Università della Svizzera italiana | Institute of Computing CI

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

Student: ALESSIO BARLETTA

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 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 *(20 Points)*

1.1. DotProduct

1.1.1.

I implemented the dot product using openMP, the final code in is DotProduct.cpp, I did slight modifications to the original code to make it easier to collect data for the next task.

Moreover, I firstly implemented the critical version just by parallelizing and making the update of the result variable critical, this version was very unefficient, and it was unfeasible to run it on array sizes larger than 10 milion elements.

Then I implemented the current version, which uses a very similar schema to what the reduction clause does, each thread computes a local sum, and at the end of the parallel region all the local sums are added to the global result using a critical section.

This version is on par with the version using the reduction clause.

1.1.2.

To perform a strong scaling analysis, I run the code on all the thread sizes suggested, and I created one plot for each array size.

The results are shown in the following plots:

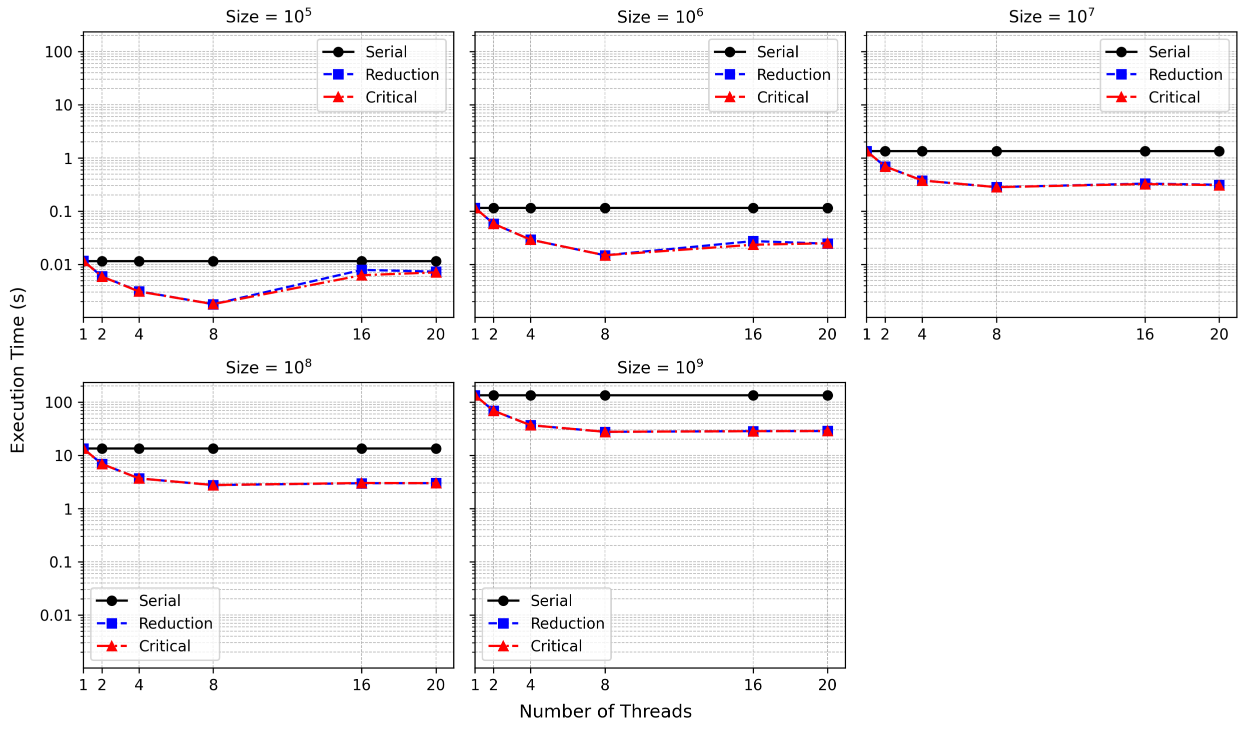

Figure 1: Strong scaling analysis for the dot product

We can see that the execution time decreases with the number of threads, but just to a certain point. After that, it remains almost constant or even increases a bit.

The difference between using critical or the reduction clause is very small, and the two versions are almost equivalent. This is because my critical version is using the same schema as the reduction clause, so the results are similar and this is to be expected.

1.1.3.

To perform a parallel efficiency analysis, I used the same data as before, and I created one plot for each array size.

On the y-axis I plotted the parallel efficiency, computed as

$$E = \frac{T_1}{T_P \cdot P}$$

where T_1 is the execution time with one thread, T_P is the execution time with P threads. The results are shown in the following plots:

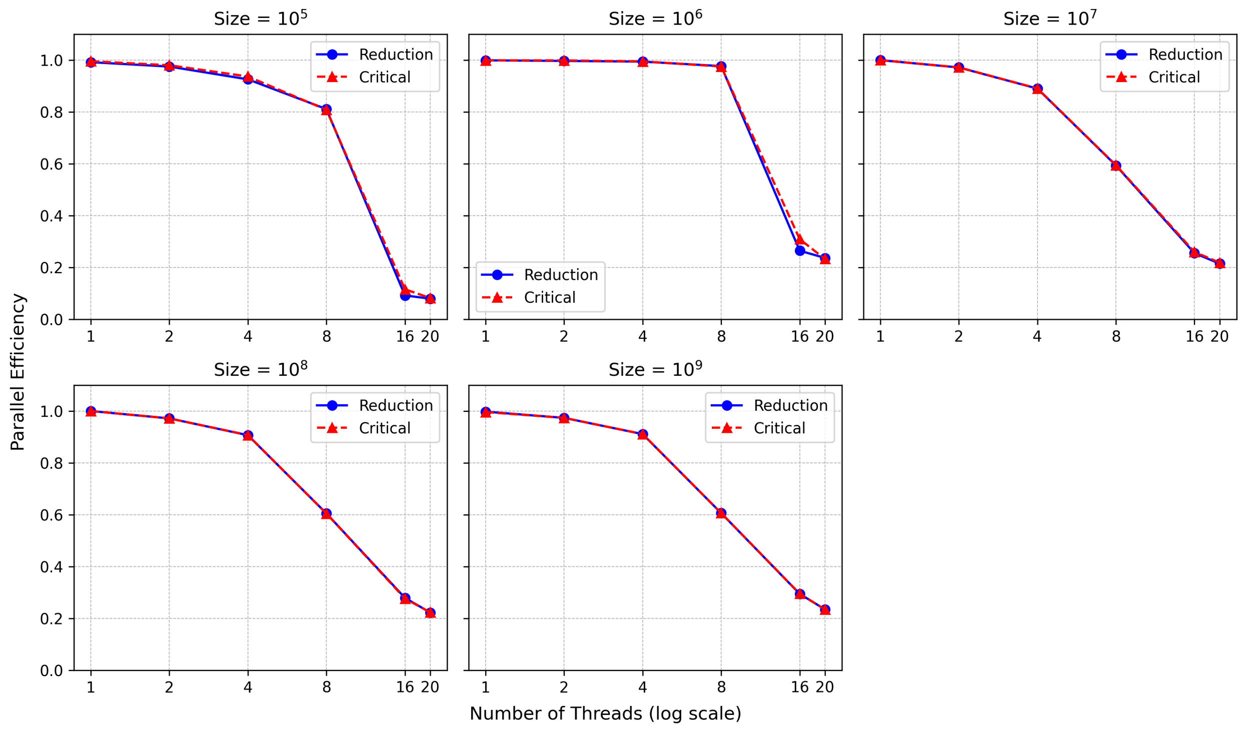

Figure 2: Parallel efficiency analysis for the dot product

We can see that the efficiency generally decreases when the thread count excedeeds 8 threads, and that the difference between 16 or 20 is minimal. This might be due to the overhead introduced by the parallelization, which becomes more significant as the number of threads increases. Using a multithread version of dot product is efficient for all the array sizes tested, but we have to make sure to choose a suitable number of threads to avoid wasting resources or in some cases even increasing the execution time.

This results are coherent with what we observed in the strong scaling analysis. Adding more threads helps to reduce the execution time, but just to a certain point, after that the increase is minimal or even negative, drastically reducing efficiency.

1.2. Approximating pi

1.2.1.

I implemented the parallel version of pi approximation using openMP, the final code in is pi.cpp, to implement it I used the reduction clause, because it is one of the easier and most efficient way to do it, based on what I learned from the previous dot product exercise.

1.2.2.

As before, I slightly modified the original code to make it easier to collect data for the speedup plot, and I run the code with the suggested number of thread on a fixed dimension of 10^{10} intervals. The results are shown in the following plot:

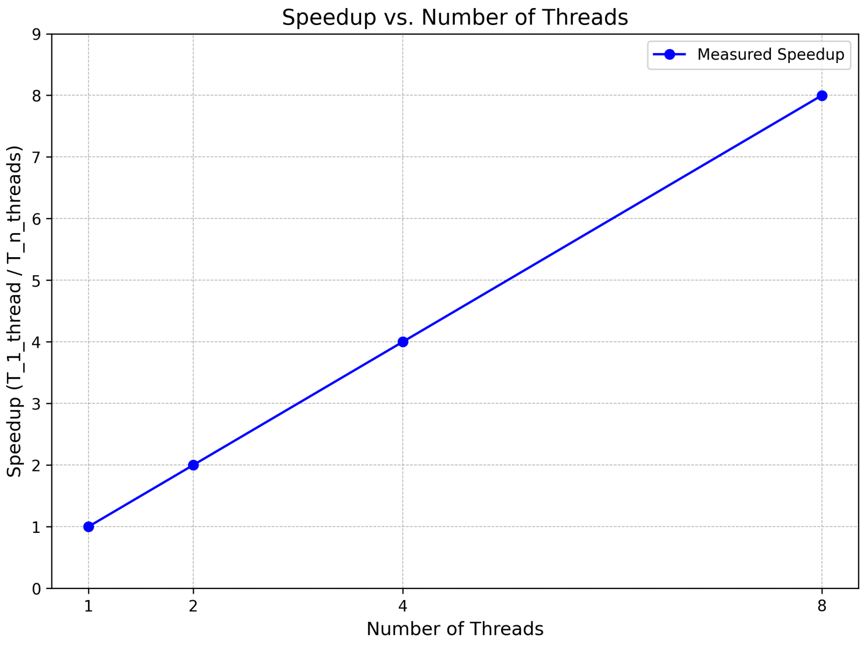

Figure 3: Speedup analysis for the pi approximation

We can see that the speedup related to the number of threads is almost linear up to eight threads. This is expected, and suggest a parallel efficiency very close to 1 up to this point. As speedup is defined as $S = \frac{T_1}{T_P}$, we can also say that the running time almost halves every time we double the number of threads, up to 8 threads.

2. The Mandelbrot set using OpenMP *(20 Points)*

2.1.

My implementation of the kernel is visible in the mandel_seq.cpp file. What I did is to finish the kernel as described in the assignment, I just had to make sure that i correctly implemented the complex number multiplication.

2.2.

To count the number of iterations I just iteratively summed the n operation that I do for each sequence until the escape condition is met.

2.3.

To parallelize the mandelbrot calculation, I did a slight modification to the code, now I'm calculating cx and cy using the loop index as reference, compared to the original version where we were incrementing them at each iteration. This way the iterations of both loops are independent of each other.

This allowed me to parallelize both the outer and inner loop using openMP pragmas, specifically I used the parallel for pragma with the collapse(2) clause, to collapse the two loops into one larger loop that can be parallelized. Moreover, I used the schedule(dynamic) clause to distribute the iterations among the threads dynamically, this is useful because the iteration time for each point in the Mandelbrot set can vary significantly, as for example, the points near the border of the set will escape much quicker than the points inside the set, resulting in less operations. Using dynamic scheduling my aim was to balance the workload among the threads more evenly. What I did not want was having one thread taking a batch of points that are all inside the set and thus taking a long time to compute, while another thread takes a batch of points that are all outside the set and thus taking a much shorter time to finish.

The strong scaling analysis is shown in the following plot:

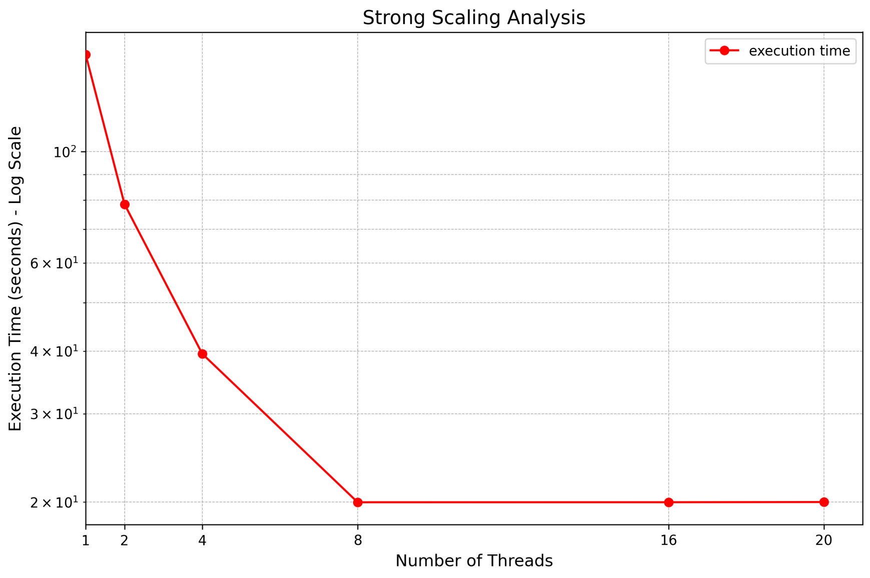

Figure 4: Strong scaling analysis for the mandelbrot set calculation

For this graph we can clearly see how performance almost doubles every time we double the number of threads, up to 8 threads. After that, as we have also seen in the previous exercise, there is not any notable improvement.

What is also really interesting to notice in my opinion, is that the performance improvement of the 8 threads parallel version compared to the sequential version is more than 8 times.

3. Bug hunt *(15 Points)*

3.1.

The problem here is that after the `#pragma parallel for ...` construct, the next instruction is not a for loop, but an assignment operation on the *tid* variable. This generates a compile error.

To fix this we can just move the `#pragma parallel for ...` statement just before the for loop, and declare outside it a parallel region where we can initialize the *tid* variable.

3.2.

The problem here is that to avoid race condition the *tid* variable should be declared private, and we should use the reduction clause for the *sum* variable. If we do not do so, we will most likely have an incorrect result at the end of the execution. Moreover, the print is done outside the parallel region, so it will just display the last thread to finish and not the partial sum for all the threads.

To fix this we can declare *tid* as private in the parallel pragma, and use the reduction clause for the *sum* variable. Finally, we can move the print statement inside the parallel region, so that each thread prints its own partial sum. if we want each thread to print it's own partial sum, we can declare a local sum variable then print it and to the total sum using a critical section, similar to what I have done for the dot product exercise.

3.3.

The problem here is in the barriers, what happens is that the threads that takes the first two sections, will wait in the barrier in the `print_results()` funciton, while the other thread will skip the section because of the *nowait* clause and will stop at the last barrier of main.

To fix the deadlock we can just remove the barrier in the `print_results()` function, and then we would also need to use two different shared variable c, one for the sum and one for the product, so that the sum in the c variable is correct.

3.4.

The problem here is the stack size limit. The default stack size for each thread is 4 MB [1], so when we try to allocate a copy of the array a for each of the threads we exceed that limit. This is because the size of a is $1048 * 1048 * 8\text{bytes} \approx 8,39\text{MB}$.

To fix this problem we can set the environment variable `OMP_STACKSIZE` to a higher value, like 10 MB for example. This way each thread will have enough stack space to allocate it's own copy of a.

3.5.

The problem here is the way the locks are used. In this way there is a high risk of deadlock, If thread 1 acquires lockA and thread 2 acquires lockB, then both threads will wait indefinitely for the other lock to be released, causing a deadlock.

The solution to this problem is to always acquire the locks in the same order. For example, we can always acquire lockA first and then lockB. This way, if thread 1 acquires lockA, thread 2 will wait for lockA to be released before it can acquire lockB, preventing the deadlock.

4. Parallel histogram calculation using OpenMP *(15 Points)*

4.1.

To parallelize the histogram calculation i choose to use the reduction clause, to calculte partial histograms for each thread, and then combine them at the end of the parallel region. I chose to do it for two main reasons:

- The first one, is because of the risk of race conditions, it might happen that two `vec[i]` hold the same value, and so two threads might try to update the same bin at the same time.
- The second one, is because of false sharing, if two threads are updating different bins that are close to each other in memory, it might happen that they are in the same cache line, and so the cache coherence protocol will cause unnecessary cache invalidations and performance degradation. To confirm this, I tried to not use the reduction clause and just update the global histogram directly (not caring about the race conditions), and as expected, I noticed a significant performance drop as the number of threads increased.

The plot for the strong scaling analysis is shown below:

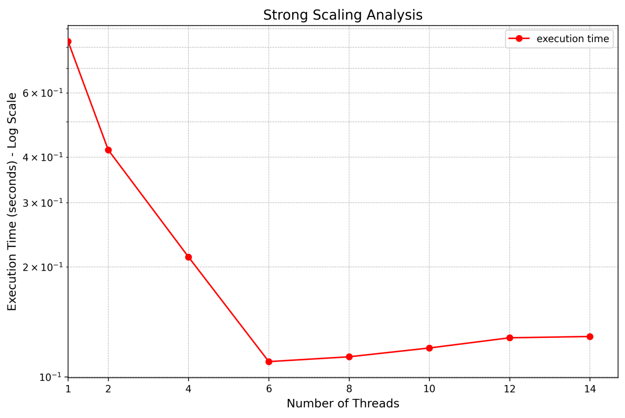

Figure 5: Strong scaling analysis for the histogram calculation

As for the other exercises, we can see that the is beneficial to increase the number of threads up to around 8, after that the performance improvement is minimal, or as in this case, negative.

5. Parallel loop dependencies with OpenMP *(15 Points)*

To parallelize this code we need to make sure that there are no dependencies between the iterations of the loop. In this case, we can see that each iteration of the loop depends on the previous one (calculation of Sn), but we can also notice how we can modify the code to make each iteration independent. Specifically, Sn is just itself multiplied for the up to the power of n, so we can calculate each opt[n] independently. The following code does this:

```
for (n = 0; n <= N; ++n) {
    opt[n] = Sn * pow(up, n);
  }
  Sn = opt[N] * up;
```

Listing 1: Parallelized loop dependencies with OpenMP

However the problem with this code is that the pow function is quite expensive, and so we do not get a significant performance increase compared to the sequential version. In this case, I tested with both 8 and 20 threads, and the results are shown in the table below:

Number of threads	Execution time (s)	Speedup
1	6.723246	1
8	14.672338	0.46
20	5.855687	1.15

Table 1: Execution time and speedup for the parallelized loop dependencies code

We can see that with 8 threads the performance is worse than the sequential version, while with 20 threads we get just a slight speedup.

6. Quality of the Report *(15 Points)*

References

[1] Stack Overflow, *Segmentation fault happening in OpenMP code*, `https://stackoverflow.com/questions/13264274/why-segmentation-fault-is-happening-in-this-openmp-code` (accessed October 23, 2025)