Assignment 2

SPM course a.a. 23/24, University of Pisa

Student: Irene Testa

April 26, 2024

1 Adopted solutions

Two parallel versions of the algorithm were developed and tested. Subsequent sections describe their implementation details.

1.1 Concurrent accesses to a shared data structure

This solution involved annotating the provided sequential implementation without modifying the code. OpenMP tasks were employed to parallelize the loop over files and the tokenization of each line. To prevent race conditions, access to shared data was encapsulated within a `#pragma omp critical` block.

1.2 Dedicated data structure for each thread

This solution was designed to mitigate the bottleneck caused by concurrent access to the shared data structure. It achieves this by allocating a separate `std::unordered_map` for each thread, eliminating the need for locking mechanisms. To ensure correct synchronization, the increment of the global variable `total_words` was annotated with `#pragma omp atomic`. As with the previous approach, OpenMP tasks were used to parallelize the loop over files and the tokenization of each line. The reduction of the data structures into a single one was not parallelized, as it already exhibited low computational times (see Figures 3 and 4), and the overhead of parallelization would likely have outweighed the benefits.

2 Performance evaluation

Metric assessment comprised measuring the execution times across 10 runs and calculating their median value. This median time was then utilized to compute performance metrics. The decision to use the median time instead of the mean was made due to the presence of outliers (see Figures 1 and 11).

3 Correctness verification

To verify the correctness of the code, the output of the parallel implementations was compared with the output of the sequential implementation for each test run (`topk` was always set to 5). All executions produced the expected output (refer to the files `./results/{machine_name}_error_log_critical.csv` and `./results/{machine_name}_error_log_maps.csv` and to their loading in the Python notebook `plots.ipynb`).

4 Results

Figures 1 to 3 and 5 to 10 report the computed performance metrics obtained from running the program on the machine ”spmcluster.unipi.it”, while Figures 4 and 11 to 18 illustrate the performance metrics from running the program on the machine ”spmnuma.unipi.it”. In the figure legends, the solution outlined in Section 1.1 is denoted as ”critical”, whereas the approach detailed in Section 1.2 is marked as ”maps”. Results exhibit consistency across both machines. Computing times on ”spmcluster.unipi.it” are generally lower than on ”spmnuma.unipi.it”.

Scalability and speedup are comparable, as the execution time of the sequential implementation is quite close to that of the parallel implementation when run on a single thread (see Figures 2 and 12). The implementation with separate data structures outperforms the one that uses a single, concurrently accessed data structure.

When `extraworkXline` is set to 0 or 1000, both the parallel implementations exhibit poor scalability. In the first case, with 4 or more threads, the parallel implementation's execution times are higher than those of the sequential implementation. In the second case, this occurs when using 12 or more threads on ”spmcluster.unipi.it” and more than 8 threads on ”spmnuma.unipi.it”. It's only when `extraworkXline` is set to 10000 that we start seeing benefits from multiple threads. However, the speedup is sublinear, and for the implementation with separate data structures, it begins to decrease when exceeding 24 threads on ”spmcluster.unipi.it” and after 20 threads on ”spmnuma.unipi.it”. With this implementation, the maximum speedup on ”spmcluster.unipi.it” is 9.41 (with 24 threads), while on ”spmnuma.unipi.it” it is 6.45 (with 20 threads). These poor results arise from the memory bound nature of the problem: the overhead involved in implementing parallel solutions often outweighs the benefits.

5 How to compile and execute the code

The parallel implementations of the program are located in the files `Word-Count-maps.cpp` and `Word-Count-critical.cpp`. Compilation of the code can be achieved by executing the command `make`. The makefile's phony target `debug` compiles the code with the `DEBUG` macro defined, enabling the printing of debug information during execution. Conversely, the `undebug` target recompiles the code without defining the `DEBUG` macro. When `DEBUG` is defined, threads output the line they are tokenizing, the filename they are processing, and the elapsed time since the clock's epoch to the standard output. The tests conducted to assess algorithm's performance and correctness can be consulted in the bash script `tests.sh`. The results of these tests are stored in the directory `results`, while figures are generated by the Python notebook `plots.ipynb`.

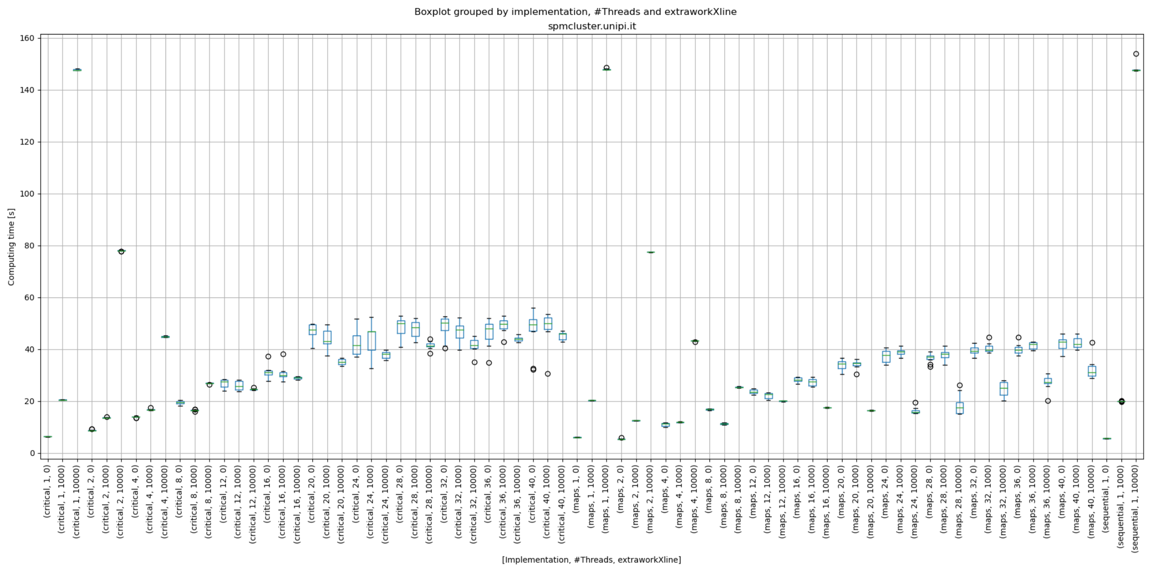


Figure 1: Execution time distributions on ”spmcluster.unipi.it”.

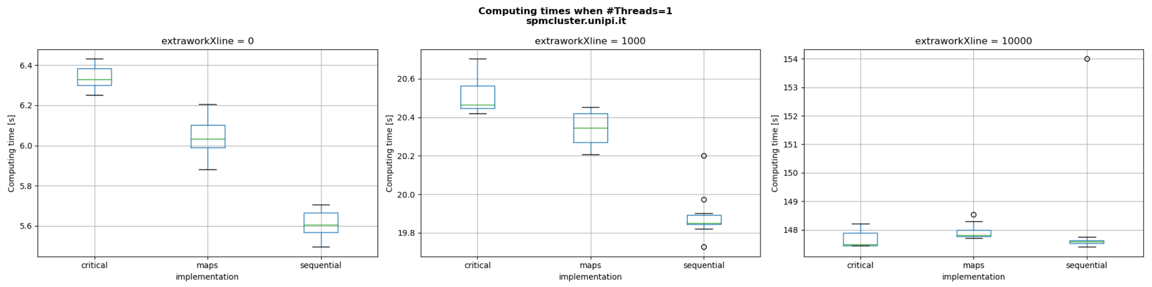
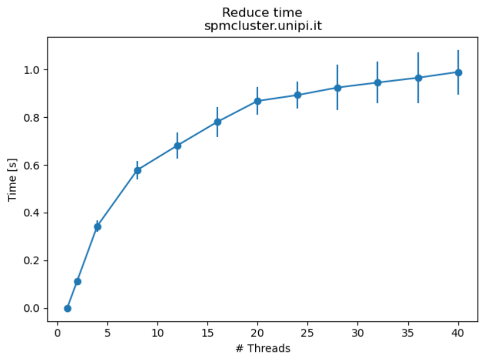


Figure 2: Execution time distributions for single-threaded program runs on ”spmcluster.unipi.it”.



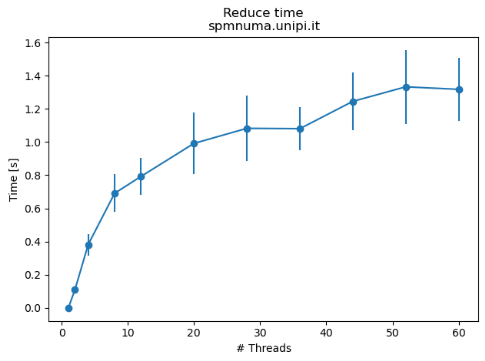


Figure 3: Mean reduce time and standard deviation on ”spmcluster.unipi.it” for the implementation using separate data structures.

Figure 4: Mean reduce time and standard deviation on ”spmnuma.unipi.it” for the implementation using separate data structures.

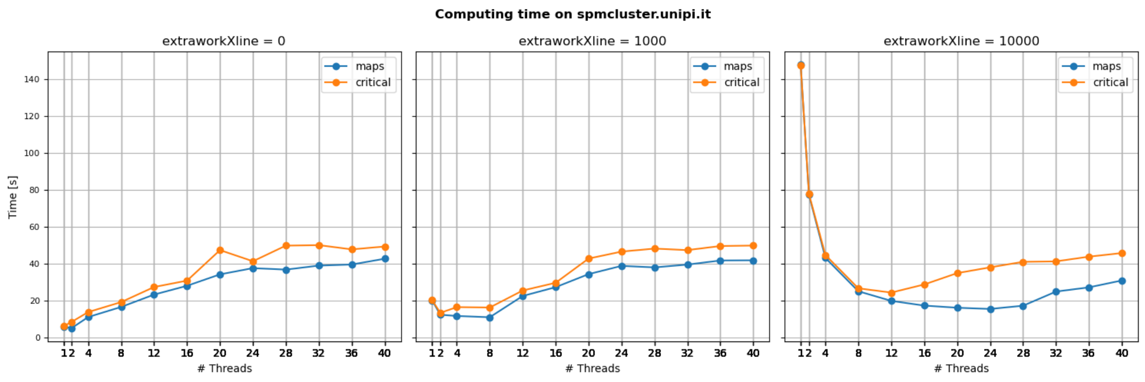


Figure 5: Median execution time on ”spmcluster.unipi.it”.

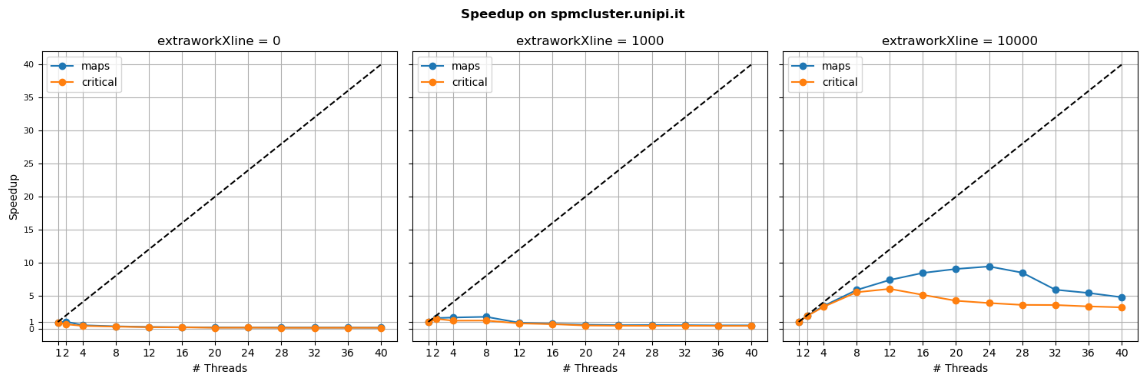


Figure 6: Speedup on ”spmcluster.unipi.it”.

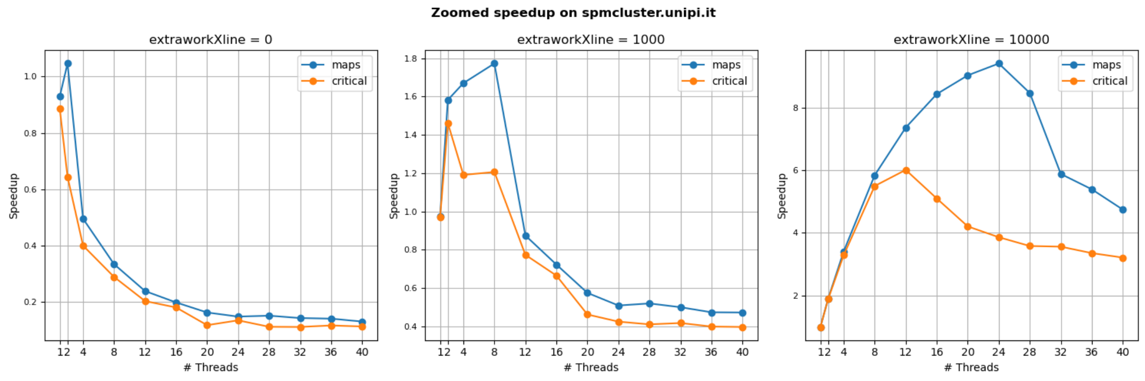


Figure 7: Zoomed speedup on ”spmcluster.unipi.it”.

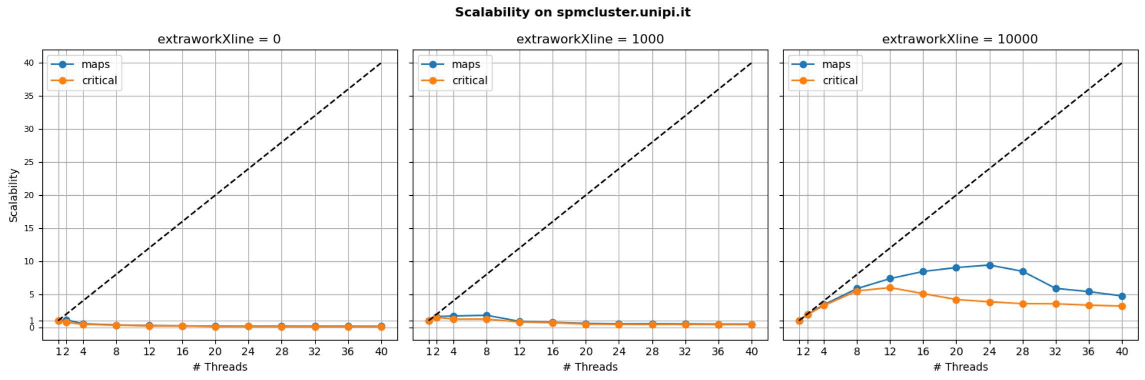


Figure 8: Scalability on ”spmcluster.unipi.it”.

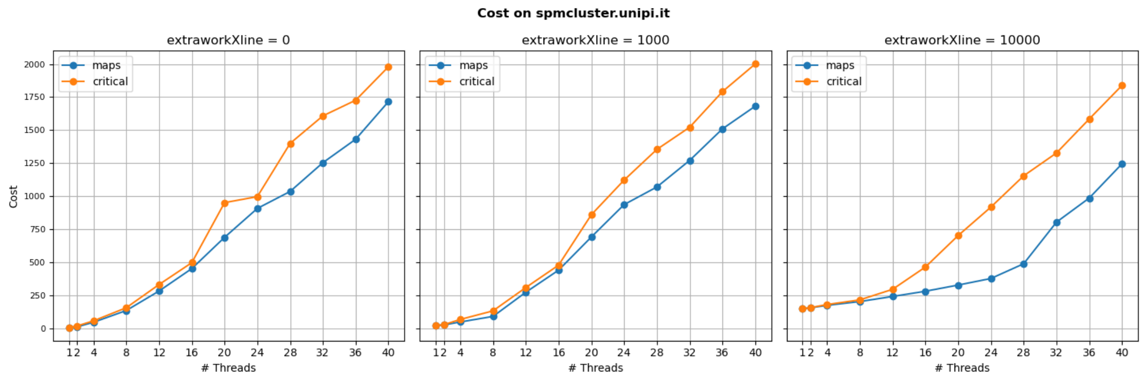


Figure 9: Execution cost on ”spmcluster.unipi.it”.

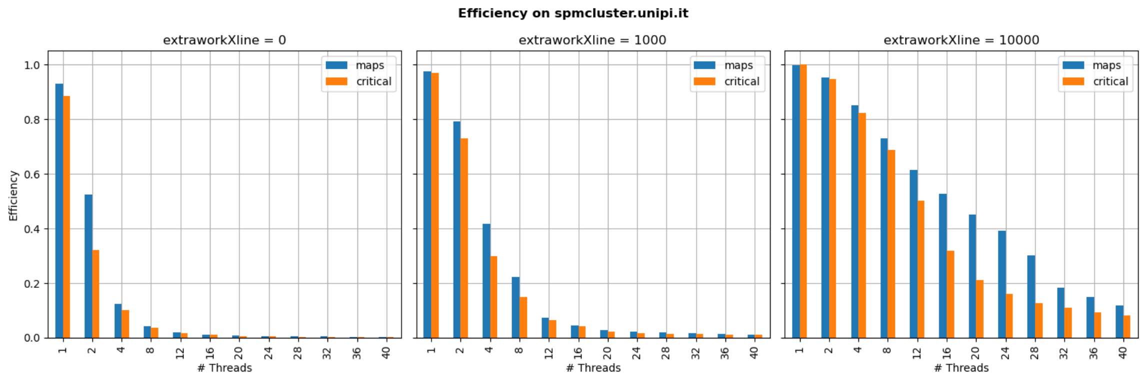


Figure 10: Efficiency on "spmcluster.unipi.it".

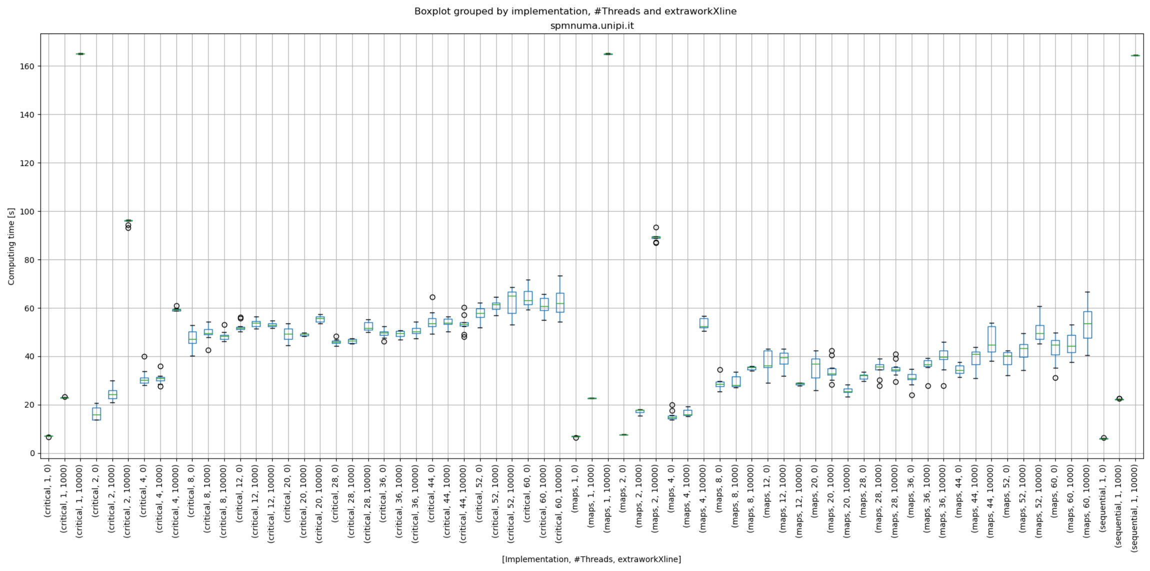


Figure 11: Execution time distributions on "spmnuma.unipi.it".

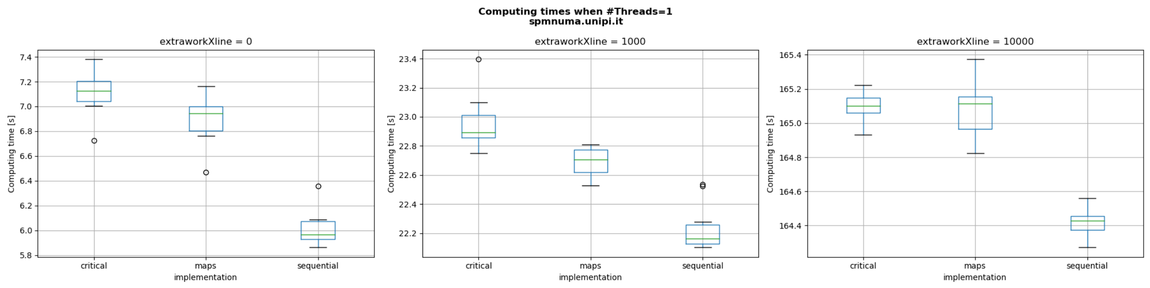


Figure 12: Execution time distributions for single-threaded program runs on "spmnuma.unipi.it".

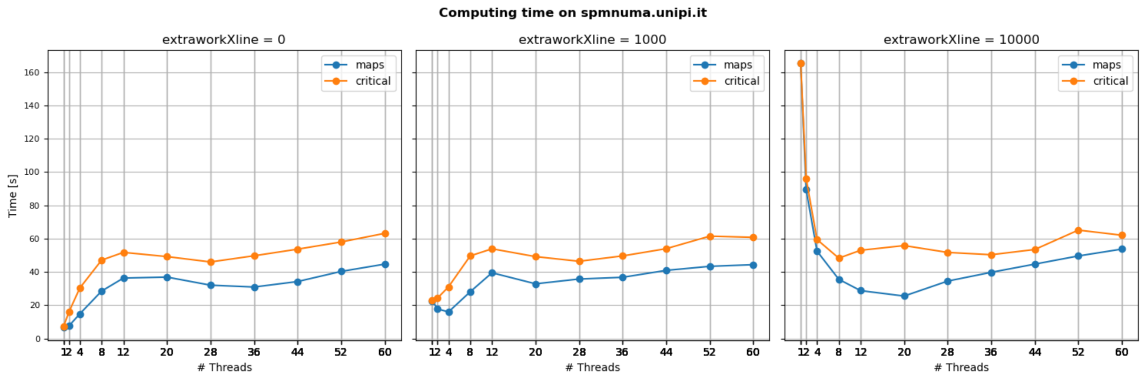


Figure 13: Median execution time on ”spmnuma.unipi.it”.

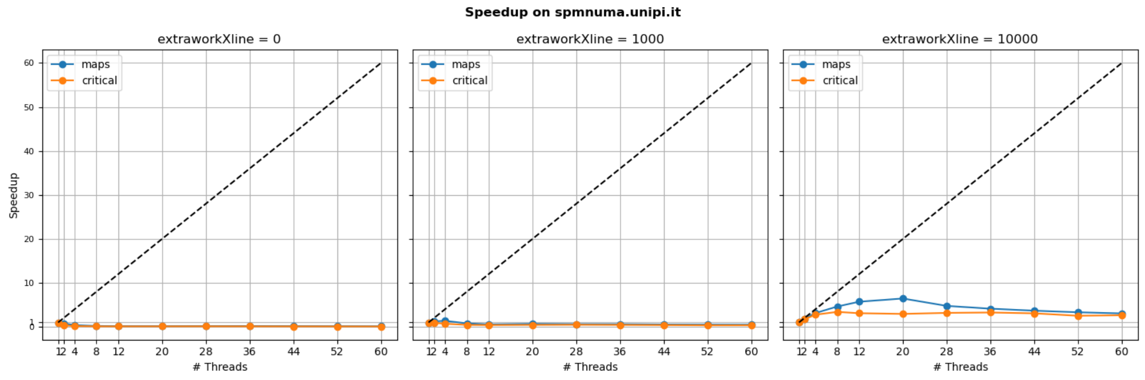


Figure 14: Speedup on ”spmnuma.unipi.it”.

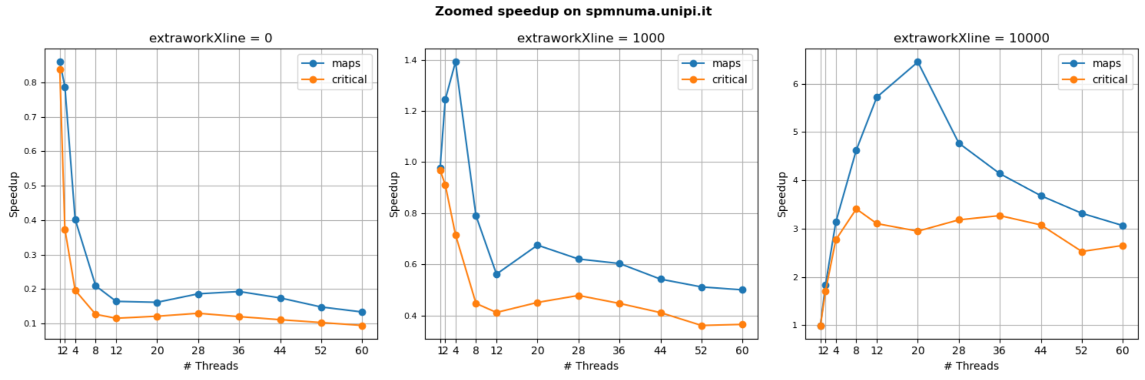


Figure 15: Zoomed speedup on ”spmnuma.unipi.it”.

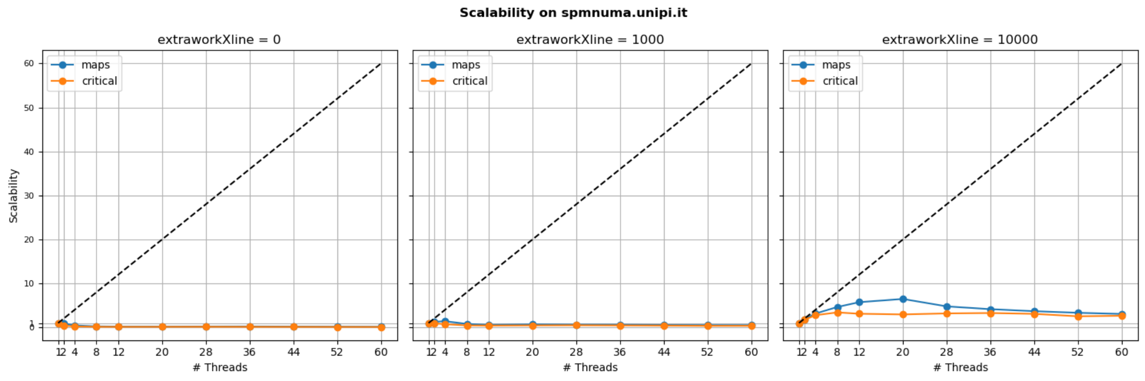


Figure 16: Scalability on ”spmnuma.unipi.it”.

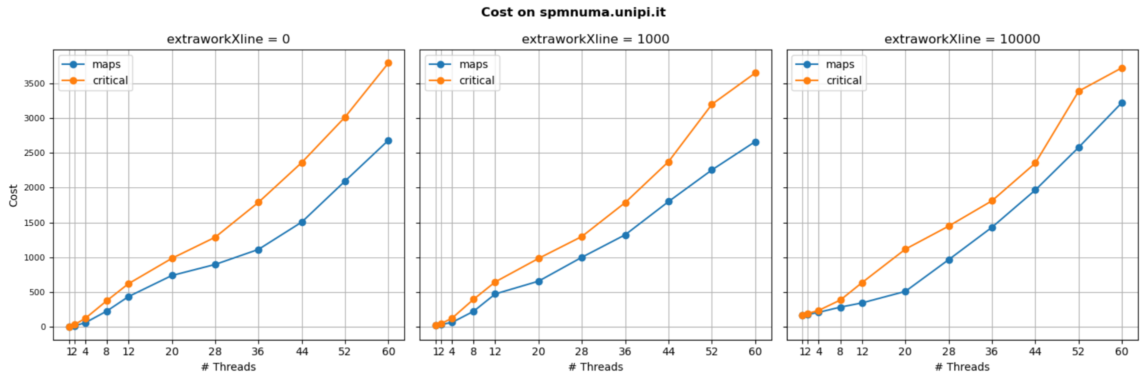


Figure 17: Execution cost on ”spmnuma.unipi.it”.

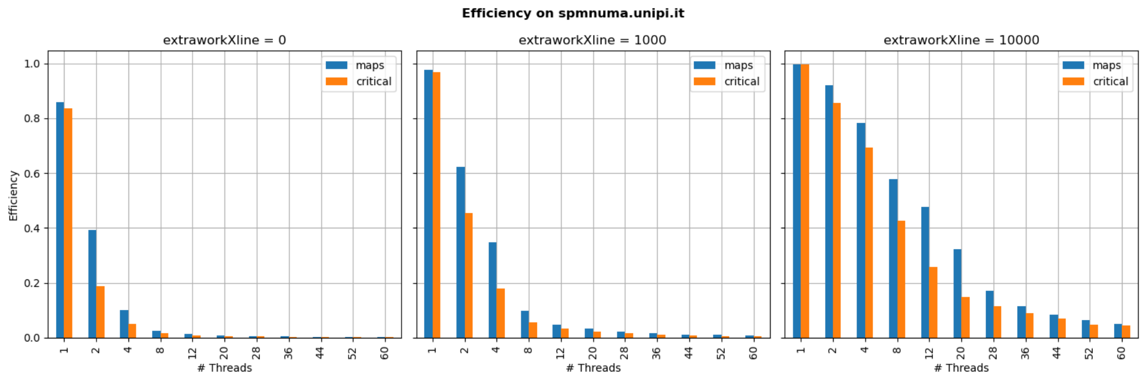


Figure 18: Efficiency on ”spmnuma.unipi.it”.