Assignment 3
SPM course a.a 23/24

Spiridon Dragoș

Problem:

We want to find the top k words of a given set of text files provided. The top k words are the k most frequent words occurring in all input files **using FastFlow**.

Solution:

The solutions will be similar with the ones from the OpenMP assignment, but altered for the FastFlow framework. For the first solution, instead of tasks, we will use farms, and also it is divided in this manner: first the filenames are split into different blocks, which each one of them will send to the next farm the lines extracted. The second farm will tokenize the lines received and update a local umap, which will be sent after that to a collector. Because the order does not matter, we will use `ff_send_out(void*)`, which will make the operation of sending it to the next layer. In Figure 1, we observe that those two farms are actually an a2a, which will improve our time significantly, and also it makes more sense using this type of block instead of two farms with one of their collectors removed.

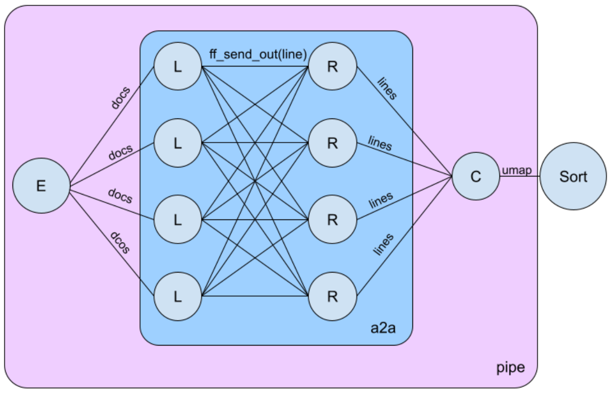

Figure 1. The skeleton of the first solution, with an a2a block and a collector for merging the umaps.

The second solution, which is similar to the Solution 2 from the previous assignment with OpenMP, we use two map-reduce patterns. Firstly, we will include a map-reduce where we will split the documents in an equal way just like in the first solution, but instead of sending them directly without waiting, we will use the reduce pattern for waiting for all of our lines to be in a single vector, and then apply another map-reduce which will separate the list of lines in an equal way and then tokenize them, like in Figure 2. The last reduce will also have the result of the word-counting.

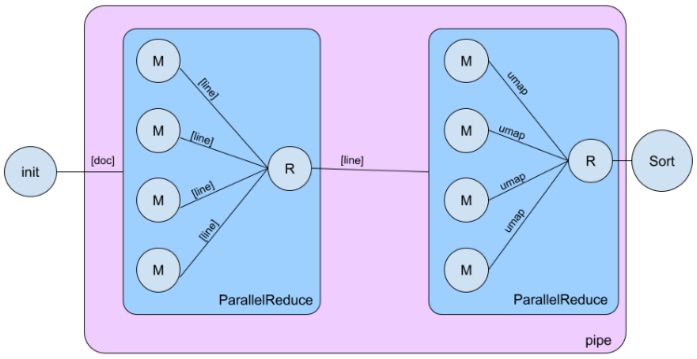

Figure 2. The skeleton of the second solution with two MapReduce blocks into a pipe.

Compile and run:

This code was run on the university cluster: spmcluster.unipi.it. The command for running the code is:

```
g++ -std=c++20 -O3 -march=native -Ifastflow Word-Count.cpp -o WC -pthread
```

Make sure you clone "fastflow" project from github before running it (link here).

For running, we will use:

```
./WC 0 5 1 16 16
```

Where the last two numbers represent the number of workers for extracting the lines from the documents respectively the number of workers for tokenizing the lines.

Results:

Solutions results: the simulations and runs presented will have the same number of workers in both parts: for extracting the lines from documents and for tokenizing the lines. In Table 1, there are presented the times taken for each solution discussed above, and also in different situations in terms of extra workload. For the first solution, which uses an "a2a" building block (called A2A in the table), it increases exponentially after 32 workers used (observable in Figure 3), which means that it should remain in a safe zone, where the number of workers in total does not exceed the number of cores (only in our environment case). In the other case, where the number of workers on each side is smaller than 20, it seems that it does not even achieve a better time than the sequential solution when workload is 0, but in the other cases of the workload, the speed up and the efficiency are having a nice curve until a certain point, which means that the model works better for a higher workload given (in comparison with the sequential time). For the second solution, which uses two MapReduce blocks (called 2MR in the table), the times are much more stable and are not increasing dramatically after using a number of workers higher than the number of cores. Moreover, it achieves the best time for all of the extraload cases, which imply big differences between the sequential and this solution. The speedup of the model is increasing well in all of the cases, which denote that this solution is having a higher stability regardless of the duration of tasks given.

threads / extraw	seq	2-2		4-4		8-8		16-16		19-19		20-20		32-32	
		A2A	2MR	A2A	2MR	A2A	2MR	A2A	2MR	A2A	2MR	A2A	2MR	A2A	2MR
0	5.644	12.4	5.74	10.2	3.80	16.7	**3.27**	13.9	3.91	13.3	3.87	15.6	3.40	17.0	4.18
1000	19.95	16.8	12.3	10.9	7.37	13.9	5.20	13.8	4.82	14.0	**4.62**	15.4	4.64	19.3	5.15
10000	149.0	78.8	75.7	39.4	39.5	21.4	21.4	14.7	12.2	14.2	10.9	16.3	**10.7**	22.1	14.0

Table 1. Results from the solutions with a2a and MapReduce given different extraload.

Comparison and better approaches: from Figure 3 first column, where it shows the time for each model, it is clearly that the 2MR solution gives better results, because it overcomes A2A model in almost all cases of number of workers. For the second column, where there is the speed up of each model on different workloads, the situation remains the same, and also the speed up is staying up on several cases, whereas A2A decreases in a smaller step. In terms of efficiency, which is seen in the third column, both of the models are having the same curve, but with the 2MR model always being a bit higher than A2A. I consider that, for a better approach and a closer examination of the efficiency of the models, the number of workers should vary between the first part (separating the lines) and second (tokenizing the lines), and not always equal. In this case, the number of workers for separating the lines should always be smaller than the number of workers for tokenizing, because the tokenizing part should always be more expensive than separating, which also gives us the time service of the pattern.

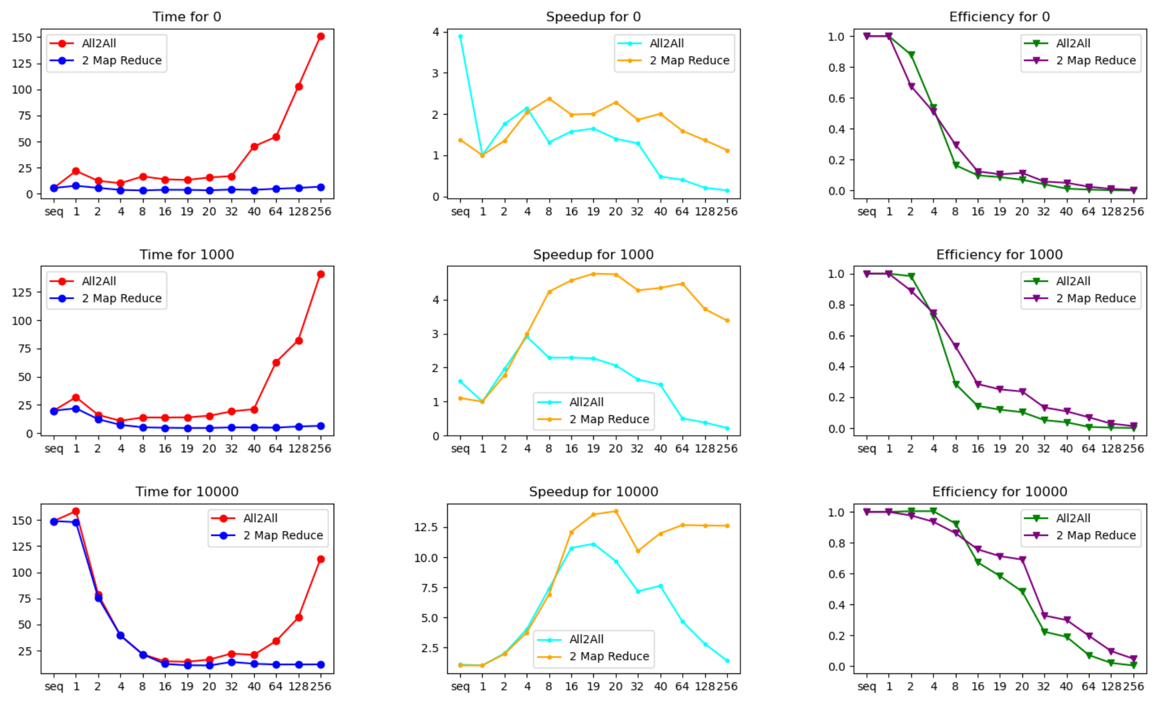

Figure 3. Results from the first solution

Conclusion

In terms of the library, FastFlow shows a much elegant way of solving the parallel paradigms by using building blocks with different properties. In comparison with OpenMP, FastFlow presents a much complex way to parallelize a program, but provides an interesting and efficient way to connect the blocks between each other, and also gives more powerful tools for stream parallel patterns. It is clearly harder to understand it (due to non-triviality of the blocks), but I consider it a tool which could be used in different parallel problems and paradigms.

For the point of view of the solutions from the perspective of the ones from the previous assignment, the inspiration of them were from that assignment, where I implemented kind-of the same models, but in OpenMP. The solutions from FastFlow have beat the other solutions in many cases, one of the examples being that they are not having threads for omp tasks which took a long time if the workload was not high enough, and also in terms of the complexity of the models. If we compare the Figures for each of them, the models from the OMP assignment needed much more explanations rather than the ones from FastFlow assignment.

A conclusion for the models is that they have a good scalability in terms of time and speedup, even better when the workload is high, and also that the overall performance of model with two Map Reduce building blocks is greatly exceeds the performance of the model with an All2All building block, but different aspects could have been touched: distribute differently the number of workers on each side, changing the CPU affinity, and also trying solutions with FastFlow properties, like concurrency control between blocking and unblocking protocols, channel capacity, and also maybe trying to change the model with the help of the feedback channels.

All things considered, the solutions describe an easy pattern for a parallel paradigm considering the FastFlow building blocks, and the MapReduce pattern seems to be more efficient than a constant flow of the data with an All2All block, even though the two Map Reduce blocks are using kind of a barrier between them. After a lot of trials, the solutions are greatly optimized, but more tests including other aspects of the flow of the data should be taken into consideration, for achieving even better results.