TU Wien

Data Science

Basic of parallel computing

Exercise 2

Mal Kurteshi 11924480

June 3, 2021

1 Introduction

In this report i am presenting my solution regarding the second assignment in the course *191.114 Basics of Parallel Computing*.

2 Tasks

2.1 Exercise 1

```
My execution code regarding exercise 1
void main(int argc, char *argv[])
{
  const int n = 15;
  int a[15] = {};
  int t[15] = {};
  int i;
  #pragma omp parallel for schedule(runtime)
    for (i=0; i<n; i++) {
      a[i] = omp_get_thread_num ();
      t[omp_get_thread_num ()]++;
    }

  printf("a[15]: ");
    for (i = 0; i < 15; i++)
      printf( "%d ", a[i]);

  printf("\nt[4]: ");
    for (i = 0; i < 4; i++)
      printf( "%d ", t[i]);
}
```

1. What do a and t count?
   - a memorizes which thread has processed which iteration i from the for loop.
   - t counts how often a particular thread is used in the for loop.
2. Give the values for all elements in a and t

Table 1: Case a

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

Table 2: Case t

case / t	**0**	**1**	**2**	**3**
static	0	9	6	0
static 1	0	15	0	0
static 3	8	0	0	7
dynamic 1	6	3	5	1
dynamic 5	0	0	15	0
guided 2	8	1	6	0

3. What is a possible performance problem with the assignment to the t array?

   - The t array assignment can lead to the so-called false-sharing effect. It happens because the different values of the t array are in the same line of cache.

2.2 Exercise 2

```
#include <math.h>
#include <omp.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
int a[8] = {1,5,7,2,1,7,22,50};
int n = 8;
int count_odd = 0;
count_odd = omp_odd_counter(&a,n);
printf("sum: %d\n",count_odd);
return 0;
```

```
}
int omp_odd_counter(int *a, int n)
{
int i;
int count_odd = 0;
#pragma omp parallel for shared(a) shared(count_odd)
for(i=0; i<n; i++) {
if( a[i] % 2 == 1 ) {
#pragma omp critical
count_odd++;
 }
 }
 return count_odd;
}
```

With the initial code i was that there was a problem with obtaining diffrent results in every execution of the code. This was because we had a race on the $count\_ODD$ because update of the variable was happening by multiple threads concurrently. In order to fix this issue, first i updated the count to shared variable in the function other wise i was not able to obtain results. The problem with race of the count was fixed by inserting the critical directive in order to mark the update of count as a critical session, this way only one thread will be in critical section at any time and in this conditions we will obtain always the same correct result.

2.3 Exercise 3

1. After the programs executed with $OMP\_NUM\_THREADS = 4$ we obtained the same results in all three cases which is res=5.
2. Analyzing the code and the output we can see that in the version A program is paralleled in 4 threads and the function is only called for the omp master. In the version B the function is called for every thread in our case 4. In the version C the code is executed by the master thread only meaning that it is executed once per master.

2.4 Exercise 4

2.4.1 4.1 Juliap

Table 3: Juliap

N	P	Running_time(s)
100	1	0.161016
100	2	0.080857
100	4	0.040686
100	8	0.020804
100	16	0.010890
100	24	0.007758
100	32	0.018633
1000	1	16.141564
1000	2	8.187303
1000	4	4.102015
1000	8	2.067961
1000	16	1.049793
1000	24	0.705878
1000	32	0.547670

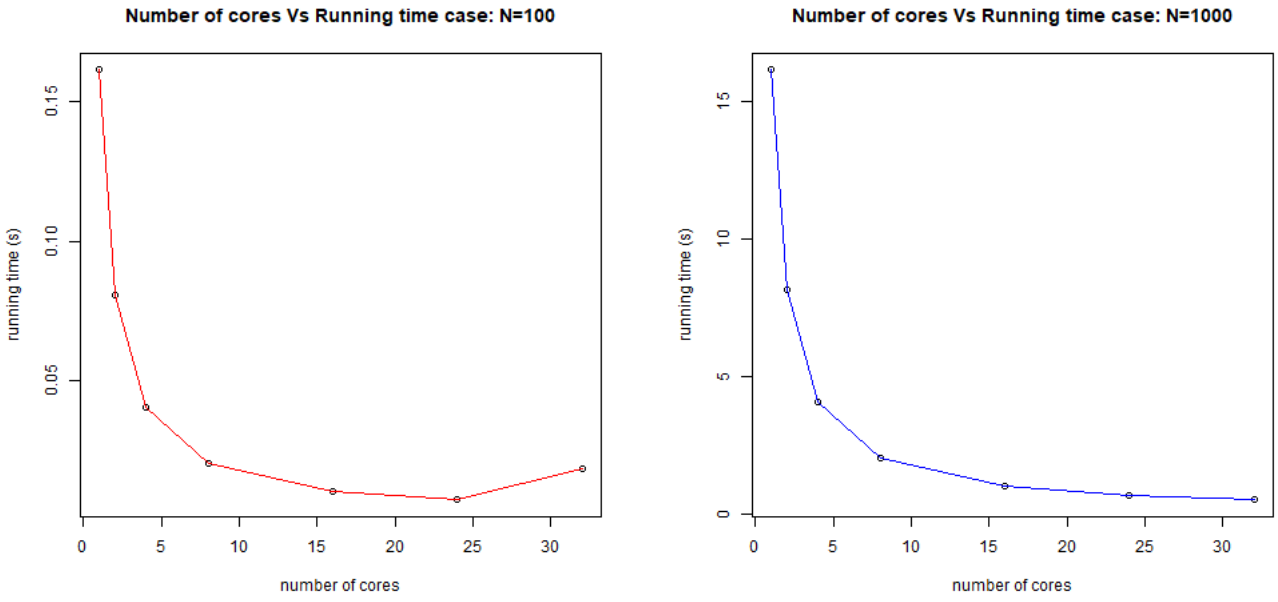


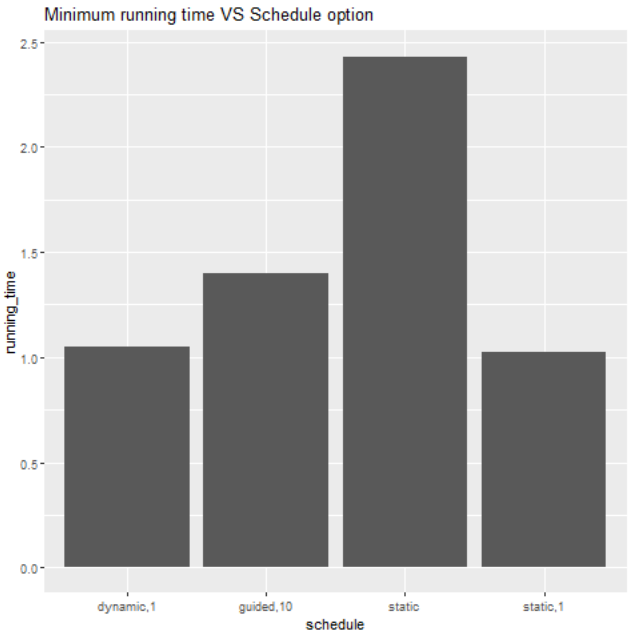
Figure 1: a) case N=100 b) case N=1000

Regarding the exercise 4 we where requested to parallelize the computation of each pixel of julia image using OpenMP. I tried several approaches regarding this case and there where several issues occurred, by the time i was trying to use *pragma omp critical* in the inner loops as a way to set execution by a single thread. i was facing either problems with running time or with the output which was not what i was expecting. As a final approach i used the *pragma omp parallel* for the nested loop and also the *pragma omp for collapse*(2) *schedule*(*runtime*), collapse(2) in order to merge the merge the outer two loops with the aim to increase the potential of parallelism. Also with the use of schedule runtime we are guaranteeing that the loops are executed them with the same number of threads using runtime scheduling so each thread will receive exactly the same iteration range in both parallel regions. Now with this set up the experiment was conducted. As we can see from the table and also from the plots, the running time of execution with the increase of the number of cores is going down quite drastically, the case with N=100 we have to note in the case of 32 cores we have a slight increase of running time, but the case with N=1000 is looking quite good in terms of efficiency of running time or latency giving us strong indication that the parallelization was good and efficient.

2.4.2 4.2 Juliap2

Table 4: Juliap2

Schedule	**N**	**P**	**Running_time(s)**
dynamic,1	1000	16	1.048249
guided,10	1000	16	1.401125
static	1000	16	2.430996
static,1	1000	16	1.026899



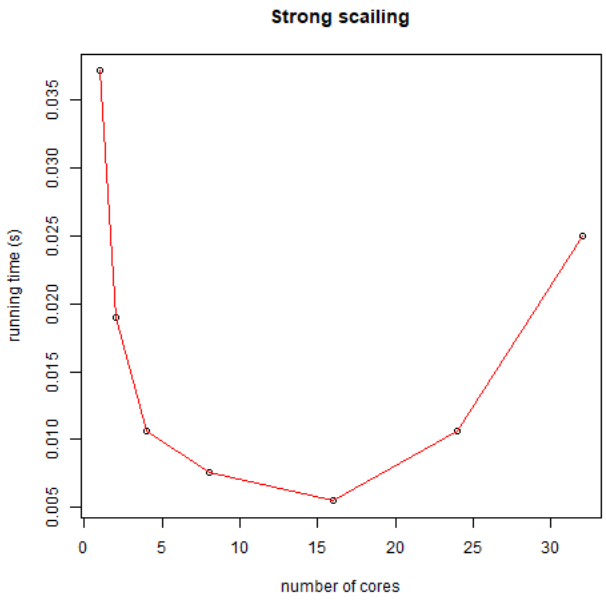
From the plot and table above we can see that we have experimented with different scheduling parameter options and analyzed the running time performance. From the plot we can see that the minimal running time was obtained with static scheduling with chunk size 1, followed by the dynamic scheduling with chunk size 1 then the guided scheduling with chunk size 10, and the worst running time was obtained with the static scheduling without specification of the chunk size. As an observation we need to note that we obtain similar values in dynamic,1 guided,1 and static,1 but very big values in static scheduling and there is a reason regarding this. First let us see the static clause which gives shows that the scheduling option is static meaning that openMp divides the iteration iterations into chunks of size chunk-size and it distributes the chunks to threads in a circular order in our case with chunk size 1 or no chunk-size specified, meaning that OpenMP divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread which is the reason also for getting the highest running time. For the dynamic,1 scheduling case openMP divides the iterations into chunks of size 1. Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available important to note that There is no particular order in which the chunks are distributed to the threads. The order changes each time when we execute the for loop. And least the guided scheduling type is similar with the dynamic scheduling but with the difference of the size of chunk.

2.5 Exercise 5

2.5.1 Strong Scaling

Table 5: Strong Scaling

r	P	Running_time(s)
1	1	0.03715400
1	2	0.01900940
1	4	0.01058980
1	8	0.00754428
1	16	0.00549996
1	24	0.01064400
1	32	0.02496750

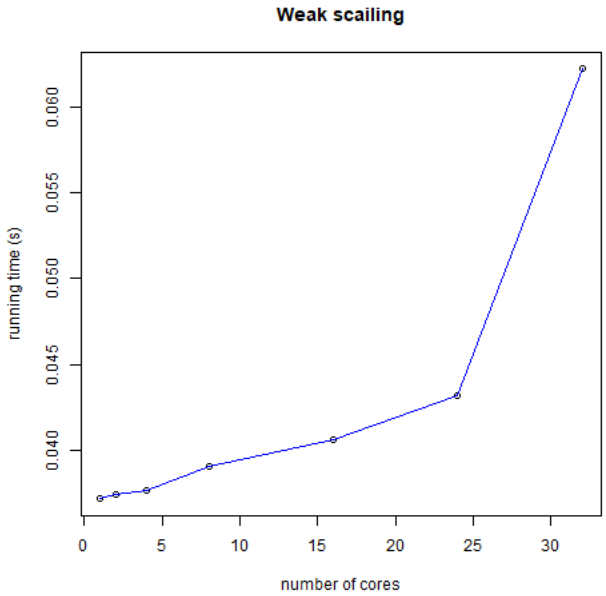


Regarding the strong scaling we can see from the plot the relation in between the running time and the number of cores. We see that with increase of the number of cores the running time is decreasing with exception of the two last cores 24 and 32 values which have a little higher value with tendency of increase. I have not calculated the speed up parameter regarding this case but from the plot of running times i would assume that the speed up will be increasing plot with exception of a little deviation at the cores 24 and 32 which will have a decrease. Considering these results i would say that we have a strong scaling paralleled case.

2.5.2 Weak Scaling

Table 6: Weak scaling

r	P	Running_time(s)
1	1	0.0371589
2	2	0.0374010
4	4	0.0376239
8	8	0.0390226
16	16	0.0405806
24	24	0.0431470
32	32	0.0622307



For the weak scaling we are conducting the experiment with increase of the parameters r and number of cores. The aim for the weak scaling was to have a linear constant curve in our case we have an increase in small values for the cases until the number of cores 32, where we have a drastic growth of the value of running time. As a conclusion our algorithm is not perfectly weak scaled but we have quite a linear case untill the number of cores 32.