SPM Assignment - 2

Simone Passèra

May 07, 2024

1 Introduction

This assignment was intended to parallelize an algorithm that 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.

I implemented a parallel versions of the `Word-Count-seq.cpp` code using OpenMP task. The implementation and the tests performed on the clusters provided by the University of Pisa are presented in more detail below.

2 Code Implementation

```
1  void tokenize_line(const std::string& line, std::vector<umap>& umap_results) {
2    char *tmpstr;
3    char *token = strtok_r(const_cast<char*>(line.c_str()), " \r\n", &tmpstr);
4
5    while(token) {
6      umap_results[omp_get_thread_num()][std::string(token)]++;
7      token = strtok_r(NULL, " \r\n", &tmpstr);
8
9      #pragma omp atomic
10     total_words++;
11   }
12
13   for(volatile uint64_t j {0}; j < extraworkXline; j++);
14 }
15
16 void compute_file(const std::string& filename, std::vector<umap>& umap_results,
17                   uint64_t num_lines) {
18   std::ifstream file(filename, std::ios_base::in);
19
20   if (file.is_open()) {
21     std::vector<std::string> *lines;
22     std::string line;
23     int l;
24
25     while(!file.eof()) {
26       lines = new std::vector<std::string>();
27       lines->reserve(num_lines);
28       l = num_lines;
29
30       while((l != 0) && !std::getline(file, line).eof())
31         if (!line.empty()) {
32           lines->emplace_back(line);
33           l--;
34         }
35
36       #pragma omp task default(none) firstprivate(lines) shared(umap_results)
37       {
38         for (auto i = lines->begin(); i != lines->end(); i++)
39           tokenize_line(*i, umap_results);
40
41         delete lines;
```

```
42        }
43      }
44    }
45
46    file.close();
47  }
48
49  int main(int argc, char *argv[]) {
50    .....
51
52    // used for storing results
53    std::vector<umap> umap_results(num_threads);
54
55    // start the time
56    auto start = omp_get_wtime();
57
58    #pragma omp parallel master num_threads(num_threads) default(shared)
59    {
60      #pragma omp taskloop default(shared)
61      for (auto f : filenames)
62        compute_file(f, umap_results, num_lines);
63    }
64
65    auto stop1 = omp_get_wtime();
66
67    if (num_threads > 1) {
68      for (int id = 1; id < num_threads; id++)
69        for (auto i = umap_results[id].begin(); i != umap_results[id].end(); i++)
70          umap_results[0][i->first] += i->second;
71    }
72
73    // sorting in descending order
74    ranking rank;
75    rank.insert(umap_results[0].begin(), umap_results[0].end());
76
77    auto stop2 = omp_get_wtime();
78
79    .....
80  }
```

Listing 1: Word-Count parallel version.

The code in Listing 1, shows the main parts of the program `Word-Count-par.cpp`. In the main function, line 49, inside the parallel region the master thread spaws a task for each file, which executes the function, *compute_file*, which takes as a parameter the vector of umaps, (each thread uses its own umap to store words), and the maximum number of rows that will be assigned to each new task that will be responsible for counting the words.

The *compute_file* function, line 16, opens the file passed as a parameter, and reads `num_lines` rows, storing them on the heap, until the end of the file is reached, and for each set of rows creates a task that executes the *tokenize_line* function for each row.

The *tokenize_line* function, line 1, differs from the sequential version by storing words in the thread umap and by the shared variable `total_words` becoming atomic through the `#pragma omp atomic` directive.

After all tasks are terminated, a merge of all umaps into one is performed (line 67) and will be subsequently sorted at line 75.

3 Evaluation

This section discusses the tests performed on the implemented parallel version in order to evaluate its performance. The tests were conducted on the course machines (spmcluster.unipi.it and spmnuma.unipi.it). Each measurement was performed 10 times and the average value was considered. The standard deviation is negligible, and is not present in this document. The tests were performed by compiling the program with the following command:

```
1 g++ -std=c++17 -O3 -march=native -fopenmp Word-Count-par.cpp -o Word-Count-par
```

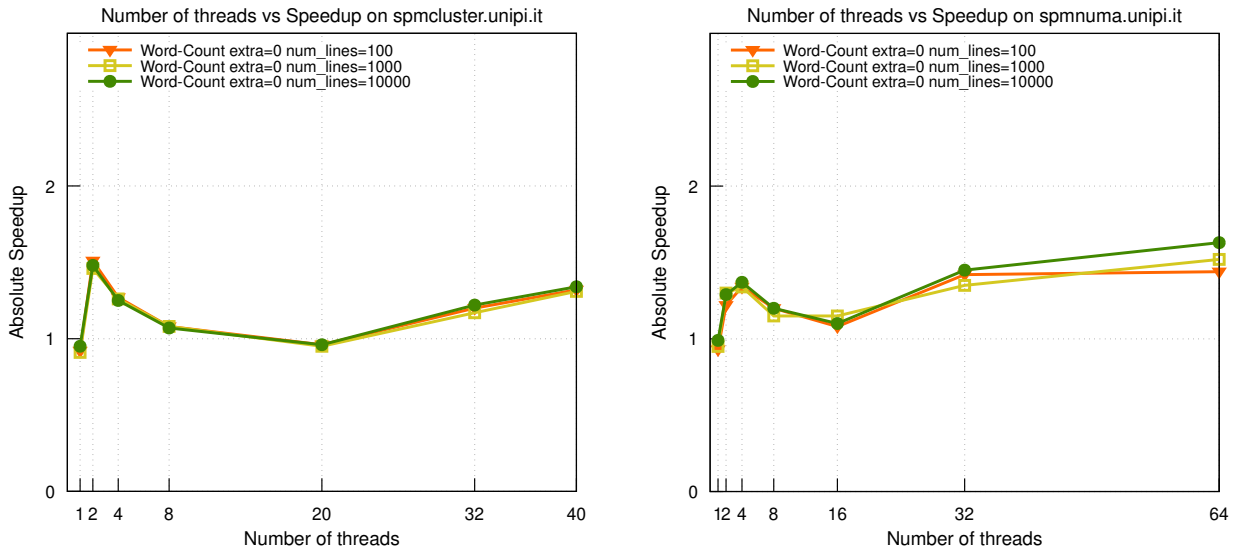

Figure 1: Absolute Speedup vs Number of threads. ($extraworkXline = 0$).
`Left`: test performed on spmcluster.unipi.it. `Right`: test performed on spmnuma.unipi.it.

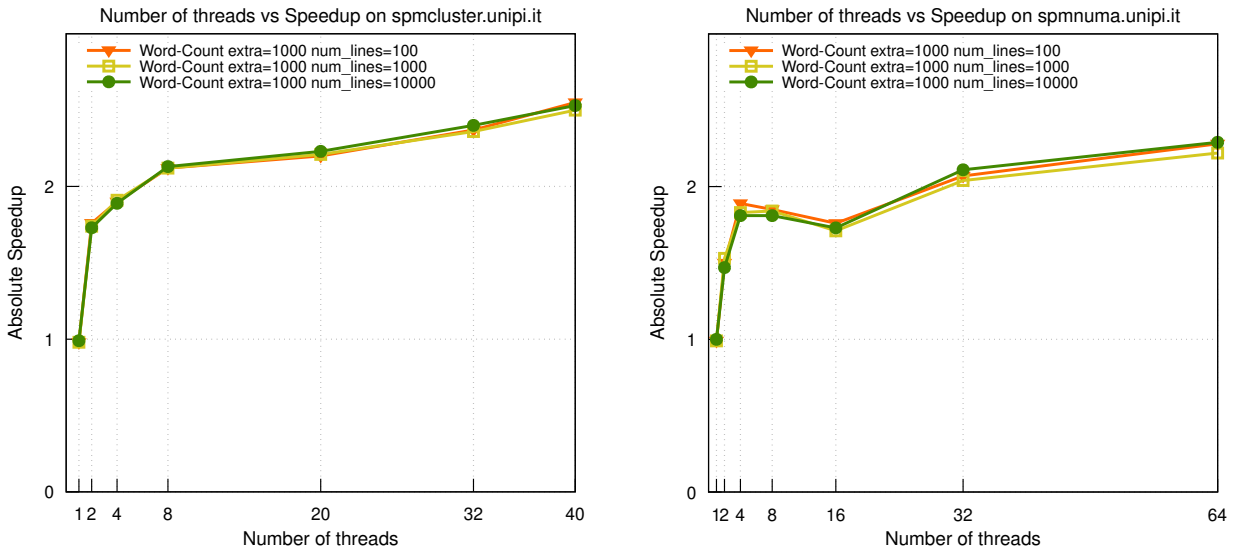

Figure 2: Absolute Speedup vs Number of threads. ($extraworkXline = 1000$).
`Left`: test performed on spmcluster.unipi.it. `Right`: test performed on spmnuma.unipi.it.

Absolute Speedup is the metric used to evaluate the performance of the parallel version. Tests were performed by varying the `num_lines` and `extraworkXline` parameters. The graphs show the speedup as the number of threads changes for the *spmcluster.unipi.it* cluster, which has a maximum of 40 logical cores and *spmnuma.unipi.it* cluster, which has a maximum of 64 logical cores.

In Figure 1, the program was run with $extraworkXline = 0$. In the graph on the left, it can be seen that the maximum speedup is achieved with the use of only a few threads, because the program is primarily memory-bound and therefore cannot scale very well. On the other hand, for the NUMA architecture, graph on the right, increasing the number of threads results in a higher speedup, perhaps because of how memory is organized. As for the parameter that determines the number of lines computed by a task, you can see that increasing it reduces the number of tasks created, thus reducing overhead and increasing speedup.

In Figure 2, the program was run with $extraworkXline = 1000$. In this case by introducing additional computation the graph trend is more linear and a higher speedup is achieved.

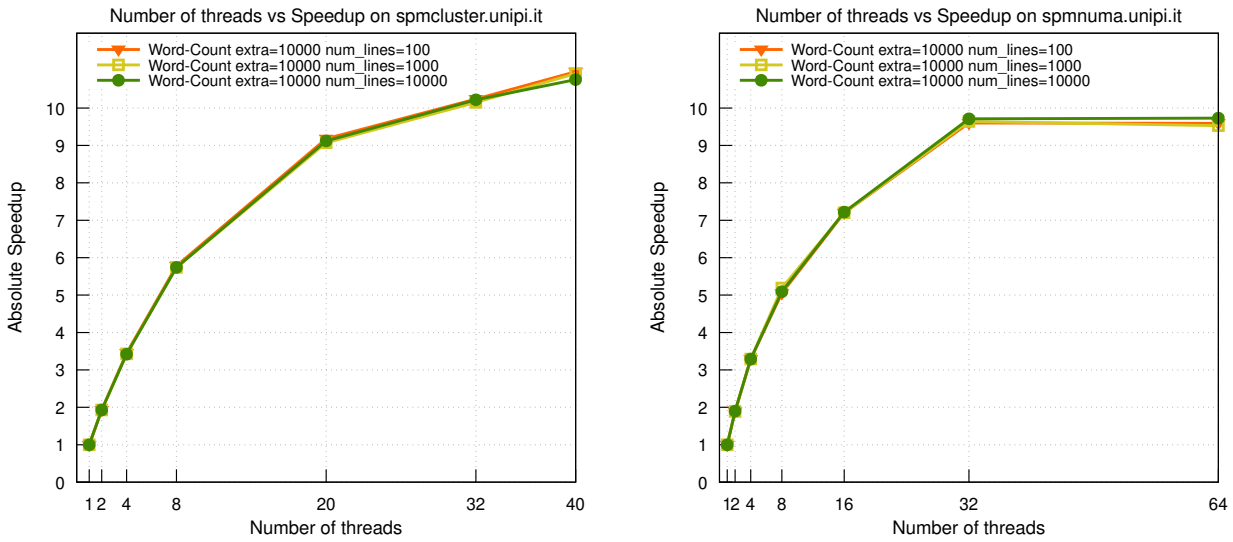

Figure 3: Absolute Speedup vs Number of threads. ($extraworkXline = 10000$).
`Left`: test performed on spmcluster.unipi.it. `Right`: test performed on spmnuma.unipi.it.

In Figure 3, the program was run with $extraworkXline = 10000$. In this case the additional computation is even higher actually making the program more cpu-bound and therefore the problem is more parallelizable. In fact, the speedup achieved is higher and the graph takes on a more linear pattern as would be expected. For the NUMA architecture, it can be seen that the switch between 32 and 64 threads, and thus the use of hyperthreading, does not lead to a significant improvement.

4 Conclusions

To summarize, the tests performed confirmed that this particular problem scales poorly. Through the *extraworkXline* parameter, used to simulate more computation, an improvement for speedup can be achieved. The implementation with OpenMP has been made trying to minimize the overhead introduced, realizations of higher performance implementations are not excluded.