Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing Lab **Institute of Computing**

Student: Fabian Flores Gobet Discussed with: -

Solution for Project 2

HPC Lab — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

This project will introduce you to parallel programming using OpenMP.

1. Parallel reduction operations using OpenMP [10 points]

For the first part of this project we were asked to optimize the dot product operation by implementing (*i*) OpenMP reduction clause and (*ii*) parallel and critical clauses.

1.1. (i) Reduction clause

For this part of the exercise I implemented the following code:

```
time_start = wall_time();
  for (int iterations = 0; iterations < NUM_ITERATIONS; iterations++) {
    alpha_parallel = 0.0;
    #pragma omp parallel for default(none) shared(a,b,N) reduction(+:
        alpha_parallel) num_threads(p)
    for (int i = 0; i < N; i++) {
      alpha_parallel += a[i] * b[i];
    }
  }
time_red = wall_time() - time_start;
```

The variables a, b and N are defined outside the scope and don't present any race condition within the for loop, hence these are simply shared by all running threads. On the other hand, *alpha_parallel* will be accessed and modified by all threads, thus being necessary do address this issue by declaring it in the reduction clause with a + operator, meaning that at the end of the loop the sum of the variable value for each thread should be summed.

1.2. (ii) Critical clause

For this part of the exercise I implemented the following code:

```
time_start = wall_time();
  for (int iterations = 0; iterations < NUM_ITERATIONS; iterations++) {
    alpha_parallel = 0.0;
    #pragma omp parallel for shared(a,b,N) num_threads(p)
    for (int i = 0; i < N; i++) {
      #pragma omp critical
      alpha_parallel += a[i] * b[i];
    }
  }
time_critical = wall_time() - time_start;
```

Like in 1.1, a, b and N are shared without potential race conditions but *alpha_parallel* has to be defined inside a pragma omp critical clause. This indicates that the scope of the statement can only be executed by one thread at the time, thus eliminating the race condition problem. This of course comes with the disadvantage of performance hindrance due to the fact that several threads will be trying to access and modify the current value of the parallel, having to wait if there's another one working on it.

1.3. Performance measures

In order to determine performance measures across several values of N and p, vector lengths and number of threads respectively, I've slightly modified the main function so it could take up to two arguments and then wrote a script in bash to automatically iterate through all these combination.

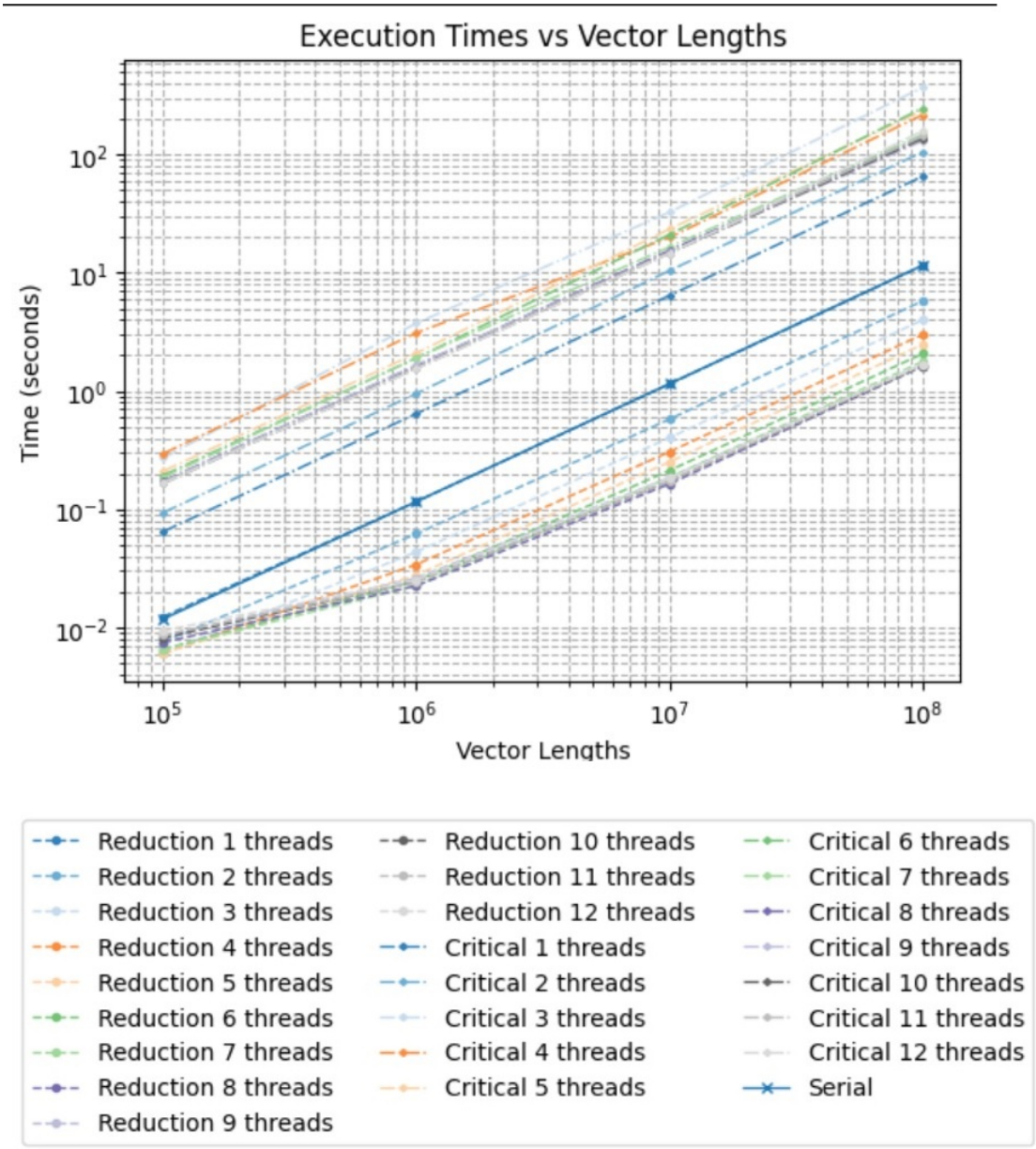
The added code to main:

```
if(argc == 3){
  p = atoi(argv[1]);
  N = atoi(argv[2]);
}
```

The script:

```
#!/bin/bash
for i in {0..3}; do
    n=$((100000 * (10 ** i)))
    for j in {1..12}; do
        ./dotProduct $j $n
    done
done
```

After running the script, I gathered all the data and made a plot in python ending up with the following graph



As we may notice on the graph, the best performance is always acquired by the reduction clause, in particular for the case of 8 running threads. The high values of the critical case may be justified by the omp critical clause introduced to avoid race condition problems and limiting one single access at a time to the variable.

2. The Mandelbrot set using OpenMP [30 points]

2.1. Exercise 1 and 2 of Mandelbrot

For this part of the project, since exercise 1 and 2 are closely tied together, I will show the whole code for it and then proceed to explain it and show some performance results

```
n = 0;
    while(x2 + y2 <= 4 && n <= MAX_ITERS){
            y = 2 * y * x + cy;
            x = x2 - y2 + cx;
            x2 = x * x;
            y2 = y * y;
            n++;
    }
nTotalIterationsCount += n;
```

This is the kernel of the Mandelbrot generating function. The while statement iterates until one of two conditions are broken: n reaches the MAX_ITERS threshold; $|z| > 2$, in other words for $x_2 = (Re(z))^2$ and $y_2 = (Im(z))^2$, $|z| = \sqrt{x_2 + y_2} \leq 2 \equiv x_2 + y_2 \leq 4$.

For every number computed, we must reset n to 0, count the number of iterations inside the kernel while, and then add them to $nTotalIterationsCount$.

For this sequential code, I ran picture sizes of 512x512 until 4096x406, always increase by a factor of 2. The results for this iterations were the following:

```
Total time:                 336847 millisconds
Image size:                 4096 x 4096 = 16777216 Pixels
Total number of iterations: 113614144161
Avg. time per pixel:        20.0776 microseconds
Avg. time per iteration:    0.00296483 microseconds
Iterations/second:          3.37287e+08
MFlop/s:                    2698.3

Total time:                 83632.1 millisconds
Image size:                 2048 x 2048 = 4194304 Pixels
Total number of iterations: 28404035419
Avg. time per pixel:        19.9395 microseconds
Avg. time per iteration:    0.00294437 microseconds
Iterations/second:          3.39631e+08
MFlop/s:                    2717.05

Total time:                 21014.4 millisconds
Image size:                 1024 x 1024 = 1048576 Pixels
Total number of iterations: 7103841491
Avg. time per pixel:        20.0409 microseconds
Avg. time per iteration:    0.00295817 microseconds
Iterations/second:          3.38046e+08
MFlop/s:                    2704.37

Total time:                 5256.54 millisconds
Image size:                 512 x 512 = 262144 Pixels
Total number of iterations: 1775318976
Avg. time per pixel:        20.0521 microseconds
Avg. time per iteration:    0.0029609 microseconds
Iterations/second:          3.37735e+08
MFlop/s:                    2701.88
```

These values will then be analysed against the optimized parallel version of the kernel.

2.2. Exercise 3 of Mandelbrot

In order to reduce the amount of overhead inherent to the creation of threads, I chose to paralleling the most outer loop of the code. But, since cy self increments by $fDeltaY$ on iteration, we have to take care of that data dependency. We can also express the value of cy as a function for the $j-th$ iteration, thus each thread should be able to easily compute the value needed for cy at each outer loop with very little cost. It's noteworthy that based on exercise 1, the number of threads chosen to run the parallel code was 8.

The resulting code is as follows:

```
#pragma omp parallel for private(cx, cy, x, y, x2, y2, i, n) reduction(+:
    nTotalIterationsCount)
  for (j = 0; j < IMAGE_HEIGHT; j++) {
    cx = MIN_X;
    cy = MIN_Y + j * fDeltaY; // this is the new computed cy
    for (i = 0; i < IMAGE_WIDTH; i++) {
      x = cx;
      y = cy;
      x2 = x * x;
      y2 = y * y;

      n = 0;
       while(x2 + y2 <= 4 && n <= MAX_ITERS){
              y = 2 * y * x + cy;
              x = x2 - y2 + cx;
              x2 = x * x;
              y2 = y * y;
              n++;
       }
       nTotalIterationsCount += n;

      int c = ((long)n * 255) / MAX_ITERS;
      png_plot(pPng, i, j, c, c, c);
      cx += fDeltaX;
    }
    //cy += fDeltaY; in this comment we can see the old cy
  }
```

As in exercise 1 and 2 of the Mandelbrot section, I also I ran picture sizes of 512x512 until 4096x406, always increase by a factor of 2, ending up with the following benchmarks:

```
Total time:                 113944 millisconds
Image size:                 4096 x 4096 = 16777216 Pixels
Total number of iterations: 113642427699
Avg. time per pixel:        6.79156 microseconds
Avg. time per iteration:    0.00100265 microseconds
Iterations/second:          9.97357e+08
MFlop/s:                    7978.86
```

```
Total time:                 28534.2 millisconds
Image size:                 2048 x 2048 = 4194304 Pixels
Total number of iterations: 28417858143
Avg. time per pixel:        6.80309 microseconds
Avg. time per iteration:    0.00100409 microseconds
Iterations/second:          9.95922e+08
MFlop/s:                    7967.38
```

```
Total time:                  7133.99 millisconds
Image size:                  1024 x 1024 = 1048576 Pixels
Total number of iterations: 7110758858
Avg. time per pixel:         6.80351 microseconds
Avg. time per iteration:     0.00100327 microseconds
Iterations/second:           9.96743e+08
MFlop/s:                     7973.94

Total time:                  1804.06 millisconds
Image size:                  512 x 512 = 262144 Pixels
Total number of iterations: 1778710421
Avg. time per pixel:         6.88193 microseconds
Avg. time per iteration:     0.00101425 microseconds
Iterations/second:           9.85951e+08
MFlop/s:                     7887.61
```

2.2.1. Discussion

By looking at the benchmark results from the sequential and versus the benchmark results from the paralleled code, we can immediately notice big differences, namely:

- All total times for the paralleled code are respectively 1/3 to 1/4 of the time for the sequential code. The same could be said regarding the average time per pixel, while average time per iteration has reduced to 1/2.
- The number of iterations per second e 3 times higher in the paralleled code, which is congruent with the first point of this list. This happens because by paralleling the process in 8 threads, we're getting much more computational output per clock cycle. Naturally, the number of MFlops/s follows a significant increase too
- The number of iterations stays the same across both codes, which is expected given the implementations. We didn't employ any loop unrolling strategy, we simply paralleled the process.

3. Bug hunt [15 points]

This section of the project presented some difficulties which I think are worth noting for future projects, namely the expected behaviour of the code in some of the exercises. The section regarding bug number 3 was not fully understood by me to what should be the expected output. I think that whenever we employ paralleling, one of the most important reasons to why we ought to be very careful is that we want to maintain the expected output of our code.

3.1. Bug 1

After employing the clause of $pragmaompparallelfor$, the very next piece of code following the prior statement should be the for loop statement.

Another bug present in this code was the misuse of the $omp_get_thread_num()$ method, which did not belong to the paralleled section. This means the we won't effectively get the desired result for each running thread because only inside paralleled zones are the creation of threads executed.

3.2. Bug 2

Assuming that the objective at sight is that for each thread to print the amount it could sum up to, we must declare *tid* and *total* variables as private in the pragma statement. The *nthreads* can remain by default as shared since that value will only be accessed and change by a single thread, where putting it in private would come with the disadvantage unnecessary memory overhead.

3.3. Bug 3

This bug was not fully understood because I didn't know what was the expected output and in all my executions I never got a deadlock type situation, though there is an observation worth noting. The barrier defined inside the function outside the effective parallel zone might lead to a never ending execution of the program because not all threads might execute the function.

3.4. Bug 4

The stack size is set to 8176Kb by default on my device. For a 1048 by 1048 array of doubles we need 8580.5 Kb per thread, which exceeds the stack size, thus giving the segmentation fault error. If we we make the calculus for which N such that $8N^2 \leq 8176$ we get that $N \leq 1022$. So if we try executing the code for $N = 1022$ and one thread only, the program runs, whereas for $N = 1023$ it gives the prior error. So we can then infer that for an array of this size, declared as private in the pragma statement directives, run by 8 threads, we greatly exceed the stack size because each thread will have a copy of that array.

```
N = 1048
(base) fabiangobet@Fabians-MacBook-Pro bugs % ./omp_bug4
zsh: segmentation fault  ./omp_bug4

N = 1022
(base) fabiangobet@Fabians-MacBook-Pro bugs % ./omp_bug4
Number of threads = 1
Thread 0 starting...
Thread 0 done. Last element= 2042.000000

N=1023
(base) fabiangobet@Fabians-MacBook-Pro bugs % ./omp_bug4
zsh: segmentation fault  ./omp_bug4
```

3.5. Bug 5

We have a 'pragma omp sections nowait' directive which indicates that the threads should proceed to run sections without the prior one having been completed. If we run at least 2 threads, it might happen that the first goes into the first section and locks lock A, and the second goes to the second section and locks lock B. They execute some code, and then the first one tries to lock lock b and the second tries to lock lock A. Since both this locks are already locked, the threads stay asleep until signalled by the lock to proceed again, once that lock has been unlocked. So this means that both threads end up in a dead lock situation, thus the execution of the program stays put and never ends.

4. Parallel histogram calculation using OpenMP [15 points]

For this section of the project I made two different codes to calculate the histogram of 16 bins. The first code doesn't scale with the number of threads employed, while the second does.

Since the access to *dist* array is done by $vec[i]$, where there may be equal values along the array *vec*, we have a race condition to which we should attend.

4.1. Non scaling parallel

The core code for the non scaling paralleled version is as follows:

```
#pragma omp parallel
{
    int nthreads;
    int thread_num = omp_get_thread_num();
    long local_dist[BINS] = {0};

    #pragma omp for
    for (long i = 0; i < VEC_SIZE; ++i) {
        local_dist[vec[i]]++;
    }

    for(int i=0; i<BINS; i++){
      #pragma omp atomic
      dist[i] += local_dist[i];
    }
}
```

Since there is no reduction clause applicable to arrays, we have to perform a 'manual' reduction. Each thread will have it's own *local_dist* array of size $BINS$, initialized with all 0's. We then declare a 'pragma omp for' statement so that the for loop's iterations may be divided amongst the threads.

In the end of the parallel section we combine all the values summed by each thread in their *local_dist* array into the shared *dist* array whilst employing atomic operations on it.

It is also worth noting that default states for the variables were used by leveraging the statements.

For this code deployment I got the following benchmark performances:

```
#pragma omp parallel
base) fabiangobet@Fabians-MacBook-Pro hist % export OMP_NUM_THREADS=1
(base) fabiangobet@Fabians-MacBook-Pro hist % ./hist_omp
dist[0]=93
dist[1]=3285
dist[2]=85350
dist[3]=1260714
dist[4]=10871742
dist[5]=54586161
dist[6]=159818704
dist[7]=273378686
dist[8]=273376192
dist[9]=159818436
dist[10]=54574834
dist[11]=10876069
dist[12]=1261215
dist[13]=85045
dist[14]=3397
dist[15]=77
Time: 0.884913 sec


(base) fabiangobet@Fabians-MacBook-Pro hist % export OMP_NUM_THREADS=8
(base) fabiangobet@Fabians-MacBook-Pro hist % ./hist_omp
dist[0]=93
dist[1]=3285
dist[2]=85350
dist[3]=1260714
dist[4]=10871742
dist[5]=54586161
dist[6]=159818704
dist[7]=273378686
dist[8]=273376192
dist[9]=159818436
dist[10]=54574834
dist[11]=10876069
dist[12]=1261215
dist[13]=85045
dist[14]=3397
dist[15]=77
Time: 0.125224 sec
```

4.2. Scaling parallel

In order to make the runtime scale by a function of the number of threads and VEC_SIZE, i chose to parallel the for loop whilst employing a pragma omp atomic statement on the access to dist, which is where we encounter a race condition.
The code is as follows:

```
#pragma omp parallel for shared(dist,vec)
  for (long i = 0; i < VEC_SIZE; ++i) {
    #pragma omp atomic
    dist[vec[i]]++;
  }
```

And for the performance benchmarks I got:

```
1 thread
(base) fabiangobet@Fabians-MacBook-Pro hist % ./hist_omp
dist[0]=93
dist[1]=3285
dist[2]=85350
dist[3]=1260714
dist[4]=10871742
dist[5]=54586161
dist[6]=159818704
dist[7]=273378686
dist[8]=273376192
dist[9]=159818436
dist[10]=54574834
dist[11]=10876069
dist[12]=1261215
dist[13]=85045
dist[14]=3397
dist[15]=77
Time: 2.29752 sec

8 threads
(base) fabiangobet@Fabians-MacBook-Pro hist % ./hist_omp
dist[0]=93
dist[1]=3285
dist[2]=85350
dist[3]=1260714
dist[4]=10871742
dist[5]=54586161
dist[6]=159818704
dist[7]=273378686
dist[8]=273376192
dist[9]=159818436
dist[10]=54574834
dist[11]=10876069
dist[12]=1261215
dist[13]=85045
dist[14]=3397
dist[15]=77
Time: 13.3301 sec
```

4.3. Sequential and benchmarks discussion

For the sequential code we got the following benchmarks:

```
(base) fabiangobet@Fabians-MacBook-Pro hist % ./hist_seq
dist[0]=93
dist[1]=3285
dist[2]=85350
dist[3]=1260714
dist[4]=10871742
dist[5]=54586161
dist[6]=159818704
```

```
dist[7]=273378686
dist[8]=273376192
dist[9]=159818436
dist[10]=54574834
dist[11]=10876069
dist[12]=1261215
dist[13]=85045
dist[14]=3397
dist[15]=77
Time: 0.881178 sec
```

We can now see by comparison that the scaling code performs worst then the sequential, and that the non scaling code performs better than the sequential for both 1 and 8 threads. this happens because in the scaling code we employ an atomic pragma statement, which means that that line of code can only be executed by one thread at the time, thus leading to cumulative overhead in running time due to thread state management.

On the other hand, the non scaling code parallel's the addition into an array part of the code, and performs a manual reduction with an atomic operation which is rather quick.

5. Parallel loop dependencies with OpenMP [15 points]

The code provided has a for loop with a recursive multiplication data dependency, so paralleling this block will not be straight forward. In order to achieve a significant gain performance wise I inspired myself in the knowledge of the prior project (blocking tiles).

For this, I defined a block size variabled named *blocksize* (later to be used as a tuning hyper-parameter), a variable double named $up2$, which differs from up and the first element of multiplication is 1 and not Sn, and sequentially computed all the values for the first *blocksize* elements of the array while also recursively computing $up2* = up$.

After having computed the first block of values, I perform a for loop with a *blocksize* step, starting at index *blocksize*, and within this loop I perform another for loop which looks at the values of the previous block and multiplies then $up2$, saving the result to the current index.

The last for loop,,since only depends on values that are already computed, can be paralleled! Furthermore, the loss in number precision is less because the error isn't carried and increased in every sequential iteration, but rather in *blocksize* steps.

In order to optimize this code, I compute *blocksize* by dividing N, the array size, by a predefined value *div*, which is then used as an hyper-parameter along with the number of threads.

The final code looks like as follows:

```
int div = 100;
int tnum = 7;

/*
if(argc==3) {
  div = atoi(argv[1]);
  tnum = atoi(argv[2]);
}
*/

long long int blocksize = (long long int)N/div;
int mini = (blocksize < N+1 ? blocksize : N+1);

double up2 = 1;
```

```
  for( int i=0; i < mini; i++){
    opt[i] = Sn;
    Sn *= up;
    up2 *= up;
  }

  long long int i, j;

  for(i=blocksize; i<N+1; i+=blocksize){
    #pragma omp parallel for shared(i,blocksize,opt,N) num_threads(tnum)
    for(j=i; j<(i+blocksize>N+1 ? N+1 : i+blocksize); j++)
      opt[j] = opt[j-blocksize]*up2;
  }

  Sn = opt[N];
```

Notice that some code was left commented. This code was used to compute several benchmark values from 1 to 10 threads, and for b*div* values of 10 to 100000 by a step factor of 10.

The script looks as follows:

```
#!/bin/bash
for j in {1..10}; do
    i=10
    while [ $i -le 100000 ]; do
        ./recur_omp $i $j
        i=$((i * 10))
    done
done
```

5.1. Results and optimal parameters

After computing all the values, I organized them in a table which can be easily analised in order to determine the best number of threads and *div* size

Table 1: Parallel RunTimes for Different Threads and Division Factors

Threads	Division Factor				
	10	100	1000	10000	100000
1	3.974430	2.988011	3.087585	3.413681	3.454532
2	3.303520	2.785839	2.614230	2.932606	3.215592
3	3.131254	2.640935	2.462706	2.558268	3.405227
4	3.026925	2.607036	2.448964	2.464168	3.775031
5	3.034897	2.426189	2.470021	2.549450	3.704696
6	2.901893	2.482427	2.553568	2.748219	4.087580
7	2.919604	2.353466	2.622207	2.733264	4.092667
8	2.818406	2.414675	2.690447	2.940233	4.254213
9	3.392950	2.502451	2.800120	3.074194	4.677608
10	2.885671	2.478327	2.887129	2.716080	4.710007

By observing the table we can infer that the best combo is 7 threads ran by a *div* of 100.
We can then compute the sequential code provided to us for the loop-dependencies and get the following result

```
(base) fabiangobet@Fabians-MacBook-Pro loop-dependencies % ./recur_seq
Sequential RunTime :   4.222464 seconds
Final Result Sn    :   485165097.62511122
Result ||opt||^2_2 :   5884629305179575.000000
```

The Runtime in the paralleled version is almost half the time in the sequential code. Furthermore, the values of Sn are equal, but the result for $||opt||^2_2$ is slightly different because in the paralleled version there is less loss regarding the rouding of doubles through iterations, as explained before.