Assignment 4

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

Student: Irene Testa

May 31, 2024

1 Data Dependency Analysis

Iterations in the loop starting at line 78[1] are interdependent. Aside from the random generation of the keys, each iteration may update the map declared at line 72, which affects the operations performed in the subsequent iteration. Within a single iteration of this loop, the branches starting at lines 92 and 98 can be executed in parallel. The function `compute` (line 42) involves a matrix multiplication that can be parallelized. Iterations in the loop starting at line 120 are independent. Within each iteration, the execution of line 124 can be performed in parallel with the execution of line 125.

2 Parallel implementation

In the adopted parallel implementation, only the master process generates keys in the first loop and updates the map. This choice was made to mimic a scenario with a single stream of data 'local' to the master process and to avoid replicating the map across other processes. In the first loop, when both `map[key1]==SIZE` and `map[key2]==SIZE`, processes split into two groups to execute the function `compute`, which involves performing matrix multiplication.

Matrix multiplication was parallelized by distributing the rows of the first matrix among the processes, while broadcasting the second matrix. Since the matrices are always of equal size, broadcasting either matrix makes no difference in terms of communication timings. Because the matrices are deterministically initialized, having the master process distribute their entries is not strictly necessary; each process could fill the entries of its local matrices without exchanging messages with the master process. However, the implemented logic simulates a more realistic scenario where data is centralized and known only by the master process. The code implementing matrix multiplication is an adaptation of the implementation provided in the GitHub repository[2] of the book 'Parallel Programming: Concepts and Practice' by B. Schmidt et al. More sophisticated algorithms, such as Scalable Universal Matrix Multiplication (SUMMA), were not used because the provided implementations require the matrix dimensions to be multiples of the square root of the number of processes, an assumption that may not always be met.

After distributing the matrix entries, each process performs matrix multiplication on its assigned block, sums the resulting entries, and sends this partial sum to a master process for further summation. The matrix size threshold below which matrix multiplication is performed sequentially rather than in parallel can be specified as an additional command line argument.

The final two nested for-loops were parallelized by collapsing them and statically distributing iterations among processes. This requires the master process to broadcast the map to all other processes. Collapsing the loop maximizes parallelization, allowing to exploit all the available processes even when the number of processes exceeds the number of keys. In the second loop, the function `compute` is executed sequentially by each process. I chose to parallelize the iterations rather than the computations

[1] Lines are referenced as in the file `nkeys_original.cpp`.

[2] `https://github.com/JGU-HPC/parallelprogrammingbook/tree/master/chapter9/matrix_matrix_mult`

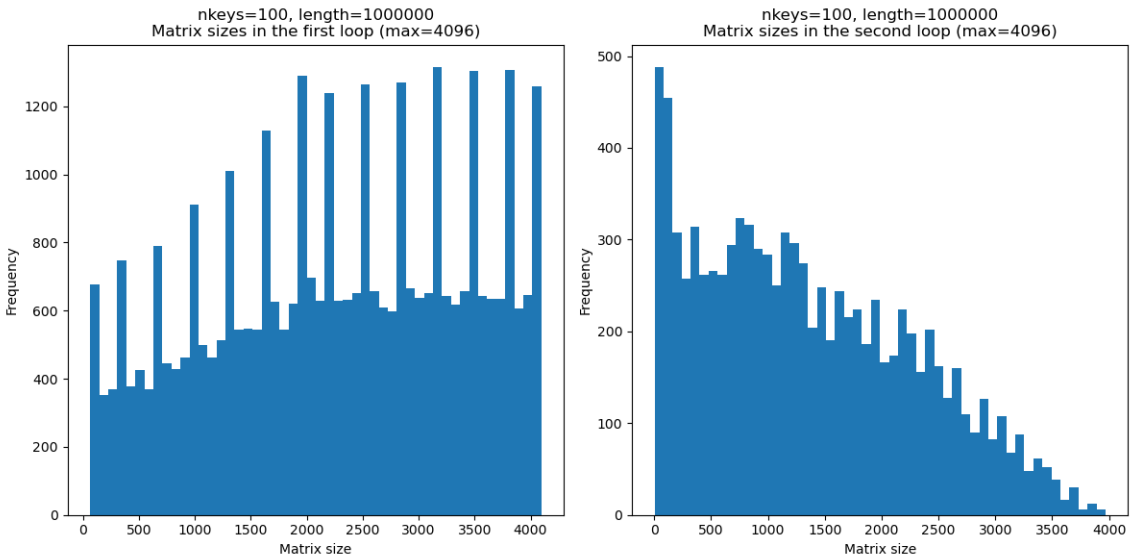

Figure 1: Matrix sizes in the first and second loop when `nkeys=100` and `length=1000000`.

within each iteration, as the smaller matrix sizes in this phase of the computation (see e.g., Figure 1) make it more efficient to focus processing resources this way.

3 Performance evaluation

Metric assessment comprised measuring the execution times across 10 runs on the internal nodes of the computer cluster "spmcluster.unipi.it" and calculating their median value. This median time was then utilized to compute performance metrics. The median was preferred over the mean because the execution times in different runs exhibit outliers (see Figures 2 and 3), and the median is less affected by them.

4 Correctness verification

To minimize errors caused by floating-point operations, all variables declared as `float` in the provided sequential implementation were changed to `double`. The results from this revised sequential implementation (located in the file `nkeys_seq.cpp`) were then used as a reference. To verify the correctness of the code, the output of the parallel implementation was compared with the output of the sequential implementation, and all the executions produced the expected results (refer to the files `seq_2_1000000_output.csv`, `par_2_1000000_output.csv` and `seq_100_1000000_output.csv`, `par_100_1000000_output.csv`, produced by the bash script `tests.sh`).

5 Results

Figures 2 to 8 report the performance metrics obtained from running the program on the internal nodes of the computer cluster "spmcluster.unipi.it". In the Figures, the parameter `threshold` represents the matrix size below which matrix multiplication is executed sequentially.

5.1 Strong Scalability Analysis

When running the program on 8 nodes, `nkeys=2` or `nkeys=100` and `length=1000000`, the execution times of the parallel implementation are generally higher than those of the sequential implementation, and the overhead introduced by using a higher number of processes degrades performance (see Figure 2). Instead, when running the program on a single node, the execution times of the parallel implementation decrease compared to those of the sequential implementation (see Figure 3). This indicates that network communication time represents a bottleneck.

Figures 4 to 6 show the performance metrics obtained when running the program on a single node. When `nkeys` is either 2 or 100, using more than 8 processes is not advantageous. Similarly, when `nkeys` is 500, using more than 16 processes is not beneficial. Among the three tested values for `nkeys`, the problem scales best when `nkeys=500`, achieving a maximum speedup between 4.5 and 5. It scales worst when `nkeys=100`, reaching a maximum speedup between 1.5 and 1.7.

On a single node, always parallelizing matrix multiplication consistently proves to be advantageous. When using `nkeys=2`, always performing matrix multiplication in parallel is almost equivalent to performing it only when the number of entries in the matrices is higher than 2048. This is because in the first loop, all matrices are sized 4096 (64×64), meaning that for both threshold values, matrix multiplication is always performed in parallel; while in the second loop, matrices are sized 256 (16×16), but only 2 iterations are performed.

5.2 Weak Scalability Analysis

To evaluate weak scaling, the problem size was increased proportionally with the number of processes by increasing the number of keys. The parameter `length` was kept fixed at 1000000 as it only determines the number of iterations in the first loop, where parallelization is limited due to interdependencies among iterations and to the fact that matrix sizes are bounded. Figures 7 and 8 report the performance metrics obtained in this setting.

Running the program on 8 nodes does not prove advantageous when the number of processes is 2, 4, 8 or 16 (the resulting speedup is lower than 1). With 32 processes, the achieved speedup is around 3 when matrix multiplication is performed sequentially. Therefore, in this latter case, efficiency is below 0.1. In this setting, parallelizing matrix multiplication does not prove to be advantageous.

On a single node, the speedup gradually increases with a shallow slope. Using 32 processes, the maximum speedup achieved is approximately 6, occurring when matrix multiplication is always performed in parallel.

6 Possible improvements

Potential improvements to the current solution include: (1) using OpenMP to parallelize the sub-matrix multiplication performed by each process; (2) implementing matrix multiplication by distributing the columns of the first matrix among processes when the number of processes exceeds the number of rows

to fully utilize the available computational resources; (3) employing the SUMMA algorithm for matrix multiplication when the matrix shapes meet its applicability conditions; (4) optimizing communication by having the master process send only the necessary map elements to the other processes for the second loop, rather than the entire map; and (5) parallelizing matrix multiplication also in the second loop by distributing the iterations among groups of processes.

7 How to compile and execute the code

The parallel implementation of the program is located in the file `nkeys_par.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 the execution of the sequential implementation. This option was used to produce Figure 1.

The tests conducted to assess algorithm's performance and correctness can be consulted in the bash script `tests.sh`. This script also invokes the `SLURM` scripts, specifically `slurm_script_1node.sh` and `slurm_script_8node.sh`, which were used to run the program on the cluster nodes. The results of the conducted tests are stored in the directory `results`, while figures are generated by the Python notebook `plots.ipynb`.

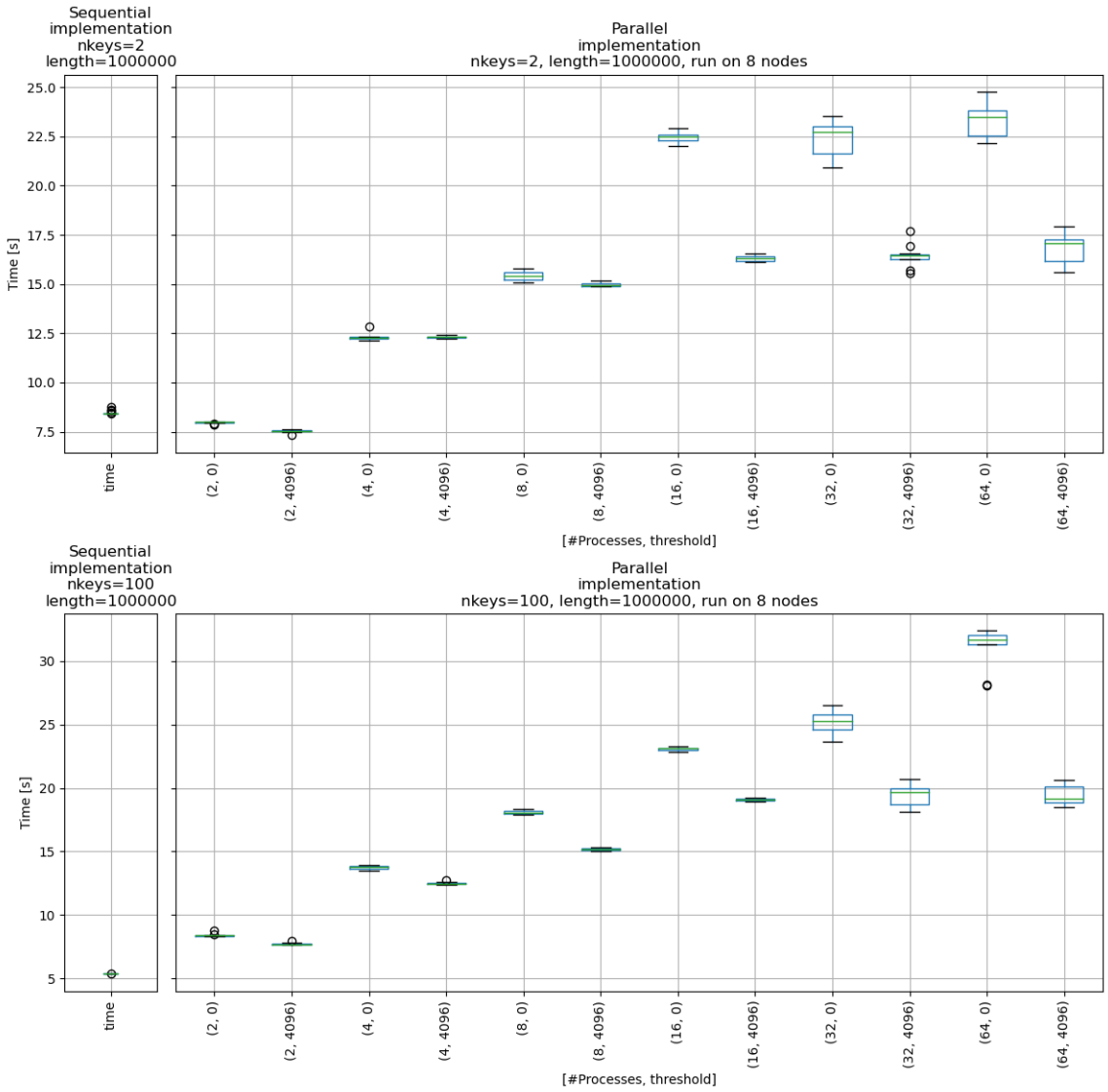

Figure 2: Distribution of execution times when running the program on 8 nodes.

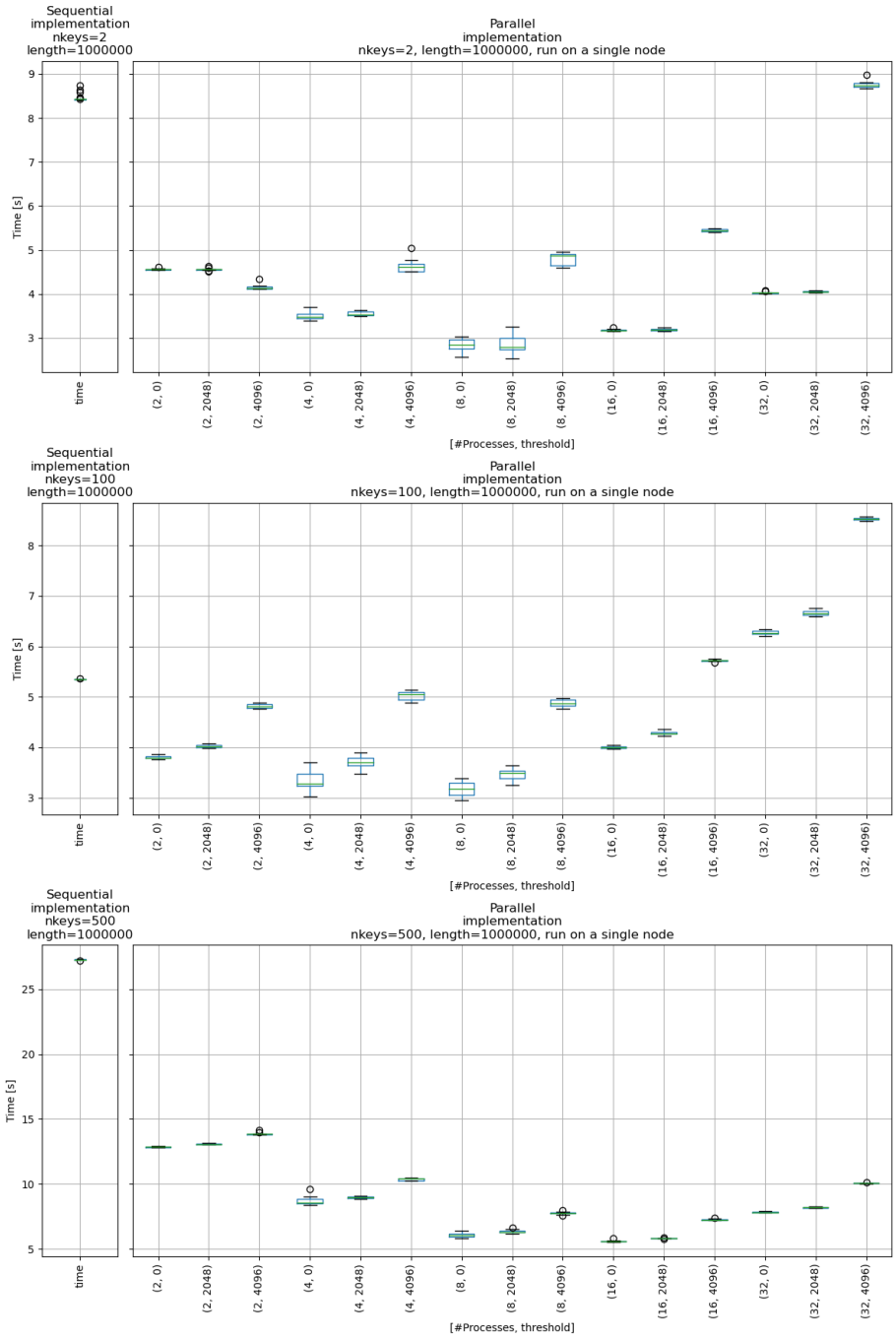

Figure 3: Distribution of execution times when running the program on a single node.

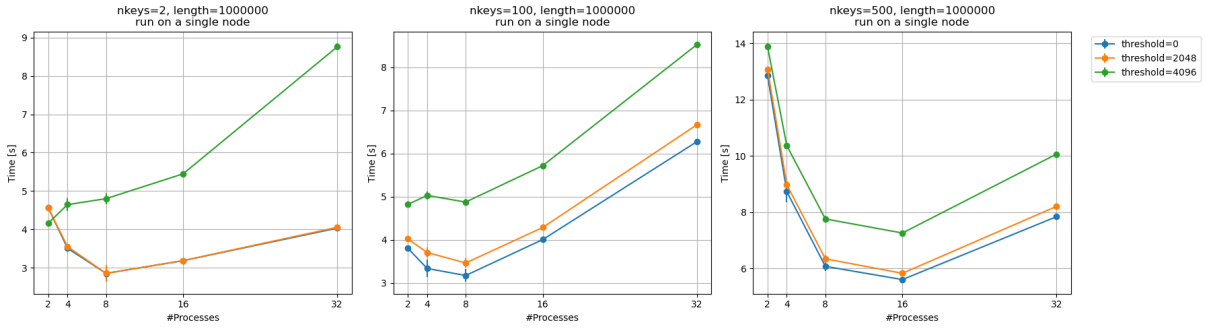

Figure 4: Mean execution time and standard deviation when running the program on a single node.

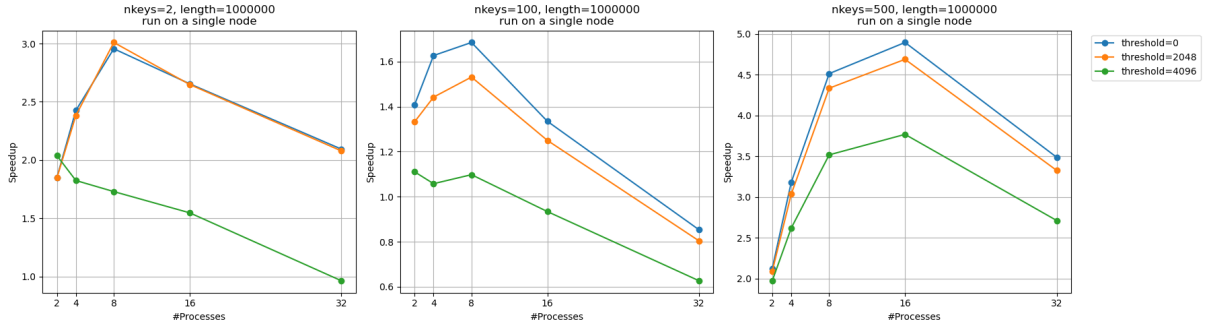

Figure 5: Speedup when running the program on a single node.

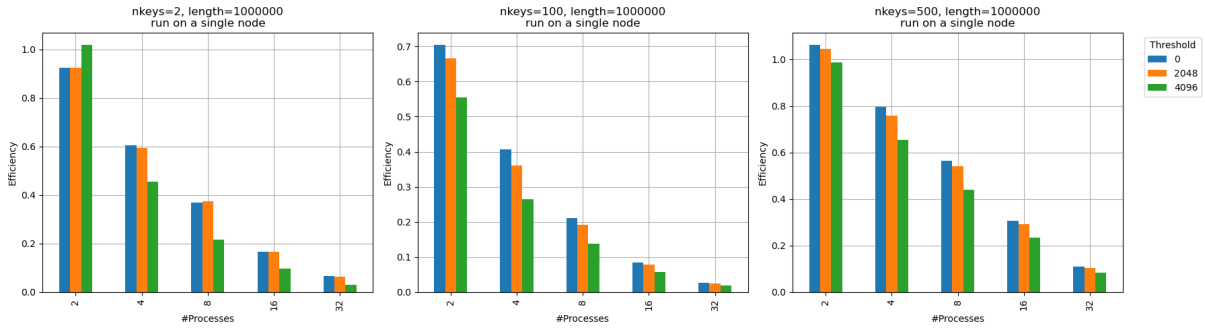

Figure 6: Efficiency when running the program on a single node.

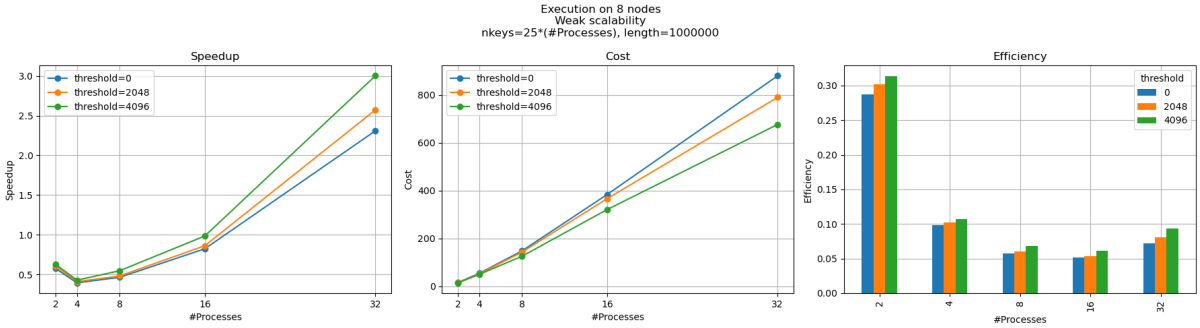

Figure 7: Performance metrics for weak scalability when running the program on 8 nodes.

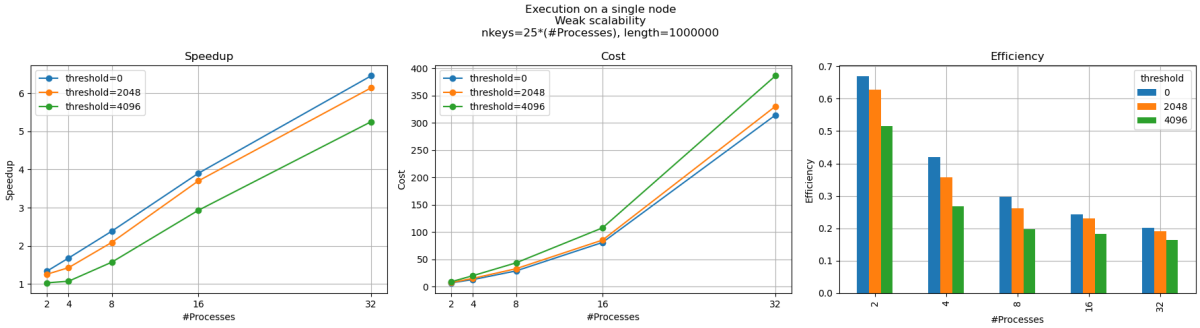

Figure 8: Performance metrics for weak scalability when running the program on a single node.