Assignment 1

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

Student: Irene Testa

April 4, 2024

1 Adopted solutions

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

1.1 Static distribution of tasks

The diagonal elements are evenly divided among threads and distributed cyclically to leverage memory spatial locality (each thread is assigned rows differing #Threads from each other). Alternative distributions (e.g. block or block cyclic) were not explored as the resulting memory access pattern does no longer preserve spatial locality.

Threads synchronize using an instance of the `std::barrier` class, initialized with a count equal to the number of threads. For each diagonal, threads process all assigned elements before invoking `std::barrier::arrive_and_wait`. When there are fewer elements in the diagonal than threads, the thread with no assigned elements invokes `std::barrier::arrive_and_drop` and returns.

1.2 Dynamic distribution of tasks

Synchronization is achieved by utilizing the thread pool implementation provided in the text book (`ThreadPool.hpp`), along with an instance of the `std::barrier` class initialized with an expected count equivalent to the number of threads in the pool.

The main thread iterates through the elements of each diagonal, dispatching tasks to the pool. If the number of elements in the current diagonal equals or exceeds the number of threads in the pool (T), only the last T submitted tasks will execute the method `std::barrier::arrive_and_wait`. Conversely, if the number of elements in the diagonal is less than the number of threads, a task executing the `std::barrier::arrive_and_wait` instruction is submitted to the pool for each excess thread.

2 Performance evaluation

Times were measured without considering the initialization of the matrix. For each configuration of the problem, metric assessment comprised measuring the execution times across 10 runs. The minimum and maximum execution times were excluded, and the average time was computed from the remaining eight values. This average time was then utilized to calculate performance metrics. In assessing weak scaling, the problem size was adjusted by augmenting both `min` and `max` parameters. The parameter `N`, which also dictates the problem size, was kept fixed for a more fair comparison (it involves the generation of additional random numbers, potentially changing the "structure" of the problem). The sequential algorithm wasn't executed; instead, expected sequential times were employed for computing performance metrics (when scaling the size of the problem, running the actual sequential algorithm

requires hours). Please, note that the number of threads in the Figures represents the number of threads spawned by the main thread (the actual number of threads is actually T+1).

3 Results

Figures 1 to 8 report the computed performance metrics obtained from running the program on the machine "spmcluster.unipi.it", while Figures 9 to 16 illustrate the performance metrics from running the program on the machine "spmnuma.unipi.it". Results exhibit consistency across both machines. Scalability and speedup remain similar when the problem size is fixed, suggesting that the expected sequential time closely aligns with the time required to execute the parallel program with a single thread. Under default parameters (`N`=512, `min`=0, `max`=1000), the parallel algorithm employing dynamic scheduling outperforms the algorithm using static scheduling. As the problem size scales with the number of threads, both solutions demonstrate comparable speedup (the relative advantage of one solution over the other remains nearly constant). In the strong scalability setting, speedup exhibits sublinear behavior, whereas in the weak scalability setting, it approaches linear and nearly perfect linear scaling. Efficiency in the strong scalability setting drops between 0.6 and 0.7 with the maximum number of threads, while in the weak scalability setting, it remains around 0.9.

Figures 17 to 20 compare the two implementations across various scenarios. Figures 17 and 18 display the distribution of execution times for the algorithms under uniform workloads (i.e., `min`=`max`=0 and `min`=`max`=500) with the number of threads equals to the numbers of cores. Generally, as expected, the implementation employing dynamic scheduling exhibits a notable increase in execution times (due to overhead). This does not hold for the execution times of the implementation with static distribution on the machine "spmcluster.unipi.it" with `min`=`max`=500. But being the time distribution of this implementation highly skewed, suggests that those times were measured when the machine was overloaded, making them not significant. Conversely, Figures 19 and 20 depict the execution time distribution for the algorithms under different workload imbalances (i.e., `min`=0, `max`=100 and `min`=0, `max`=2000) with the number of threads equals to the numbers of cores. When `min`=0 and `max`=100, the timings of the implementation with dynamic scheduling are greater than those of the implementation with static scheduling. In this scenario, the overhead of dynamically scheduling tasks outweighs the benefits. Instead, for greater workload imbalances, i.e., when `min`=0 and `max`=2000, dynamic scheduling proves advantageous.

4 How to compile and execute the code

The code implementing the algorithms is located in the file `wavefront.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 each operation they execute to the standard output, along with the elapsed time from the start of the program, allowing to verify the correctness of the algorithm. To view the available command line arguments, run `./wavefront -h`.
The tests conducted to assess algorithm performance can be consulted in the bash script `tests.sh`. The results of these tests are stored in `.csv` files within the `results` directory, while figures are generated by the Python notebook `plot.ipynb`.

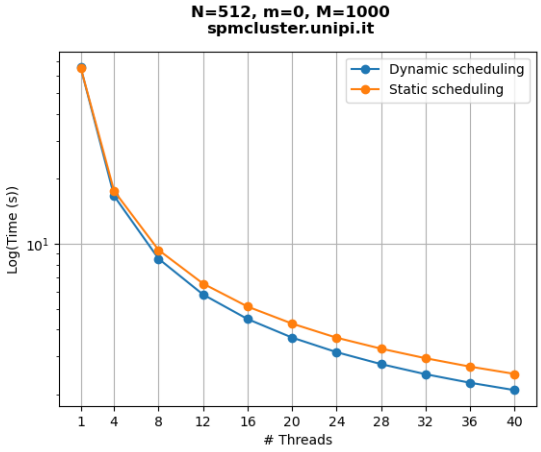

Figure 1: Execution time in seconds on "spmcluster.unipi.it" (strong scalability).

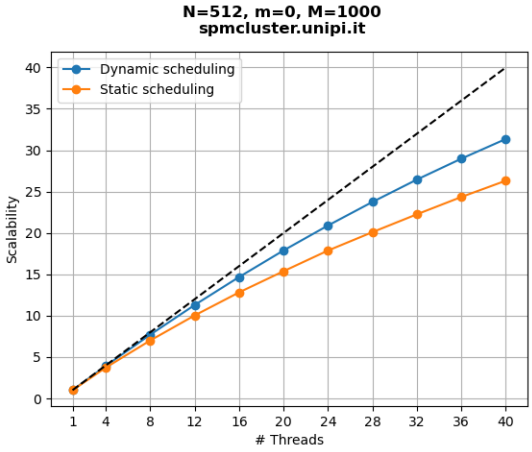

Figure 2: Scalability on "spmcluster.unipi.it" (strong scalability).

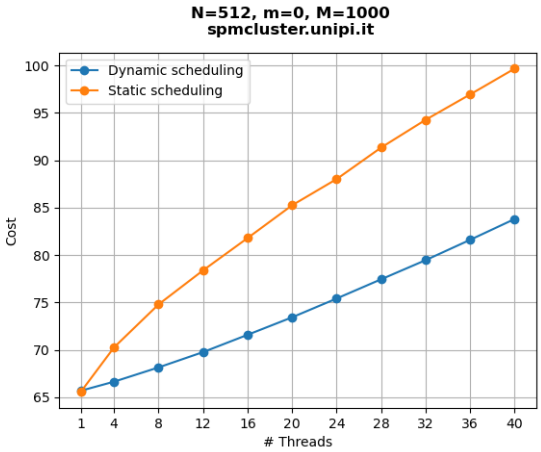

Figure 3: Execution cost on "spmcluster.unipi.it" (strong scalability).

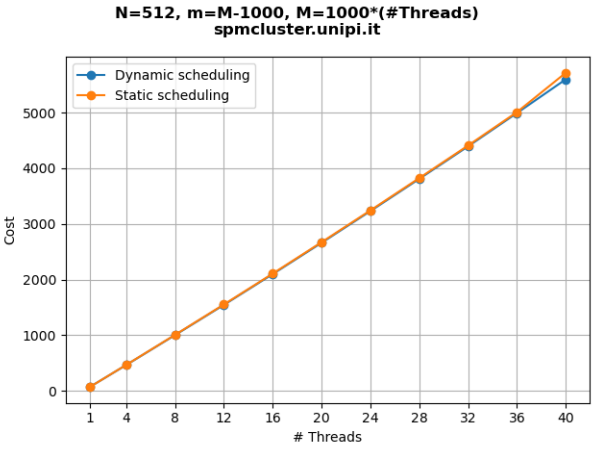

Figure 4: Execution cost on "spmcluster.unipi.it" (weak scalability).

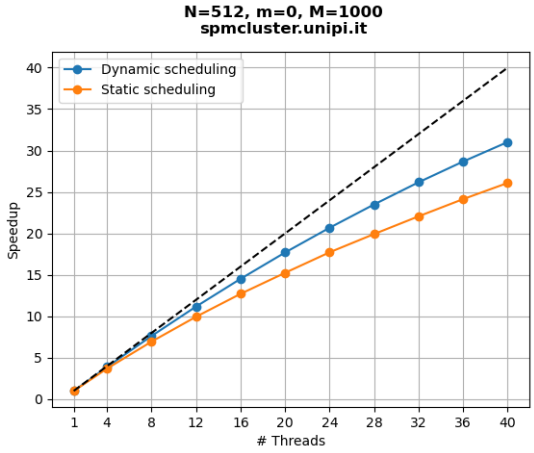

Figure 5: Speedup on "spmcluster.unipi.it" (strong scalability).

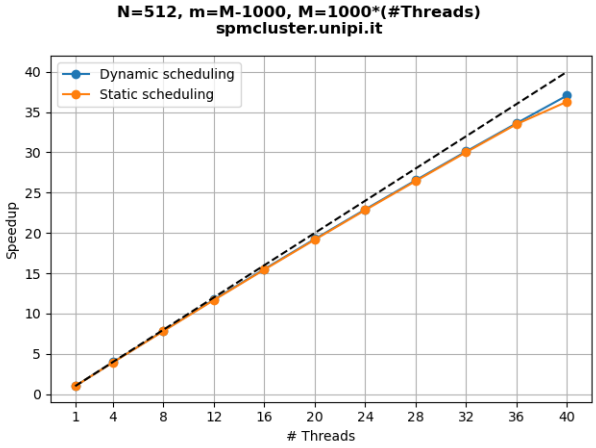

Figure 6: Speedup on "spmcluster.unipi.it" (weak scalability) .

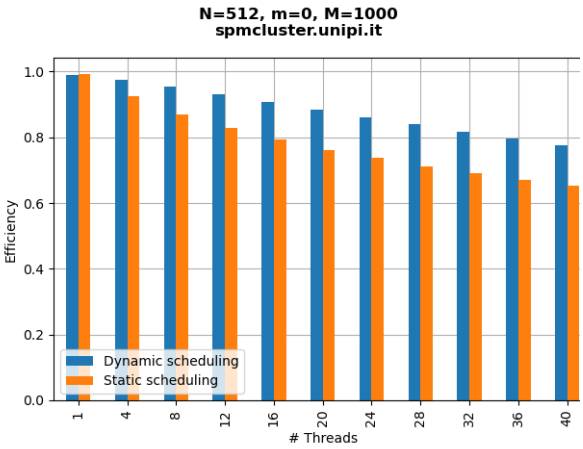

Figure 7: Efficiency on "spmcluster.unipi.it" (strong scalability). Please note that when the number of threads equals 1, efficiency is computed by dividing the expected sequential time by the average time of the parallel implementation executed with a single thread.

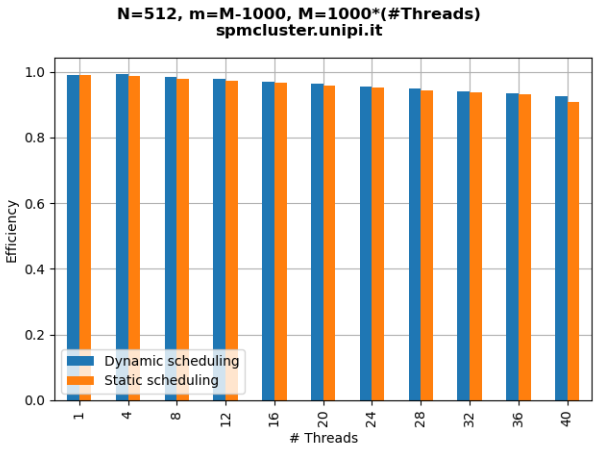

Figure 8: Efficiency on "spmcluster.unipi.it" (weak scalability).

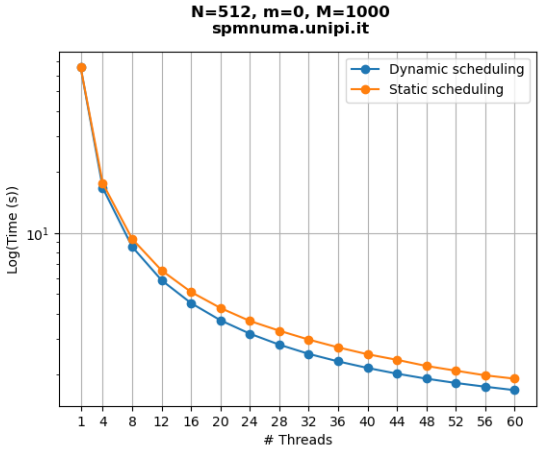

Figure 9: Execution time in seconds on "spmnuma.unipi.it" (strong scalabilty).

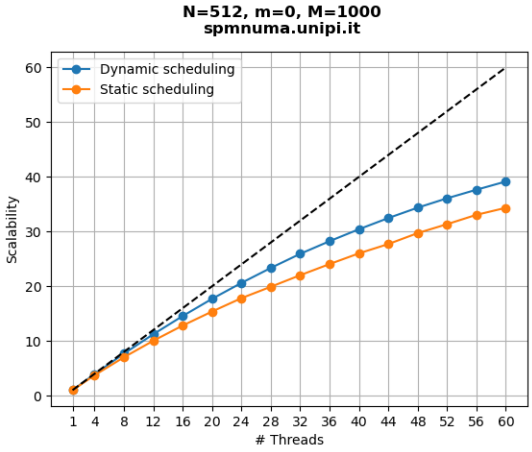

Figure 10: Scalability on "spmnuma.unipi.it" (strong scalability).

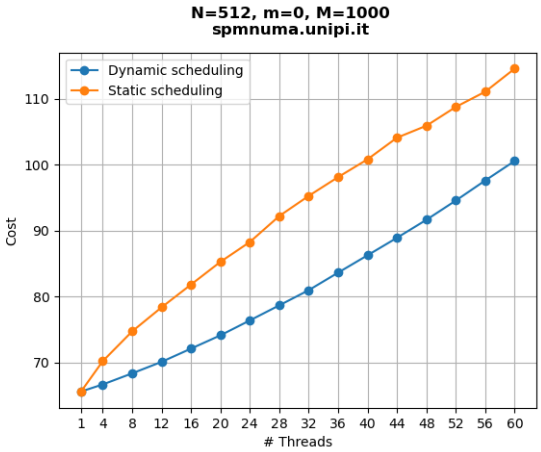

Figure 11: Execution cost on "spmnuma.unipi.it" (strong scalability).

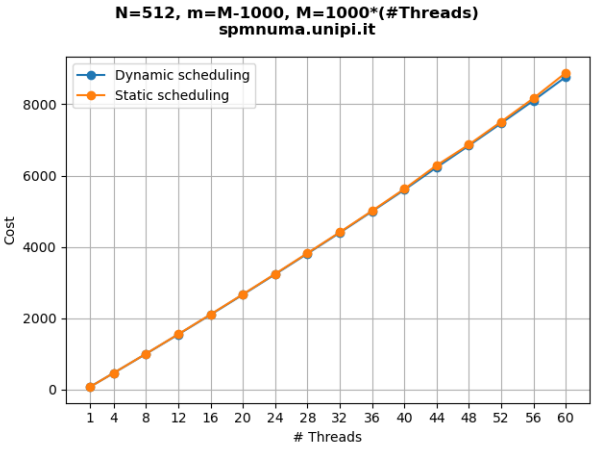

Figure 12: Execution cost on "spmnuma.unipi.it" (weak scalability).

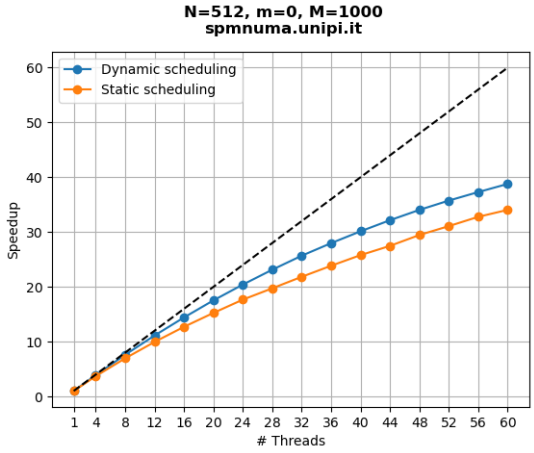

Figure 13: Speedup on "spmnuma.unipi.it" (strong scalability).

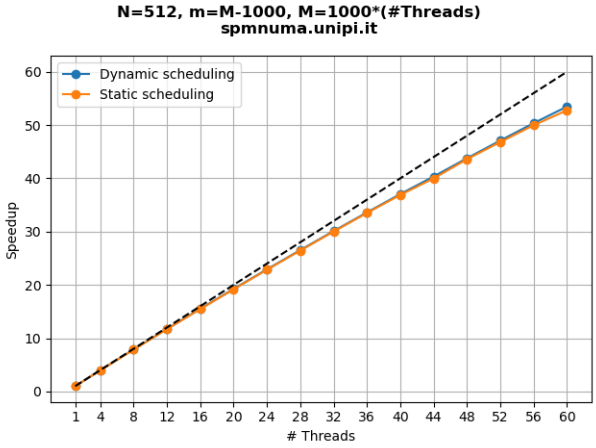

Figure 14: Speedup on "spmnuma.unipi.it" (weak scalability) .

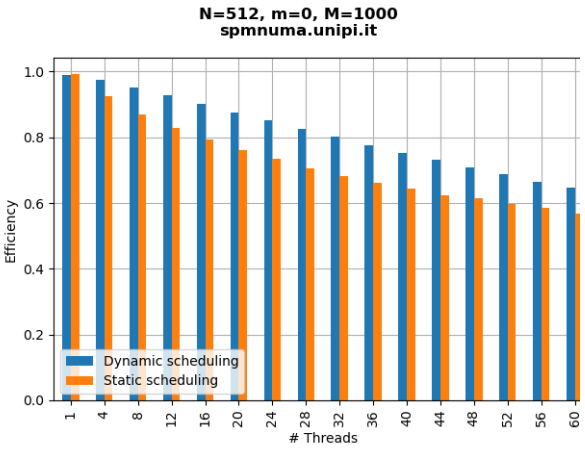

Figure 15: Efficiency on "spmnuma.unipi.it" (strong scalability). Please note that when the number of threads equals 1, efficiency is computed by dividing the expected sequential time by the average time of the parallel implementation executed with a single thread.

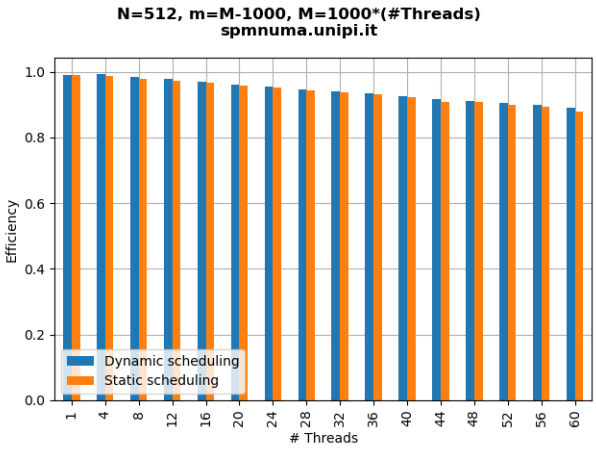

Figure 16: Efficiency on "spmnuma.unipi.it" (weak scalability).

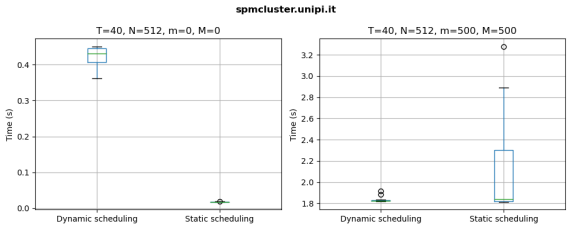

Figure 17: Comparison between the execution times of the two parallel implementations on "spmcluster.unipi.it" under uniform workloads (i.e., `min`=`max`=0 and `min`=`max`=500).

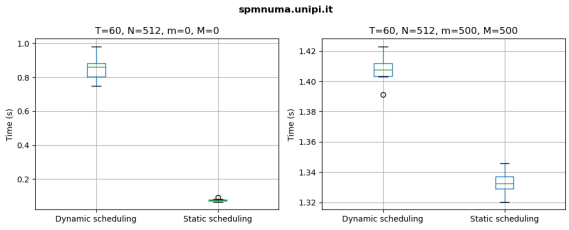

Figure 18: Comparison between the execution times of the two parallel implementations on "spmnuma.unipi.it" under uniform workloads (i.e., `min`=`max`=0 and `min`=`max`=500).

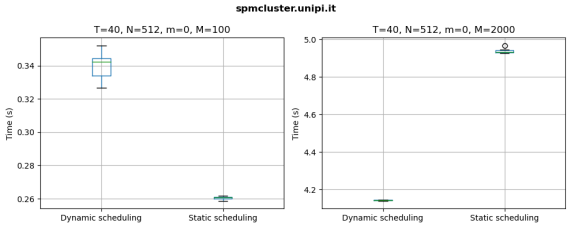

Figure 19: Comparison between the execution times of the two parallel implementations on "spmcluster.unipi.it" under different workload imbalances (i.e., `min`=0, `max`=100 and `min`=0, `max`=2000).

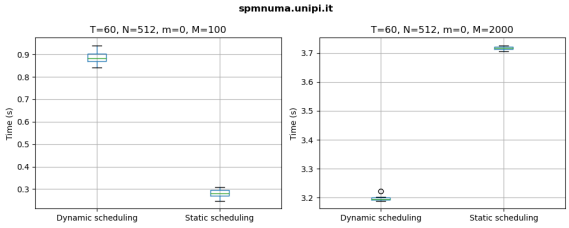

Figure 20: Comparison between the execution times of the two parallel implementations on "spmnuma.unipi.it" under different workload imbalances (i.e., `min`=0, `max`=100 and `min`=0, `max`=2000).