Assignment 2
SPM course a.a 23/24

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.

Solution:

For a first solution, I applied a kind-of MapReduce pattern over our problem: first use only a single thread to separate each line of each document, and when a line is separated, create an "omp" task which tokenize the line, and each token is put into a special dictionary for that **thread**. This means that each thread has its own dictionary where they can insert/ update the keys. Second, we will reduce them to one dictionary by combining the dictionaries two by two (not actually two by two, the dictionaries are taken in a different way: first with last, second with second to last etc.), until it will be only one dictionary, as in Figure 1. The last step is the same as the last step from the sequential one: sort the dictionary and take only the top k items in which we are interested in.

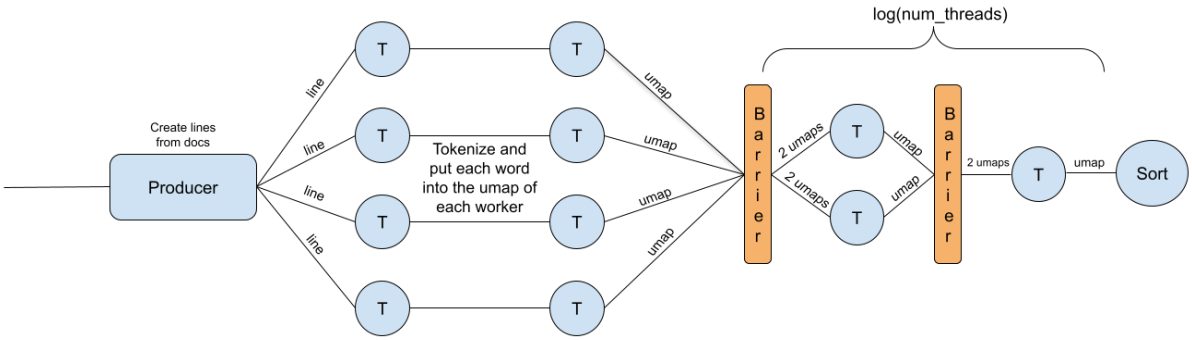

Figure 1. The pattern that the first program is following. T and its lines come from Task created with a certain input and "returning" an output.

The first part is more of a "task-farm" instead of map, because they are not split equally and we spawn the tasks for whatever thread can work with it (they are more as a worker). To be precise, after a line is extracted and put into a task, the thread number will have an important role, because that line would affect a specific dictionary. In this way, we can work with shared variables without interfering between them.

A second solution, which can be observed in Figuer 2, is to literally implement the MapReduce by using the map pattern for files and also for all of the lines. We also use different lines for reduction, in which we implement special reductions for lists. After we get a list of pairs, in the official solution, we should sort and shuffle them for running another reduction on the code. Instead, we will map the pairs into a local umap and then merge them. For merging, we can also use the method used in Figure 1 with a binary tree of log(num_threads), but in our results, a "Map Merge reduce" is applied. "Map Merge reduce" means that from all of the local umaps we will merge them into only one, and this was made with two methods: the first one is by declaring a specific reduction for maps, and the second method is by using a critical method. All of these solutions have approximately the same time for merging from the tests made (the service time is given by the tokenization part), but the second solution uses only the "Map Merge Reduce" in our tests.

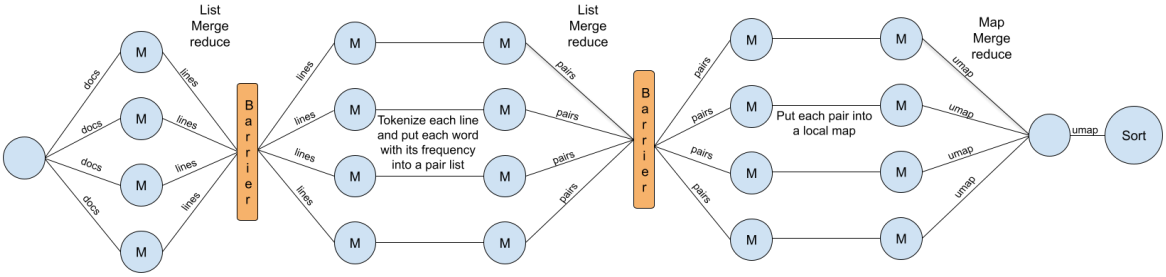

Figure 2. The pattern that the second program is following. M and its lines come from Map created by getting a certain input and "returning" an output.

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 -fopenmp Word-Count-line.cpp -o WCl
```

And for running, I suggest to call it with the environment variable "OMP_NUM_THREADS" before actually calling it. For example, this code will run with 16 omp threads:

```
OMP_NUM_THREADS=16 ./WCl 0 5 1
```

Results:

First solution: by taking the model of the first solution, we will run it in two different methods: the separation of tasks by the producer will be made by line (L) or by document (D) (for observing different experiments). From Table 1 we can observe that for adding extra work per tokenization will give a high number in the sequential case, but in the other case, we see that we have good numbers in that term and it is actually improving. Weird scenarios happened at 16, 32 threads, as if the tokenization has extra work, it actually improves the time somehow. We can assume also that the threads in those cases had to open and close so often that it increases the computational time, but in the case with highest extra work, the threads never "close" and they always have work to do. In figure 3, we observe the results of time, speedup and efficiency for each extra work used. For extra work 0, it doesn't improve at all, instead it goes down for lines separation, in terms of speedup. Interesting that the best time we got in all of the tests are got from thread 2 by separating the tasks by docs. For the other two times, we see that tasks over lines is starting to improve, but with a weird spike which increases a lot on thread 40, but flattens immediately after that. At extra work 1000, they have similarly the same time, but at extra work 10000, we see a big decreasing of time compared to the sequential time. This means that a thread in terms of tasks works better if it has a task prepared for him before, because if they do not have work until the next task comes, then they will work worse from what have been observed.

threads / extraw	seq	1		2		4		8		16		32		64	
		L	D	L	D	L	D	L	D	L	D	L	D	L	D
0	5.644	5.95	5.50	5.37	**3.70**	10.8	5.26	20.5	5.47	43.8	5.20	67.8	5.65	15.1	5.32
1000	19.95	20.4	19.9	12.4	**12.3**	10.7	13.2	13.5	**12.3**	42.1	12.6	66.8	12.7	15.7	13.2
10000	149.0	148.	149.	79.2	87.4	43.2	78.4	24.8	74.7	**17.1**	73.3	21.7	78.9	17.5	73.2

Table 1. Results from the Task parallelization from Figure 1

Second solution: the results from the second solution are having an improvement in all of the extra work, and also without spikes. In Figure 4, it is observed that at the 40 threads we have the best result in terms of time and speedup, but the efficiency seems to behave the same as the previous results from Figure 3. In Table 2, we have two types of columns: the tokenization separated by pairs (P) and also represented in Figure 3, and the tokenization separated by umaps (M), where instead of doing the second "List Merge

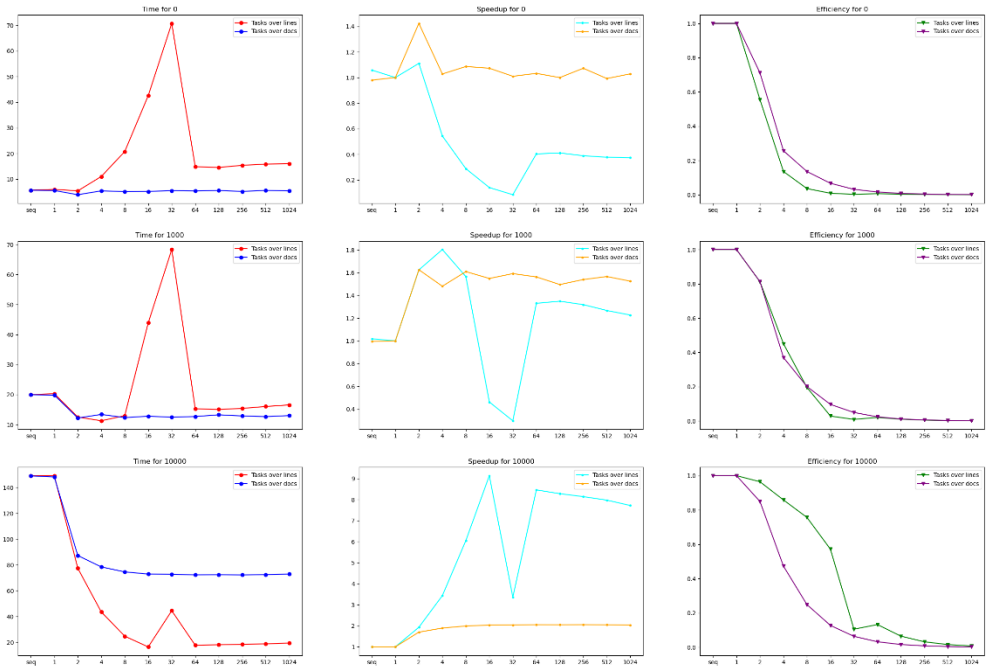

Figure 3. Results from the first solution

Reduce", it will jump directly to the last part. In comparison, the column M will always have a better time rather than the P column, which is somehow logic, because the pairs have to introduce an operation in addition. Because we only compare the latency, the first model with umaps is better than the second model with pairs. The lowest time is always on thread 40, which is also the number of threads given by OpenMP when we do not define the variable OMP_NUM_THREADS. After 40 threads used, the time will start to increase, and obviously the speedup starts to decrease.

threads / extraw	seq	2		4		8		16		32		40		64	
		M	P	M	P	M	P	M	P	M	P	M	P	M	P
0	5.644	4.71	8.65	6.16	9.41	6.57	9.23	6.23	8.76	5.06	7.12	**4.62**	6.74	4.72	7.10
1000	19.95	12.1	16.1	10.2	13.8	8.99	11.3	8.22	10.3	7.45	9.26	**7.17**	9.68	7.53	9.85
10000	149.0	76.9	80.5	42.7	45.9	25.2	28.2	16.9	19.2	13.9	15.1	**13.4**	15.5	12.9	15.5

Table 2. Results from Map-Reduce solution from Figure 2

Comparison and better approaches: In comparison with the first model which separates the task by lines, the results from this model are obviously more stable (without spikes), more trivial in terms of the architecture, and also provides a predictable time in terms of number of threads used. For the solution which uses tasks per document, the results are having a satisfactory time, until the extra work is 10000, in which it only doubles the speedup with any threads it is using, which makes sense as it is taking a lot to finish a document, and a lot of threads probably would do nothing after they finish with their parts of documents. The separation by docs is really unsatisfactory, because we do not know how large are the documents, so the separations of the documents should be more balanced. This part can also be applied to the task by lines, because I think that if a task would have more lines rather than one, the time on a thread is better split, and the thread won't turn on and off with the usage of the tasks.

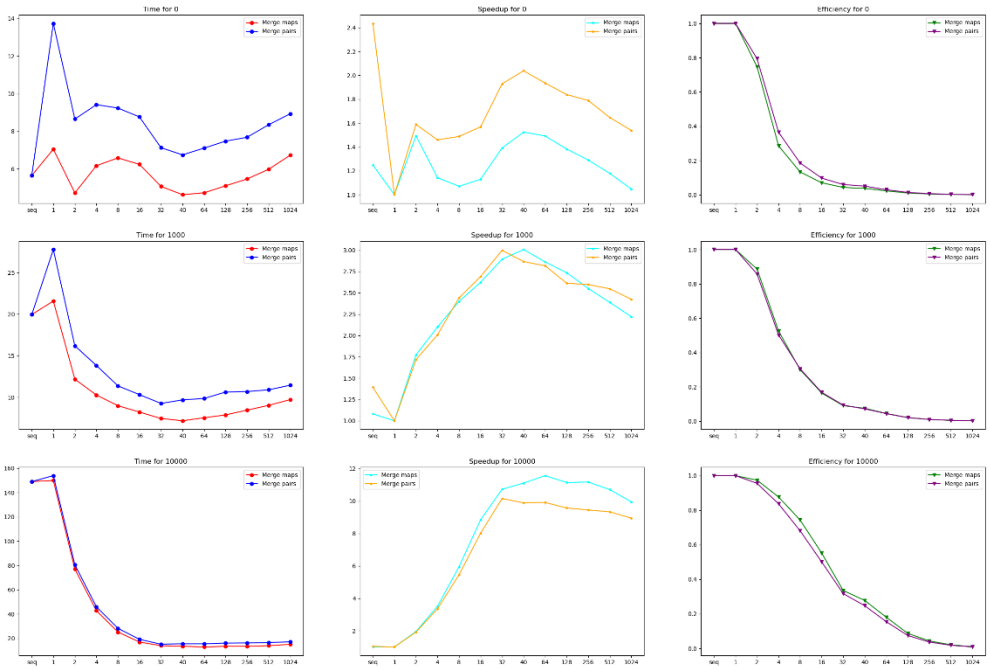

Figure 4. Results from the second solution

Conclusion

OpenMP is a strong API multiplatform for C++, which can make the code cleaner, and also have a lot of useful tools that helps the parallelization part of the code be much simpler to make. In terms of the MapReduce pattern, it is not so trivial to implement it at the beginning, but after understanding more and more techniques (like declare reductions), and how to order them, the scenario becomes easier to implement. From the perspective of OMP tasks, even though they seem simple (you put a task in a queue), their behavior is not that simple to understand. From Table 1, we observe that it is taking much more time when the task is actually smaller, which is really questionable, but not if you think that if it has more tasks in the queue, the thread does not have to go to “sleep”, and so it can take the next task instead of waiting for another task to be put in the queue and do nothing. If we separate the lines in a clever way maybe we can get better results even compared to the second model used.

Even though the assignment asked specifically for OMP tasks, I consider that it is important to observe the difference between two solutions using different strategies, and which one is actually more accessible and, in some way, easier. MapReduce is an easy concept to understand, but complex to implement, and with that in mind, these two different versions are showing solutions for solving that exercise, which can evolve into merging part of their skeletons used from both of them (like it was done with different versions of code which merged the umaps per thread).

The perspective of the service time should have been also taken into consideration. Even if we already know that the service time is actually the time of tokenizing the line (because that is the part which is taking the most), it is an important factor of improving the code, for offering better versions from the perspective of throughput and stream parallelization.