TU Wien
Faculty of Informatics
Research Group for Parallel Computing

Basics of Parallel Computing
2024S
Assignment 2

May 26, 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 / `a`	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 / `t`	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 OTP 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, 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"\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, 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
```

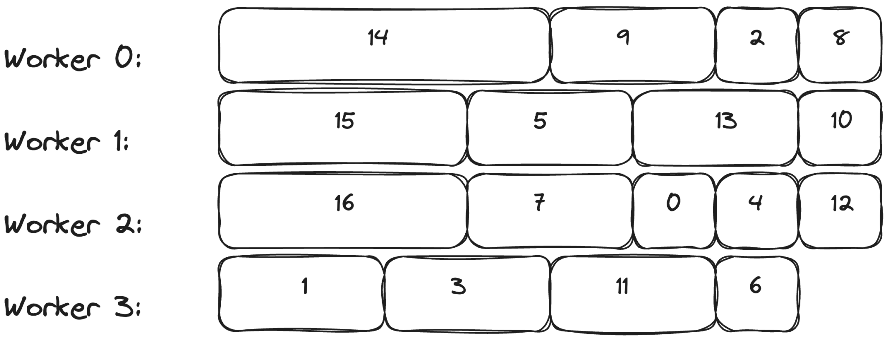

Figure 1: Gantt chart of the LPT schedule (which happens to be optimal).

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.

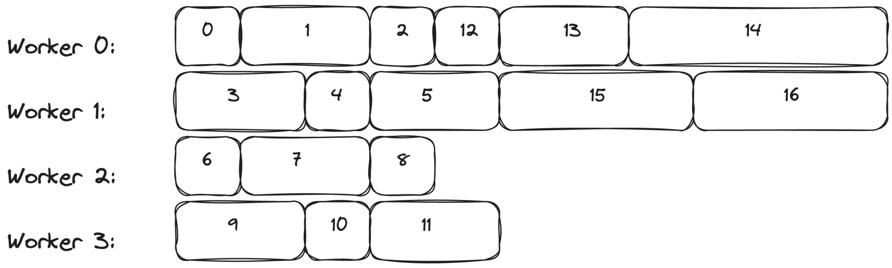

Figure 2: Gantt chart of the schedule `static,3`.

2.3 Schedule `dynamic,2`

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

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.

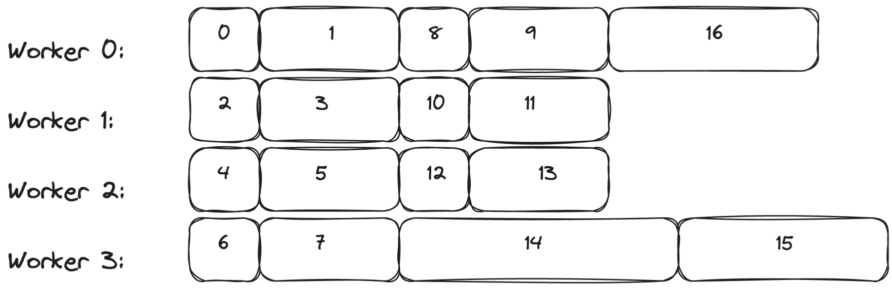

Figure 3: Gantt chart of the schedule `dynamic,2`.

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?

Results of executing the 3 given scripts 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
```

4.2 How often is the function `omp_tasks` called?

```
$ 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

$ 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

$ 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=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=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
```

5 Exercise 5

5.1 Parallelize the pixel computation

5.2 Running time analysis

5.3 Influence of schedule parameter

6 Exercise 6

6.1 Parallelize the filter computation

6.2 Strong scaling analysis

6.3 Weak scaling analysis

7 Addendum: Raw Data

1168	1	1	0.0603872
1168	1	1	0.0607409
1168	1	1	0.0600319
1168	2	1	0.196807
1168	2	1	0.2452
1168	2	1	0.19003
1168	4	1	3.45923
1168	4	1	3.90704
1168	4	1	3.45583
1168	8	1	5.395
1168	8	1	5.45436
1168	8	1	4.53896
1168	16	1	10.7055
1168	16	1	10.5507
1168	16	1	10.2593
1168	24	1	17.3402
1168	24	1	18.5362
1168	24	1	17.2604
1168	32	1	26.1056
1168	32	1	25.1663
1168	32	1	27.9486

Figure 4: Raw output from "filter strong" job.

1168	1	1	0.060196
1168	1	1	0.0609
1168	1	1	0.060195
1168	2	2	0.401089
1168	2	2	0.635222
1168	2	2	1.18221
1168	4	4	14.4383
1168	4	4	13.3359
1168	4	4	9.2267
1168	8	8	44.0875
1168	8	8	44.8141
1168	8	8	42.5354

Figure 5: Raw output from "weak scaling" job. Timed out on *slurmstepd* due to time out / time limit.

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	0.045880
90	4	0.045966
90	4	0.045863
90	8	0.031120
90	8	0.031132
90	8	0.031170
90	16	0.018182
90	16	0.018227
90	16	0.018220
90	24	0.013238
90	24	0.013257
90	24	0.013180
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	4.586490
1100	16	2.447131
1100	16	2.448894
1100	16	2.447200
1100	24	1.731222
1100	24	1.718731
1100	24	1.718424
1100	32	1.312658
1100	32	1.313263
1100	32	1.320209

Figure 6: Raw output from "juliap" job.

"static"	1100	16	2.450491
"static"	1100	16	2.448260
"static"	1100	16	2.449136

Figure 7: Raw output from "juliap2" job.