Università della Svizzera italiana

Institute of Computing
CI

High-Performance Computing **2021**

Student: Alessandro De Grandi Discussed with: Davide Casnici, Stefano Eportentosi

Solution for Project 3

Due date: 24.10.2021 (midnight)

HPC 2021 — 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]

2. The Mandelbrot set using OpenMP [30 points]

3. Bug hunt [15 points]

4. Parallel histogram calculation using OpenMP [15 points]

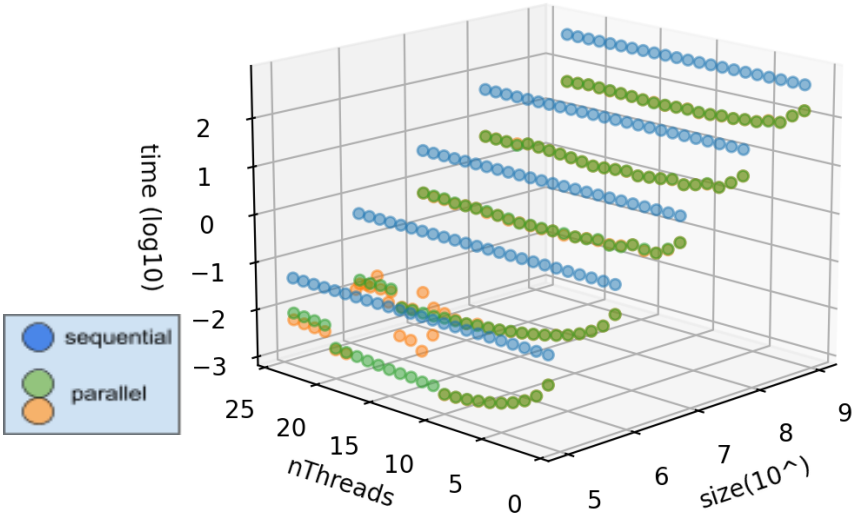
5. Parallel loop dependencies with OpenMP [15 points]

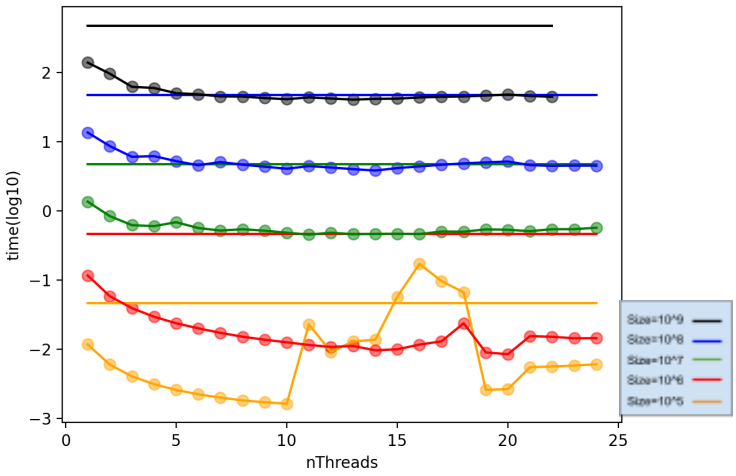
6. Task: Quality of the Report [15 Points]

Problem 1: Array dot product

The following graphs summarize the performance of parallelized and serialized dot products of an array, for various array lengths and number of threads. The two versions of the parallelized program(using: reduction,critical pragmas) show roughly the same performance.

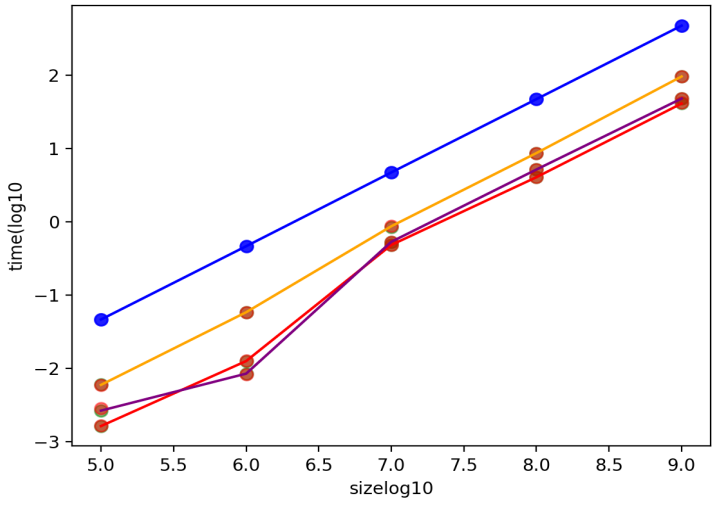
The array sizes used increase exponentially(10^), as a consequence of it so does the computation time, therefore these two parameters are plotted using logarithmic scales.
To save time, the serialized version is computed only once per size, that's why it's constant.

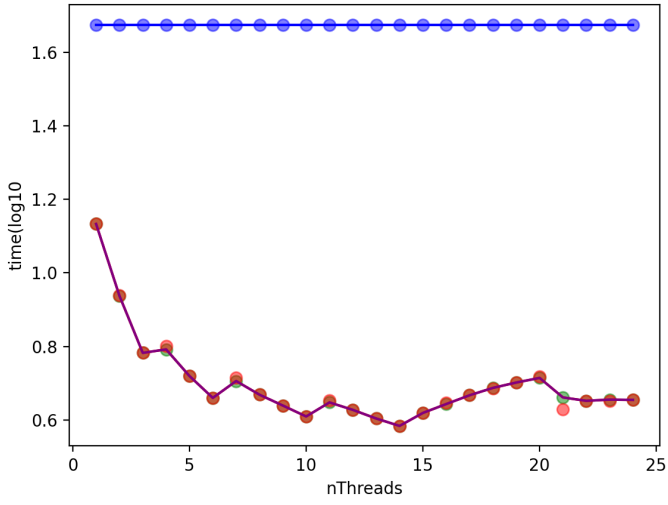




From the 2D representation, we can see how the serialized version execution time, is very similar to the parallelized version time of the "upper size level"(eg: look how the straight red line fits the green dotted one). Meaning that the parallelized version with many threads is roughly 10 times faster for any of these sizes.

The following graphs. On the left , we can see (in order of decreasing time), serial, 2, 10 ,20 threads; what is highlighted by this graph is that from 10 to 20 threads there is not much difference in performance. On the right we can again see that performance stops increasing after roughly 10 Threads, maybe due to the overhead of handling many threads, or the limit on nThreads that can run in parallel.





Problem 2: MandelBrot

Every pixel coordinate is mapped to the parameter plane :

```
#define MIN_X -2.1
#define MAX_X 0.7
#define MIN_Y -1.4
#define MAX_Y 1.4
```

So for every pixel we have the real and imaginary part of the complex number, mapped between these boundaries.

Then for every pixel, Iterate on these numbers using the formula $z = z^2 + c$. If the magnitude of the complex number reaches 2 then it will diverge, and it's not part of the set.

The parallelized version distributes the inner loop over a number of threads. Each thread is basically calculating its columns of the image row. Being careful of what variables are shared and private, for example cx(x coordinate in the parameter plane) is local and needs to be adjusted so that each thread calculates the correct point. Also a reduction is needed to correctly calculate the number of iterations.

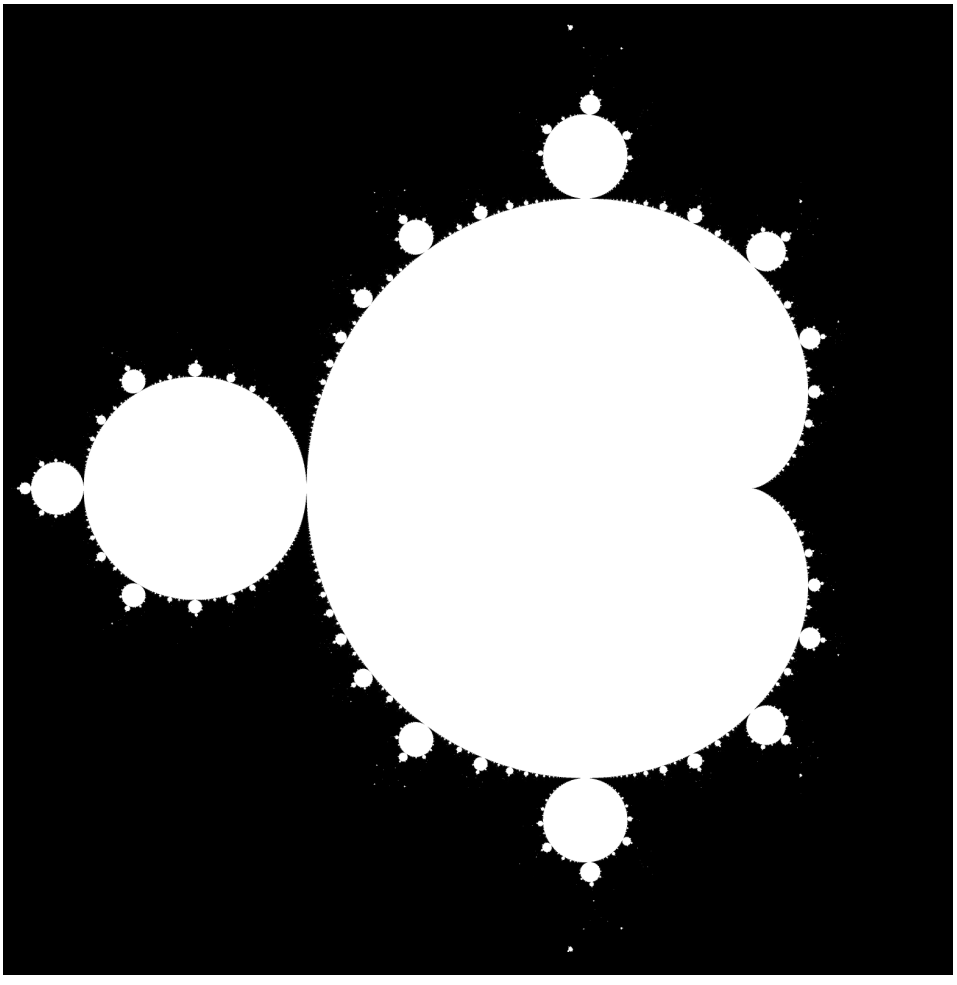
Performances:

Parallel:

[stud44@icsnode24 mandel]$ export OMP_NUM_THREADS=10
[stud44@icsnode24 mandel]$ gcc mandel_parallel.c pngwriter.c -lpng -lm -fopenmp -o mandel_parallel
[stud44@icsnode24 mandel]$ ./mandel_parallel
MaxIter: 35207
X: -2.100000->0.700000
Y: -1.400000->1.400000
Total time: 274683 milliseconds
Image size: 4096 x 4096 = 16777216 Pixels
Total number of iterations: 113546226265
Avg. time per pixel: 16.3724 microseconds
Avg. time per iteration: 0.00241913 microseconds
Iterations/second: 4.13372e+08
MFlop/s: 3306.97
(4.5 min)

Sequential:

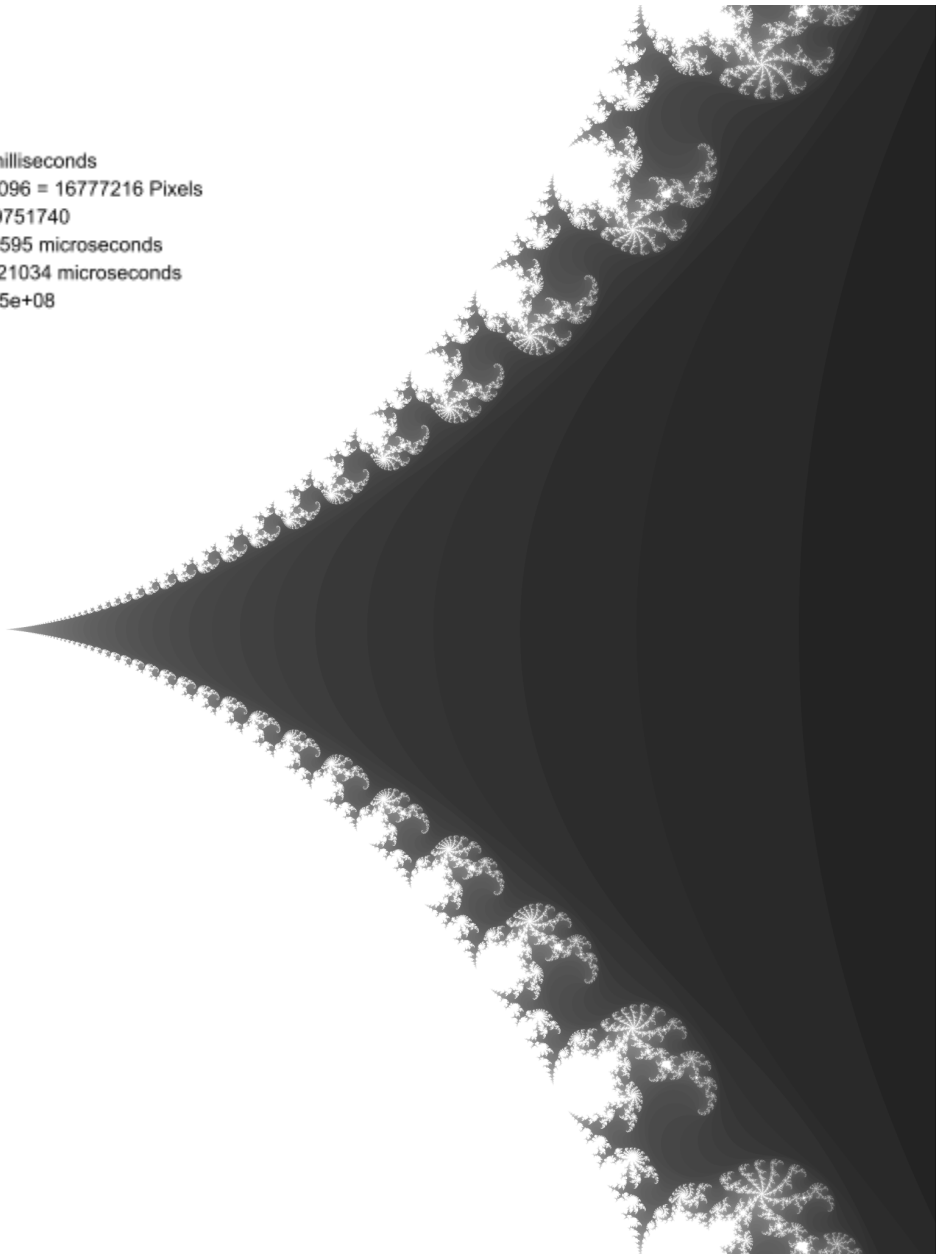
[stud44@icsnode24 mandel]$ gcc mandel_seq.c
pngwriter.c -lpng -o mandel_seq
[stud44@icsnode24 mandel]$ ./mandel_seq
Total time: 1.05025e+06 milliseconds
Image size: 4096 x 4096 = 16777216 Pixels
Total number of iterations: 113546289757
Avg. time per pixel: 62.5996 microseconds
Avg. time per iteration: 0.0092495 microseconds
Iterations/second: 1.08114e+08
MFlop/s: 864.912
(17.5 min)



Increasing the number of iterations leads to a more precise calculation of the set. Increasing the image size leads to a “thinner” sampling of parameter space, meaning better resolution. One could also get better resolution restricting the parameters space, focus on a more precise area of the set, and “zoom in”, for example at the edges of the set, where all the cool patterns appear:

Elephants Valley:

MaxIter: 100
X: 0.200000->0.400000
Y: -0.100000->0.100000
Total time: 1343.18 milliseconds
Image size: 4096 x 4096 = 16777216 Pixels
Total number of iterations: 1109751740
Avg. time per pixel: 0.0800595 microseconds
Avg. time per iteration: 0.00121034 microseconds
Iterations/second: 8.26215e+08
MFlop/s: 6609.72



Problem 3: Bug hunt

1) -Compile error: (after removal of parenthesis)Tid was after a parallel for pragma , only a for block can be after that
   -Solution: move `tid = omp_get_thread_num();` inside the for loop
2) Runtime: hard to “debug” without information on what the correct behaviour is supposed to be.
   Probably the fact that tid is shared, make it private:
   Output:

```
[stud44@icsnode37 bugs]$ export OMP_NUM_THREADS=4
[stud44@icsnode37 bugs]$ ./omp_bug2
Number of threads = 4
Thread 0 is starting...
Thread 3 is starting...
Thread 2 is starting...
Thread 1 is starting...
Thread 1 is done! Total= 1.982973e+11
Thread 3 is done! Total= 1.982973e+11
Thread 2 is done! Total= 1.982973e+11
Thread 0 is done! Total= 1.982973e+11
```

3) In this example there are only 2 omp sections so there is no need to use more than 2 threads.
   export OMP_NUM_THREADS=2
   Each thread is doing one section and the program is exiting without a problem.

   The other threads were stuck on a barrier:
   https://stackoverflow.com/questions/30671651/sections-and-openmp-code-hangs-sometimes/30674028
   “Make sure to place barriers only in places where you know that all threads will execute it.”

   The barrier in the function printresults is blocking the program.Remove it for a more stable solution.

4) Apparently the stack size(private variables ecc) for threads is fixed, the preprocessor allocates it statically, I suppose, unlike with the main thread that grows dynamically. So by using too many threads the virtual memory size available might be exceeded. Or one thread might exceed its own stack size limit, and this is the case because the problem persists running 1 thread only.

   For example reducing N fixes the problem. Less memory needed for a thread.
   Otherwise if large N is needed, we need to request a larger stack for threads if allowed:

   1048*1048*8= 8786432 bytes+ #= 9MB

5)Fixed locks order

```
    #pragma omp section
    {
      printf("Thread %d initializing a[]\n", tid);

      omp_set_lock(&locka);
      for (i = 0; i < N; i++)
        a[i] = i * DELTA;
      omp_unset_lock(&locka);

      omp_set_lock(&lockb);
      omp_set_lock(&locka);
      printf("Thread %d adding a[] to b[]\n", tid);
      for (i = 0; i < N; i++)
        b[i] += a[i];
      omp_unset_lock(&lockb);
      omp_unset_lock(&locka);
    }

#pragma omp section
    {
      printf("Thread %d initializing b[]\n", tid);

      omp_set_lock(&lockb);
      for (i = 0; i < N; i++)
        b[i] = i * PI;
      omp_unset_lock(&lockb);

      omp_set_lock(&lockb);
      omp_set_lock(&locka);
      printf("Thread %d adding b[] to a[]\n", tid);
      for (i = 0; i < N; i++)
        a[i] += b[i];
      omp_unset_lock(&lockb);
      omp_unset_lock(&locka);

    }
```

Both threads were waiting on a lock that was held by the other.

In the fixed version this does not happen, each thread can do it's own initialization, then it waits for the other to finish, then one of them acquires both locks, and releases both in order for the other thread's use.

Problem 4: Histogram

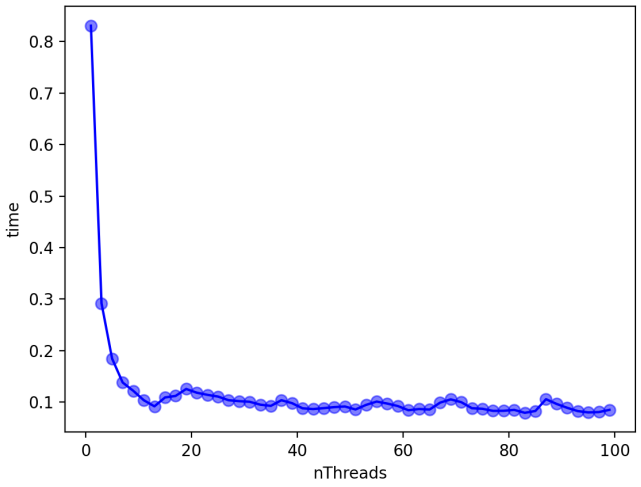
In hist_omp the program shares the array between a number of threads(nThreads), each thread shares the same array but accesses and calculates the histogram for only its portion of it on a local histogram. Then the access to the shared histogram is synchronized using a lock, each thread saves its results there in a serial manner. (similar results with pragma critical or reduction, also tried synchronizing each bin with its own lock)

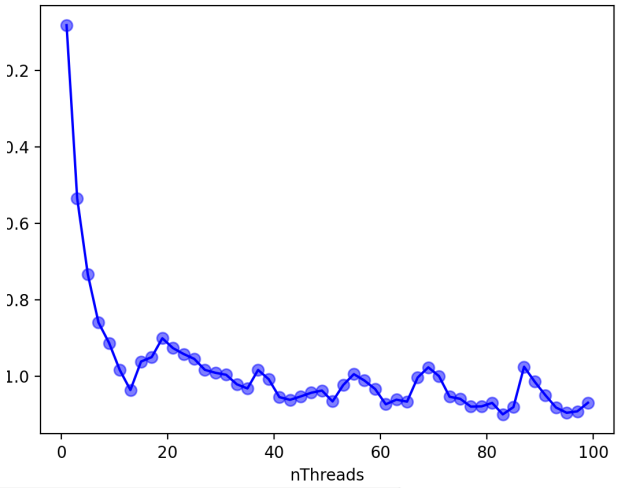
Scaling of the solution:

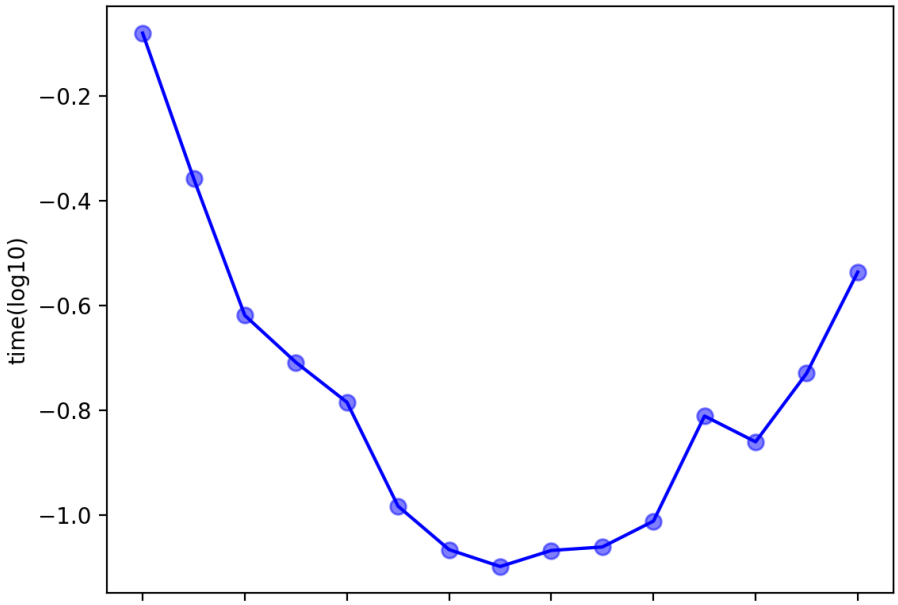
The performance scales very well up to around 15 threads, then the performance benefits start slowing down, and it slowly keeps going faster up to 128 Threads.

1 Thread = 0.83048 s
13 Threads = 0.0920341s
128 Threads = 0.0799048;

After 128 Threads there is noticeable bottleneck by synchronizing so many threads on one lock(critical section) and also there is a limit on the number of threads that can actually run in parallel.







Problem 5

Sequential:
With N = 3, up = 3, Sn =2 this program is calculating:
Opt=[2 , 6 , 8, 54]
[2, 2*3, 2*3*3, 2*3*3*3]
[2*3^0, 2*3^1, 2*3^2, 2*3^3]

So the smart way to do it is using the result from the previous iterations and multiplying just once. But by doing it the smart way, the iterations are not independent from each other. This of course is a problem if we want to parallelize the program.

Parallel:
Make the iterations completely independent:
It's clear from the example above that the program is actually calculating the following formula.

Iteration(N) = base * up^N
where base is the initial Sn and N is the iteration number starting from 0.

So implementing that formula in the for loop and distributing the iterations to any number of threads will do the job, with the same results.
The problem is that the efficiency gained from the sequential "trick" is lost; every thread needs to calculate from the beginning and do many multiplications. So the parallelized version is much slower.

Transforming an O(n) algorithm into an O($\sum^n \mathrm{n}$) is a really bad idea, it scales terribly of course, but it is a great example of a non-parallelizable algorithm.

```
N: [20,2000,200000,20000000,2000000000]
Time serial: [0.000000,0.000007,0.000837,0.067525, 8.816151])
Time parallel: [0.000024,0.000142,0.011969 ,1.161233,113.938886])
```