertools import groupby
from operator import itemgetter

def schedule_tasks(task_durations):
    # fst: task_id, snd: duration
    sorted_tasks = sorted(enumerate(task_durations), key=lambda x: x[1], reverse=True)

    worker_utilization = [0] * 4 # time spent on work by each worker so far

    scheduled_tasks = []
    for task_id, duration in sorted_tasks:
        # get least utilized worker
        min_time = min(worker_utilization)
        worker_index = worker_utilization.index(min_time)

        # assign task
        worker_utilization[worker_index] += duration

        # keep track of assigned task
        start_time = min_time
        end_time = start_time + duration
        scheduled_tasks.append((worker_index, task_id, start_time, end_time))

    return scheduled_tasks

task_durations = [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 4, 3, 3]
print(f"sorted tasks: {sorted(task_durations, reverse=True)}\n")
scheduled_tasks = schedule_tasks(task_durations)
```

```
# group tasks by worker
scheduled_tasks.sort(key=itemgetter(0))
for worker, tasks in groupby(scheduled_tasks, key=itemgetter(0)):
    print(f"worker {worker}:")
    for task in tasks:
        print(f"\t\ttask {task[1]}, start: {task[2]}, end: {task[3]} (duration: {task[3] - task[2]})")
    print()

# effective time
print(f"time spent: {max(map(itemgetter(3), scheduled_tasks))}")
```

sorted tasks: [4, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1]

worker 0:

- task 14, start: 0, end: 4 (duration: 4)
- task 9, start: 4, end: 6 (duration: 2)
- task 2, start: 6, end: 7 (duration: 1)
- task 8, start: 7, end: 8 (duration: 1)

worker 1:

- task 15, start: 0, end: 3 (duration: 3)
- task 5, start: 3, end: 5 (duration: 2)
- task 13, start: 5, end: 7 (duration: 2)
- task 10, start: 7, end: 8 (duration: 1)

worker 2:

- task 16, start: 0, end: 3 (duration: 3)
- task 7, start: 3, end: 5 (duration: 2)
- task 0, start: 5, end: 6 (duration: 1)
- task 4, start: 6, end: 7 (duration: 1)
- task 12, start: 7, end: 8 (duration: 1)

worker 3:

- task 1, start: 0, end: 2 (duration: 2)
- task 3, start: 2, end: 4 (duration: 2)
- task 11, start: 4, end: 6 (duration: 2)
- task 6, start: 6, end: 7 (duration: 1)

time spent: 8

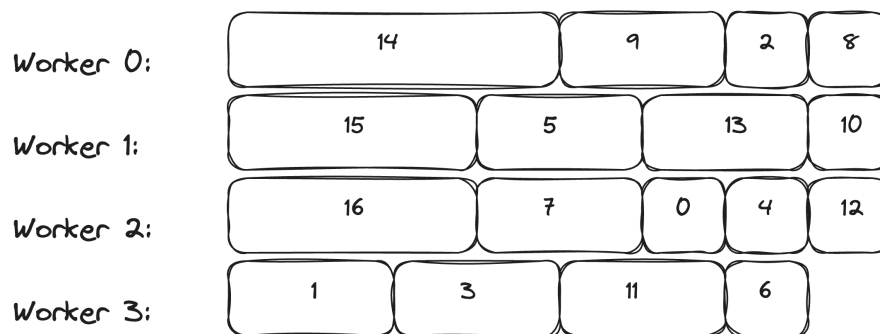


Figure 1: Gantt chart of the LPT schedule (which happens to be optimal).

## 2.2 Schedule static,3

The schedule static,3 assigns each task to a worker in a round-robin fashion with a chunk size of 3.

The makespan of the schedule static,3 is 11, which is suboptimal compared to the LPT schedule. The Gantt chart in Figure 3 shows the schedule.

## 2.3 Schedule dynamic,2

The schedule dynamic,2 assigns chunks of 2 tasks to a random worker that is currently idle.

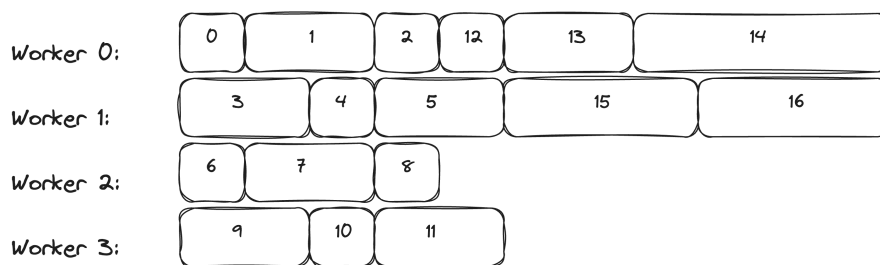


Figure 2: Gantt chart of the schedule static,3.

The makespan of the schedule dynamic,2 is 10, which is suboptimal compared to the LPT schedule but better than the static,3 schedule. The Gantt chart in Figure 3 shows the schedule.

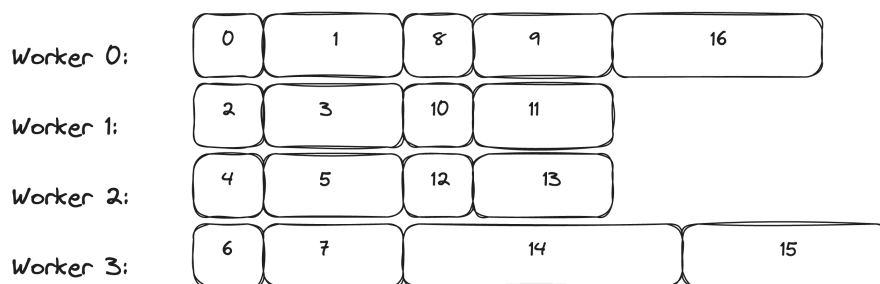


Figure 3: Gantt chart of the schedule dynamic,2.

### 3 Exercise 3

#### 3.1 Fix the problems with this OpenMP code

Here are some suggestions to fix the problems with the given code snippet:

- The instruction `count_odd += my_count_odd;` doesn't make a lot of sense given that only the `#pragma` region inside the `for` loop is parallelized. Replace the `my_count_odd` variable with the shared variable `count_odd` and use an atomic operation to increment it.
- Or even better: Use a reduction clause to sum up the number of odd numbers in the array instead of declaring a shared variable and incrementing it atomically yourself.
- Turn the function `static` so it's just visible in a single translation / compilation unit (optional).

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

static int omp_odd_counter(int *a, int n) {
    int count_odd = 0;

    #pragma omp parallel for reduction(+:count_odd)
    for (int i = 0; i < n; i++) {
        if (a[i] % 2 == 1) {
            count_odd++;
        }
    }

    return count_odd;
}

int main() {
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = sizeof(a) / sizeof(a[0]);
```

```

    int out = omp_odd_counter(a, n);

    printf("expected: 5\n");
    printf("got: %d\n", out);
    return EXIT_SUCCESS;
}

```

## 4 Exercise 4

### 4.1 What is the output of the three different versions?

Here are the results of the 3 given versions of the code snippet when compiled with `gcc -fopenmp` and executed with `omp_set_num_threads(4)`:

```
$ gcc -fopenmp -o version_a version_a.c && ./version_a
res=1
```

```
$ gcc -fopenmp -o version_b version_b.c && ./version_b
res=5
```

```
$ gcc -fopenmp -o version_c version_c.c && ./version_c
res=5
```

The results differ because of the `shared` and `private` clauses in the `omp_task` function. The `shared` clause is used to share the result of the sub-task with the parent task, while the `private` clause is used to create a local copy of the result for each thread.

Here's some ASCII art to illustrate the stack-trace in the case of one sub-task result being shared and the other being private:

```

v=5          -> final result: 1
├─ v=4        -> shared: 1
│ └─ v=3      -> shared: 1
│   └─ v=2    -> shared: 1
│     └─ v=1   -> private
└─ v=2        -> private
    └─ v=3     -> private
        └─ v=2 -> shared: 1
          └─ v=1 -> private

```

And here's the stack-trace for the case where both sub-task results are shared:

```

v=5          -> final result: 5
├─ v=4        -> shared: 3
│ └─ v=3      -> shared: 2
│   └─ v=2    -> shared: 1
│     └─ v=1   -> shared: 1
└─ v=2        -> shared: 1
    └─ v=3     -> shared: 2
        └─ v=2 -> shared: 1
          └─ v=1 -> shared: 1

```

### 4.2 How often is the function `omp_tasks` called?

Let's add a print statement to the `omp_tasks` function to see how often it is called / what values are passed to it.

This is the most straightforward way to print the stack-trace for our use case. I also tried to write a custom 11db python script and a bunch of C-libraries to achieve the same result, but they all failed when combined with the OpenMP runtime.

Here are the modified versions of the code snippets:

```

/*
 * Version A
 */

#include <stdio.h>
#include <omp.h>

static int omp_tasks(int v) {
    printf("called with v=%d\n", v);

    int a = 0, b = 0;
    if (v <= 2) {
        return 1;
    } else {
        #pragma omp task shared(a) // shared
        a = omp_tasks(v - 1);
        #pragma omp task private(b) // private

```

```

        b = omp_tasks(v - 2);
        #pragma omp taskwait
        return a + b;
    }
}

void main() {
    omp_set_num_threads(4);

    int res;
    #pragma omp parallel
    {
        #pragma omp master // master
        res = omp_tasks(5);
    }
    printf("res=%d\n", res);
}

```

```

/*
 * Version B
 */

#include <stdio.h>
#include <omp.h>

static int omp_tasks(int v) {
    printf("called with v=%d\n", v);

    int a = 0, b = 0;
    if (v <= 2) {
        return 1;
    } else {
        #pragma omp task shared(a) // shared
        a = omp_tasks(v - 1);
        #pragma omp task shared(b) // shared
        b = omp_tasks(v - 2);
        #pragma omp taskwait
        return a + b;
    }
}

void main() {
    omp_set_num_threads(4);

    int res;
    #pragma omp parallel
    {
        #pragma omp single nowait // single nowait
        res = omp_tasks(5);
    }
    printf("res=%d\n", res);
}

```

```

/*
 * Version C
 */

#include <stdio.h>
#include <omp.h>

static int omp_tasks(int v) {
    printf("called with v=%d\n", v);

    int a = 0, b = 0;
    if (v <= 2) {
        return 1;
    } else {
        #pragma omp task shared(a) // shared
        a = omp_tasks(v - 1);
        #pragma omp task shared(b) // shared
        b = omp_tasks(v - 2);
    }
}

```

```

        #pragma omp taskwait
        return a + b;
    }
}

void main() {
    omp_set_num_threads(4);

    int res;
    #pragma omp parallel
    {
        #pragma omp critical // critical
        res = omp_tasks(5);
    }
    printf("res=%d\n", res);
}

```

And here's the output of the modified scripts:

```

$ gcc -fopenmp -o version_a version_a.c && ./version_a
called with v=5
called with v=3
called with v=4
called with v=1
called with v=3
called with v=2
called with v=2
called with v=1
called with v=2
res=1

```

```

$ gcc -fopenmp -o version_a version_a.c && ./version_a | grep -c "called with v" | xargs echo "num calls:"
num calls: 9

```

```

$ gcc -fopenmp -o version_b version_b.c && ./version_b
called with v=5
called with v=3
called with v=1
called with v=4
called with v=2
called with v=3
called with v=1
called with v=2
called with v=2
res=5

```

```

$ gcc -fopenmp -o version_b version_b.c && ./version_b | grep -c "called with v" | xargs echo "num calls:"
num calls: 9

```

```

$ gcc -fopenmp -o version_c version_c.c && ./version_c
called with v=5
called with v=3
called with v=1
called with v=2
called with v=4
called with v=2
called with v=3
called with v=1
called with v=2
called with v=5
called with v=1
called with v=2
called with v=4
called with v=2
called with v=3
called with v=1
called with v=2
called with v=5
called with v=3
called with v=1
called with v=2
called with v=4

```

```
called with v=3
called with v=1
called with v=2
called with v=2
called with v=5
called with v=3
called with v=1
called with v=2
called with v=4
called with v=2
called with v=3
called with v=1
called with v=2
res=5
```

```
$ gcc -fopenmp -o version_c version_c.c && ./version_c | grep -c "called with v" | xargs echo "num calls:"
num calls: 36
```

We've already discussed the effect the inner `shared` and `private` clauses have on the final result. The number of calls to the `omp_tasks` function however depends on the the pragma used in the `main` function:

- Version A / `master`: This pragma causes the recursive function to be called from the `main` function just once. The `master` pragma ensures that only the master thread executes the critical section.
- Version B / `single nowait`: The `single` pragma ensures that only one thread executes the critical section, while the `nowait` clause allows the thread to continue executing the next section of code without waiting for the other threads to finish. The `nowait` clause in this case doesn't have any effect as it is the last section of code in the `parallel` region of the `main` function.
- Version C / `critical`: This pragma caused the recursive function to be executed 36 (4 threads · 9 recursive calls) times. The `critical` clause ensures that only one thread can execute the critical section at a time but doesn't have any guarantees on how often the critical section is executed.

## 5 Exercise 5

### 5.1 Parallelize the pixel computation

The `for` is used to parallelize the loops and needs to be called inside parallel blocks to be effective which explains the `#pragma omp parallel for`. In order to guarantee a canonical form `i` and `j` are set to private variables. Setting `nit` and `z` to `private` ensures that each thread has its own instance of these variables so that there are no errors in the computation. Since the remaining variables are constants, there is no need for synchronization. Lastly, a `collapse(2)` allows the nested loops to be treated as a single loop, which can improve parallel efficiency. This is only possible because they are perfectly nested loops.

### 5.2 Running time analysis

The plot 4 shows that using more cores reduces running time significantly up to 16 cores. From 1 to 4 cores, the running time drops sharply, then continues to improve up to 16 cores. Beyond 16 cores, the benefits decrease, and the running time stays around 0.02 seconds. Adding more than 24 cores slightly increases the running time, likely due to overhead. Overall, the program performs best with up to 16 cores, with minimal gains and some overhead issues beyond that.



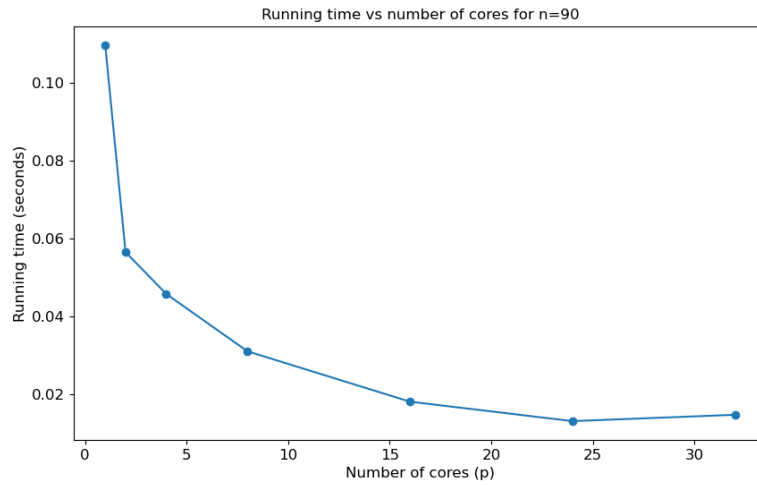


Figure 4: Minimum running time vs number of cores for an input size of 90

When setting the input size to 1100, figure 5 reveals that the running time decreases significantly up to 16 cores, similar to the  $n=90$  case. Beyond 16 cores, the running time continues to decrease but at a much slower rate, and eventually stabilizes. Unlike the  $n=90$  case, there is no noticeable increase in running time when using more than 24 cores, suggesting that the overhead is better managed or less impactful for larger workloads. The larger problem size of  $n=1100$  benefits more from parallelization, as distributing the increased workload across multiple cores leads to more substantial performance gains.

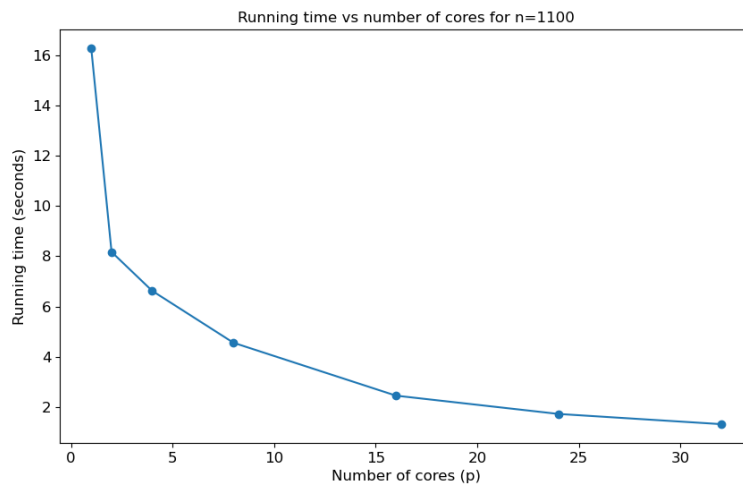


Figure 5: Minimum running time vs number of cores for an input size of 1100

### 5.3 Influence of schedule parameter

The graph 6 demonstrates minimal differences in running times for different scheduling options with  $n=1100$  and 16 cores. This suggests that the workload is well-balanced across the cores regardless of the schedule used which can also be confirmed by the previously discussed figure 4 and figure 5. `Guided,8` performs slightly better than `dynamic,10`, likely due to better adjustment of chunk sizes. Both `static` and `static,1` schedules show similar performance, with `static,1` marginally better. Overall, the uniform nature of the workload leads to evenly distributed loads, making the scheduling overhead negligible.

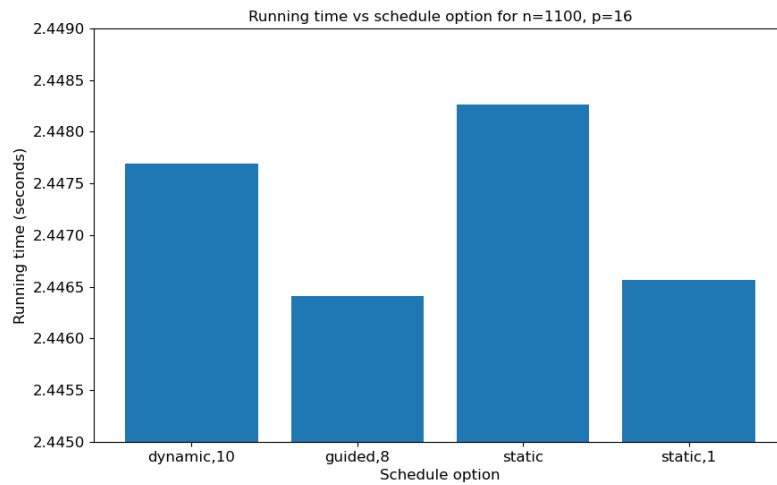


Figure 6: Minimum running time vs schedule option for an input size of 1100 and 16 cores

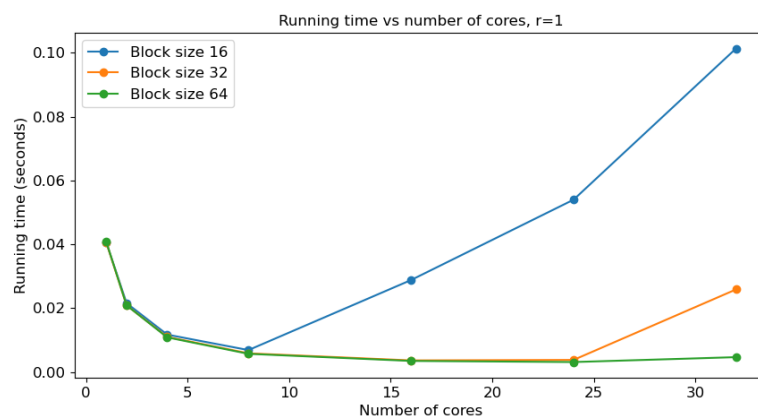
## 6 Exercise 6

### 6.1 Parallelize the filter computation

Again, the thread team is created with `#pragma omp parallel` and the tasks are created inside the parallel section. Without `single` each thread of the team would call `filter_on_pixel`. The private variables are initialized with the value of the outside variable, by defining them as `firstprivate` because otherwise all tasks created within the loop would reference the same `i` and `j` variables. The task-based parallelism would create a task for each pixel, which is why the `block_size` was adapted and experimented with.

### 6.2 Strong scaling analysis

Figure 7 shows that for a fixed problem size and varying the number of cores, the performance shows a clear trend where smaller block sizes (like 16) result in significantly higher running times as the number of cores increases. This sharp increase, especially noticeable at 32 cores, suggests that block size 16 is not well-suited for higher core counts. Larger block sizes (32 and 64) maintain relatively low running times with a gradual increase, indicating better parallel efficiency, likely due to reduced overhead and better cache utilization. The optimal core range for block sizes 32 and 64 appears to be 16 to 24, where the running time is minimized.

Figure 7: Minimum running time vs number of cores for  $r=1$  and various block sizes

### 6.3 Weak scaling analysis

In the weak scaling scenario, where both the number of cores and problem size increase proportionally, the running time should ideally remain constant if the system scales perfectly. However, as figure 8 shows, the running time increases with a block size of 16 as there is a sharp increase indicating poor scalability and higher overheads as the workload grows. On the other hand, block sizes 32 and 64 exhibit much flatter curves, indicating better workload distribution. As a result, they provide a more stable and predictable performance across different core counts.

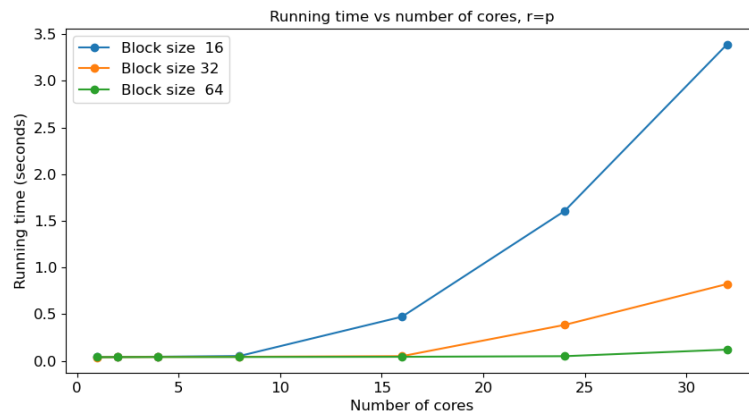


Figure 8: Minimum running time vs number of cores for  $r=p$  and various block sizes

**7 Appendix**

| n    | p  | time      |
|------|----|-----------|
| 90   | 1  | 0.110155  |
| 90   | 1  | 0.109749  |
| 90   | 1  | 0.109885  |
| 90   | 2  | 0.056617  |
| 90   | 2  | 0.056599  |
| 90   | 2  | 0.056612  |
| 90   | 4  | 0.04588   |
| 90   | 4  | 0.045966  |
| 90   | 4  | 0.045863  |
| 90   | 8  | 0.03112   |
| 90   | 8  | 0.031132  |
| 90   | 8  | 0.03117   |
| 90   | 16 | 0.018182  |
| 90   | 16 | 0.018227  |
| 90   | 16 | 0.01822   |
| 90   | 24 | 0.013238  |
| 90   | 24 | 0.013257  |
| 90   | 24 | 0.01318   |
| 90   | 32 | 0.014816  |
| 90   | 32 | 0.017296  |
| 90   | 32 | 0.014814  |
| 1100 | 1  | 16.306608 |
| 1100 | 1  | 16.316588 |
| 1100 | 1  | 16.284397 |
| 1100 | 2  | 8.175213  |
| 1100 | 2  | 8.178992  |
| 1100 | 2  | 8.170321  |
| 1100 | 4  | 6.621239  |
| 1100 | 4  | 6.678632  |
| 1100 | 4  | 6.639713  |
| 1100 | 8  | 4.557337  |
| 1100 | 8  | 4.554004  |
| 1100 | 8  | 4.58649   |
| 1100 | 16 | 2.447131  |
| 1100 | 16 | 2.448894  |
| 1100 | 16 | 2.4472    |
| 1100 | 24 | 1.731222  |
| 1100 | 24 | 1.718731  |
| 1100 | 24 | 1.718424  |
| 1100 | 32 | 1.312658  |
| 1100 | 32 | 1.313263  |
| 1100 | 32 | 1.320209  |

Table 4: julia strong scaling

| schedule   | n    | p  | time     |
|------------|------|----|----------|
| static     | 1100 | 16 | 2.450491 |
| static     | 1100 | 16 | 2.448260 |
| static     | 1100 | 16 | 2.449136 |
| static,1   | 1100 | 16 | 2.448672 |
| static,1   | 1100 | 16 | 2.446850 |
| static,1   | 1100 | 16 | 2.446568 |
| dynamic,10 | 1100 | 16 | 2.447841 |
| dynamic,10 | 1100 | 16 | 2.447689 |
| dynamic,10 | 1100 | 16 | 2.453576 |
| guided,8   | 1100 | 16 | 2.447433 |
| guided,8   | 1100 | 16 | 2.447667 |
| guided,8   | 1100 | 16 | 2.446407 |

Table 5: julia schedule

| n    | p  | r | time       |
|------|----|---|------------|
| 1168 | 1  | 1 | 0.0406543  |
| 1168 | 1  | 1 | 0.0406029  |
| 1168 | 1  | 1 | 0.0406245  |
| 1168 | 2  | 1 | 0.0215648  |
| 1168 | 2  | 1 | 0.0216172  |
| 1168 | 2  | 1 | 0.0215491  |
| 1168 | 4  | 1 | 0.0117266  |
| 1168 | 4  | 1 | 0.0117184  |
| 1168 | 4  | 1 | 0.0116864  |
| 1168 | 8  | 1 | 0.00683683 |
| 1168 | 8  | 1 | 0.00684586 |
| 1168 | 8  | 1 | 0.00686033 |
| 1168 | 16 | 1 | 0.0287889  |
| 1168 | 16 | 1 | 0.0288035  |
| 1168 | 16 | 1 | 0.0287749  |
| 1168 | 24 | 1 | 0.0764752  |
| 1168 | 24 | 1 | 0.0769229  |
| 1168 | 24 | 1 | 0.0539545  |
| 1168 | 32 | 1 | 0.115842   |
| 1168 | 32 | 1 | 0.101349   |
| 1168 | 32 | 1 | 0.107299   |

Table 6: filter strong scaling - blocksize = 16

| n    | p  | r | time       |
|------|----|---|------------|
| 1168 | 1  | 1 | 0.0407042  |
| 1168 | 1  | 1 | 0.040608   |
| 1168 | 1  | 1 | 0.0407692  |
| 1168 | 2  | 1 | 0.0208633  |
| 1168 | 2  | 1 | 0.0210453  |
| 1168 | 2  | 1 | 0.020857   |
| 1168 | 4  | 1 | 0.0110084  |
| 1168 | 4  | 1 | 0.0110562  |
| 1168 | 4  | 1 | 0.0109584  |
| 1168 | 8  | 1 | 0.0058259  |
| 1168 | 8  | 1 | 0.00581915 |
| 1168 | 8  | 1 | 0.005857   |
| 1168 | 16 | 1 | 0.00464876 |
| 1168 | 16 | 1 | 0.0035982  |
| 1168 | 16 | 1 | 0.00361837 |
| 1168 | 24 | 1 | 0.00375564 |
| 1168 | 24 | 1 | 0.0188177  |
| 1168 | 24 | 1 | 0.0156947  |
| 1168 | 32 | 1 | 0.0258458  |
| 1168 | 32 | 1 | 0.0266334  |
| 1168 | 32 | 1 | 0.0262409  |

Table 7: filter strong scaling - blocksize = 32

| n    | p  | r | time       |
|------|----|---|------------|
| 1168 | 1  | 1 | 0.0409708  |
| 1168 | 1  | 1 | 0.0408295  |
| 1168 | 1  | 1 | 0.0408123  |
| 1168 | 2  | 1 | 0.020855   |
| 1168 | 2  | 1 | 0.0209798  |
| 1168 | 2  | 1 | 0.0209425  |
| 1168 | 4  | 1 | 0.0108324  |
| 1168 | 4  | 1 | 0.0108178  |
| 1168 | 4  | 1 | 0.0108282  |
| 1168 | 8  | 1 | 0.00569162 |
| 1168 | 8  | 1 | 0.0056635  |
| 1168 | 8  | 1 | 0.00570401 |
| 1168 | 16 | 1 | 0.00343863 |
| 1168 | 16 | 1 | 0.0035551  |
| 1168 | 16 | 1 | 0.00346598 |
| 1168 | 24 | 1 | 0.00310194 |
| 1168 | 24 | 1 | 0.0030884  |
| 1168 | 24 | 1 | 0.00310673 |
| 1168 | 32 | 1 | 0.00461903 |
| 1168 | 32 | 1 | 0.00574673 |
| 1168 | 32 | 1 | 0.00664918 |

Table 8: filter strong scaling - blocksize = 64

| n    | p  | r  | time      |
|------|----|----|-----------|
| 1168 | 1  | 1  | 0.0406386 |
| 1168 | 1  | 1  | 0.0406179 |
| 1168 | 1  | 1  | 0.0406126 |
| 1168 | 2  | 2  | 0.0429599 |
| 1168 | 2  | 2  | 0.0431386 |
| 1168 | 2  | 2  | 0.0429649 |
| 1168 | 4  | 4  | 0.04605   |
| 1168 | 4  | 4  | 0.0456966 |
| 1168 | 4  | 4  | 0.0459851 |
| 1168 | 8  | 8  | 0.0682753 |
| 1168 | 8  | 8  | 0.0537896 |
| 1168 | 8  | 8  | 0.0532043 |
| 1168 | 16 | 16 | 0.473704  |
| 1168 | 16 | 16 | 0.503771  |
| 1168 | 16 | 16 | 0.515288  |
| 1168 | 24 | 24 | 1.60518   |
| 1168 | 24 | 24 | 1.67481   |
| 1168 | 24 | 24 | 1.69138   |
| 1168 | 32 | 32 | 3.39004   |
| 1168 | 32 | 32 | 3.48726   |
| 1168 | 32 | 32 | 3.60491   |

Table 9: filter weak scaling - blocksize = 16

| n    | p  | r  | time      |
|------|----|----|-----------|
| 1168 | 1  | 1  | 0.0406117 |
| 1168 | 1  | 1  | 0.0405664 |
| 1168 | 1  | 1  | 0.0405647 |
| 1168 | 2  | 2  | 0.0415773 |
| 1168 | 2  | 2  | 0.0418443 |
| 1168 | 2  | 2  | 0.0415992 |
| 1168 | 4  | 4  | 0.0436769 |
| 1168 | 4  | 4  | 0.0434178 |
| 1168 | 4  | 4  | 0.0438448 |
| 1168 | 8  | 8  | 0.045633  |
| 1168 | 8  | 8  | 0.0448665 |
| 1168 | 8  | 8  | 0.0452338 |
| 1168 | 16 | 16 | 0.0519127 |
| 1168 | 16 | 16 | 0.0517237 |
| 1168 | 16 | 16 | 0.0517178 |
| 1168 | 24 | 24 | 0.401326  |
| 1168 | 24 | 24 | 0.386013  |
| 1168 | 24 | 24 | 0.3997    |
| 1168 | 32 | 32 | 0.83737   |
| 1168 | 32 | 32 | 0.833949  |
| 1168 | 32 | 32 | 0.823811  |

Table 10: filter weak scaling - blocksize = 32

| n    | p  | r  | time      |
|------|----|----|-----------|
| 1168 | 1  | 1  | 0.0409068 |
| 1168 | 1  | 1  | 0.0408104 |
| 1168 | 1  | 1  | 0.0408199 |
| 1168 | 2  | 2  | 0.04175   |
| 1168 | 2  | 2  | 0.0416734 |
| 1168 | 2  | 2  | 0.0418532 |
| 1168 | 4  | 4  | 0.0432387 |
| 1168 | 4  | 4  | 0.0423335 |
| 1168 | 4  | 4  | 0.0429878 |
| 1168 | 8  | 8  | 0.0431827 |
| 1168 | 8  | 8  | 0.0437831 |
| 1168 | 8  | 8  | 0.0427322 |
| 1168 | 16 | 16 | 0.0462124 |
| 1168 | 16 | 16 | 0.0447389 |
| 1168 | 16 | 16 | 0.0450728 |
| 1168 | 24 | 24 | 0.0512439 |
| 1168 | 24 | 24 | 0.0512753 |
| 1168 | 24 | 24 | 0.0519572 |
| 1168 | 32 | 32 | 0.121691  |
| 1168 | 32 | 32 | 0.1507    |
| 1168 | 32 | 32 | 0.15177   |

Table 11: filter weak scaling - blocksize = 64