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`

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

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

2.1 Optimal Schedule

2.2 Schedule `static,3`

2.3 Schedule `dynamic,2`

3 Exercise 3

3.1 Fix the problems with this OpenMP code

4 Exercise 4

4.1 What is the output of the three different versions?

4.2 How often is the function `omp_tasks` called?

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 Exercise 7

7.1 Convert OpenMP code to CUDA

7.2 Running time analysis

7.3 Impact of block size

7.4 Running time: CPU vs GPU code

8 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 1: 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 2: 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 3: Raw output from "juliap" job.

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

Figure 4: Raw output from "juliap2" job.