Assignment 3

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

Student: Irene Testa

May 31, 2024

1 Adopted solution

The adopted solution employs the FastFlow building block `ff_a2a`. Left workers read lines from files and send them, one at a time, to right workers, who tokenize the lines and count word occurrences storing them in a `std::unordered_map`. Each right worker accesses a separate map. To ensure correct synchronization, the global variable `total_words` is declared as `std::atomic<uint64_t>` and incremented using atomic operations. The reduction of these data structures into a single one was not parallelized because it already exhibited low computational times (see Figure 1), and the overhead of parallelization would likely have outweighed the benefits. The program takes as additional command line arguments the number of left and right workers and the value of the `ondemand` parameter in the `add_firstset` method of the class `ff_a2a`.

2 Performance evaluation

Metric assessment comprised measuring the execution times across 10 runs on the machine "spmnuma.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 Figure 2), and the median is less affected by them. Sorting time was not included in either the parallel or sequential execution times. Scalability (i.e. relative speedup) was not evaluated because the `ff_a2a` building block requires at least two nodes, and therefore, at least two threads.

3 Correctness verification

To verify the correctness of the code, the output of the parallel implementation 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 file `./results/error_log.csv` and to its loading in the Python notebook `plots.ipynb`).

4 Results

Figures 1 to 7 report the performance metrics obtained from running the program on the machine "spmnuma.unipi.it". Please note that the number of threads in the figures represents the number of FastFlow nodes, excluding the main thread. Results were obtained by running the program with default settings: non-blocking mode and unbounded buffers (buffers are sized to 1 when `ondemand=1`, regardless of the flag's value).

When `extraworkXline` is set to 0 or 1000, the parallel implementation exhibits poor scalability. It's only when `extraworkXline` is set to 10000 that the execution time keeps decreasing with the

number of threads. However, the speedup is sublinear, and it reaches a plateau at around 36 threads, achieving a value between 8 and 9.

As expected, on-demand scheduling performs similarly to non-on-demand scheduling, given the lengthy stream of tasks and their balanced nature. When `extraworkXline` is set to 10000, the on-demand policy performs better than the non-on-demand policy, likely because it promotes data sharing between right and left workers in low-level caches. Overall, we can conclude that, given the comparable performance of the two policies, on-demand scheduling can be effectively used for this specific problem in memory-bound scenarios without performance degradation.

When `extraworkXline` is 10000, using twice as many left workers as right workers is not beneficial. This is probably because the tokenization task (performed by right workers) takes longer than the task performed by left workers. Therefore, in this scenario, it is better to increase the parallelization of the tokenization task.

Using more than one left worker does not appear to be beneficial. This is probably due to the fact that left workers are statically assigned to files but file sizes are not balanced. The largest file appears towards the end of the file list, causing the left worker assigned to it to continue reading while other left workers already finished their tasks. To test this hypothesis, the program was ran with a total of 28 workers, using 1, 4, and 14 left workers, with `ondemand` set to 0 and `extraworkXline` set to 10000, on a list of 70 files, each 508 Kb (all copies of the file `/opt/SPMcode/A2/files/pg78.txt`). The squares in Figure 4 represent the speedup achieved by these configurations, relative to the execution time of the sequential implementation on the same list of files. It is evident that using 4 left workers significantly increases speedup compared to using one left worker. However, employing 14 left workers results in worse performance, likely because the degree of parallelization in the reading task is excessively high compared to that in the tokenization task.

When using one left worker, the cost of parallelization decreases when moving from one right worker to three right workers. This means that with three right workers, the computing time is less than half of the computing time when using only one right worker. Similarly, parallelization cost also decreases when using four left workers as the number of right workers increases from four to eight. The highest efficiency is achieved with one left worker and three right workers.

Figure 7 compares the execution times with and without default thread mapping (i.e., compiling the code with or without the `-DNO_DEFAULT_MAPPING` flag) when using 14 left workers, 46 right workers, `extraworkXline` set to 10000, and `ondemand` set to 0. In this specific case, not using default mapping reduces computing times. FastFlow's default mapping pins threads to sibling cores following a depth-first graph traversal. For a `ff_a2a` building block, this means visiting one left worker first, then all right workers, and finally the remaining left workers. Consequently, on the machine "spmnuma.unipi.it", most left workers are pinned to the second CPU, therefore caching benefits are only utilized on that CPU, when right workers access the file lines stored in cache by left workers. Without FastFlow's default mapping, the operating system dynamically moves threads across cores, allowing cache sharing on both CPUs and improving performance.

The program implemented with FastFlow achieves higher performance than the version built with OpenMP tasks, especially when `extraworkXline` is set to 10000. In this scenario, timing keeps reducing by increasing the number of threads. However, all tested configurations exhibit poor scalability due to the memory-bound nature of the problem.

5 How to compile and execute the code

The parallel implementation of the program is located in the file `Word-Count-par.cpp`. Compilation of the code can be achieved by executing the command `make`. Defining the environmental variables `NO_DEFAULT_MAPPING`, `BLOCKING_MODE` and `BOUNDED_BUFFER` enables the flags `-DNO_DEFAULT_MAPPING`, `-DBLOCKING_MODE` and `-DFF_BOUNDED_BUFFER` respectively. 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`.

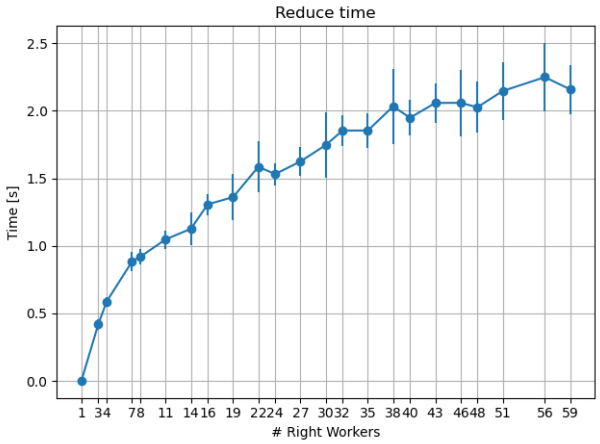

Figure 1: Mean time and standard deviation required to reduce the maps accessed by each thread into a single map, while varying the number of right workers (i.e. the number of maps to be reduced).

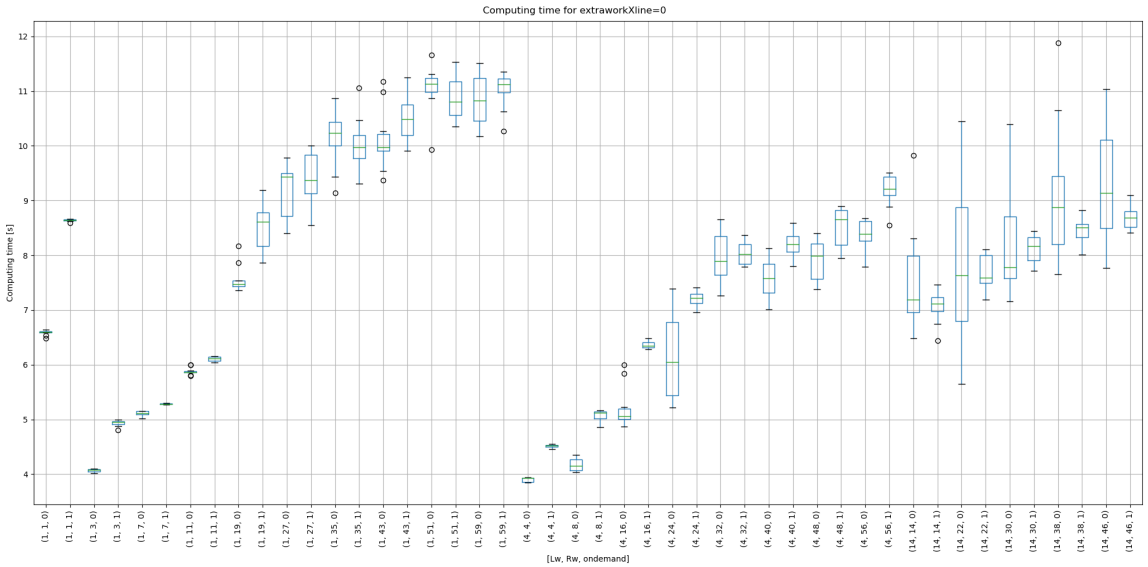

Figure 2: Execution time distributions when `extraworkXline` is set to 0.

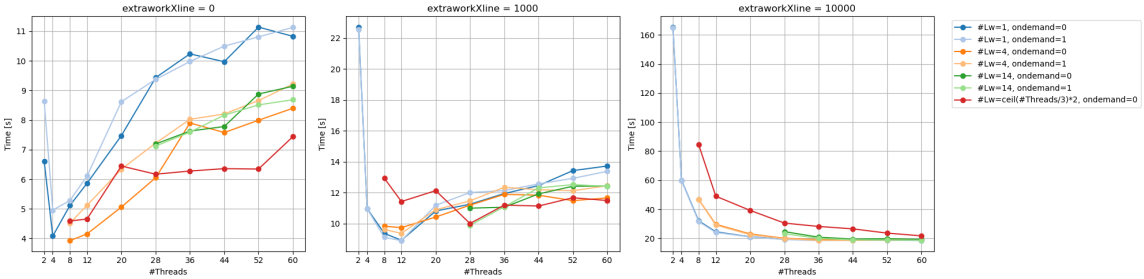

Figure 3: Median execution time with different values of `extraworkXline`, numbers of left workers (`#Lw`) and scheduling policies (`ondemand`). The x-axis indicates the number of threads, including right workers (`#Threads = #Lw + #Rw`). The red line shows the timing when using approximately twice as many left workers as right workers.

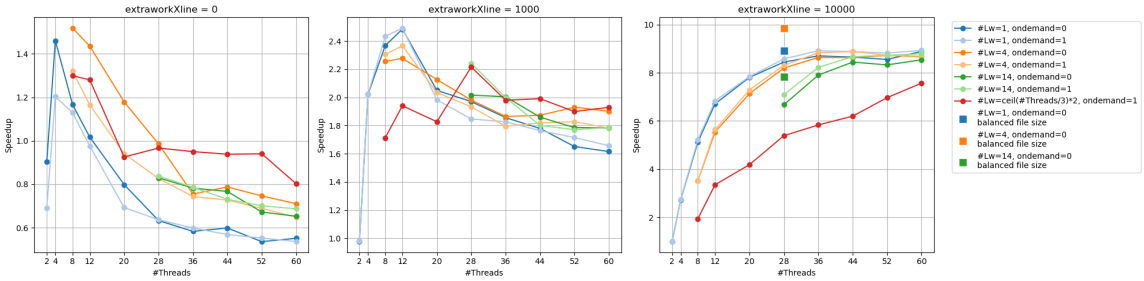

Figure 4: Speedup with different values of `extraworkXline`, numbers of left workers (`#Lw`) and scheduling policies (`ondemand`). The x-axis indicates the number of threads, including right workers (`#Threads = #Lw + #Rw`). The red line shows the timing when using approximately twice as many left workers as right workers. The squares indicate the speedup when executing the program on a list of 70 files, 508 Kb each.

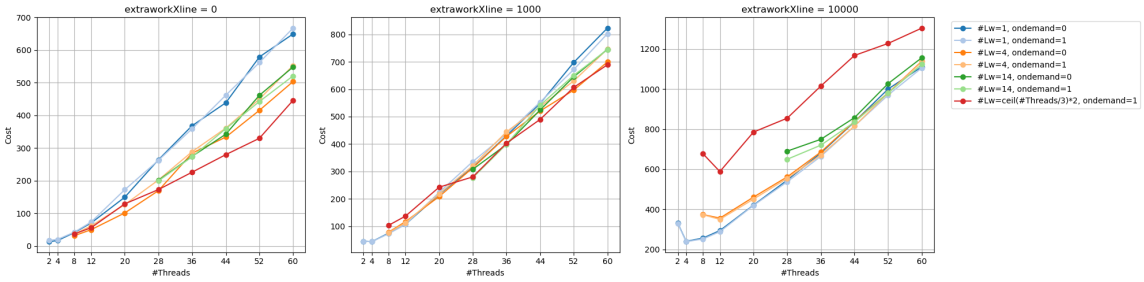

Figure 5: Cost of parallelization with different values of `extraworkXline`, numbers of left workers (`#Lw`) and scheduling policies (`ondemand`). The x-axis indicates the number of threads, including right workers (`#Threads = #Lw + #Rw`). The red line shows the timing when using approximately twice as many left workers as right workers.

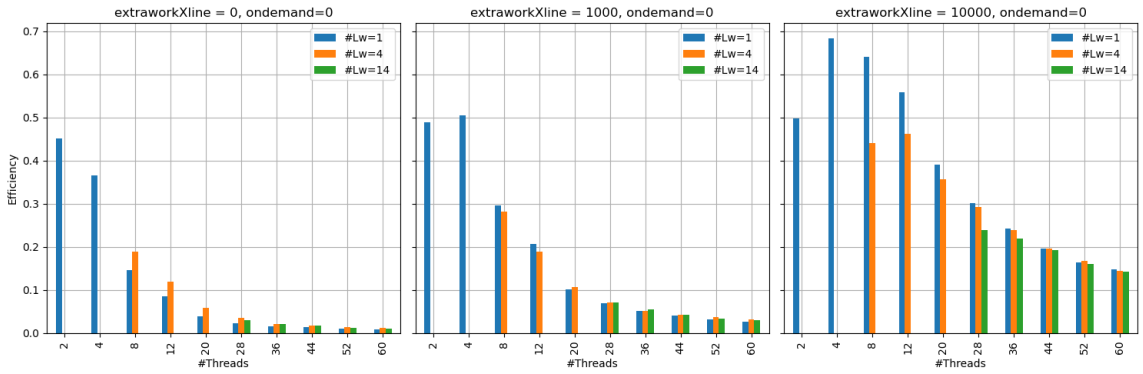

Figure 6: Efficiency with different values of `extraworkXline` and numbers of left workers (`#Lw`). The x-axis indicates the number of threads, including right workers (`#Threads = #Lw + #Rw`).

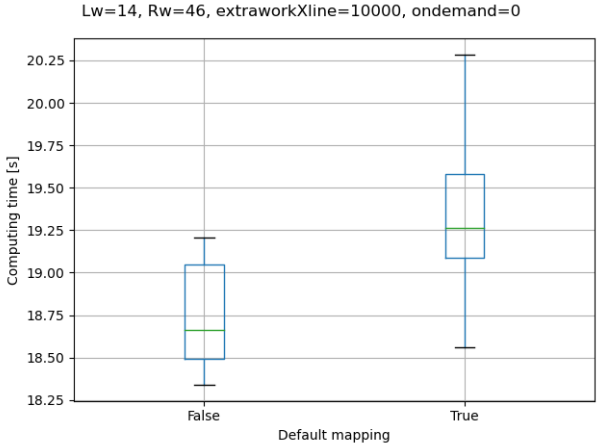

Figure 7: Execution times with and without default thread mapping when the number of left workers is set to 14, the number of right workers is 46, `extraworkXline` is 10000 and `ondemand` is 0.