SPM Assignment - 3

Simone Passèra

May 14, 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 FastFlow. The implementation and the tests performed on the cluster provided by the University of Pisa are presented in more detail below.

2 Code Implementation

```cpp
struct Reader : ff_monode_t<std::vector<std::string>> {
  Reader(const std::vector<std::string> &filenames_, const uint64_t num_lines_,
         const uint64_t left_workers_) : filenames(filenames_), num_lines(num_lines_),
         left_workers(left_workers_) {}
  std::vector<std::string>* svc(std::vector<std::string>*) {
    uint64_t id = get_my_id();
    uint64_t filenames_size = filenames.size();
    uint64_t block_size = filenames_size / left_workers;
    uint64_t num_files = block_size;

    if (filenames_size % left_workers != 0)
      if (id < left_workers)
        num_files++;

    for (uint64_t i = id * block_size; i < ((id * block_size) + num_files); i++) {
      std::ifstream file(filenames[i], std::ios_base::in);

      if (file.is_open()) {
        std::vector<std::string> *lines;
        std::string line;
        int l;

        while(!file.eof()) {
          lines = new std::vector<std::string>();
          lines->reserve(num_lines);
          l = num_lines;

          while((l != 0) && !std::getline(file, line).eof())
            if (!line.empty()) {
              lines->emplace_back(line);
              l--;
            }

          ff_send_out(lines);
        }
      }
      file.close();
    }
    return EOS;
  }

  const std::vector<std::string> &filenames;
  const uint64_t num_lines;
  const uint64_t left_workers;
};

struct Tokenize : ff_minode_t<std::vector<std::string>> {
  Tokenize(umap &um_) : um(um_) {}
  std::vector<std::string>* svc(std::vector<std::string>* lines) {
    for (auto l = lines->begin(); l != lines->end(); l++) {
      char *tmpstr;
      char *token = strtok_r(const_cast<char*>(l->c_str()), " \r\n", &tmpstr);

      while(token) {
        um[std::string(token)]++;
        token = strtok_r(NULL, " \r\n", &tmpstr);
        total_words++;
      }

      for(volatile uint64_t j {0}; j < extraworkXline; j++);
    }

    delete lines;
    return GO_ON;
  }

  umap &um;
};

int main(int argc, char *argv[]) {
  .....

  // used for storing results
  std::vector<umap> umap_results(right_workers);
  // start the time
  ffTime(START_TIME);

  ff_a2a a2a;
  std::vector<ff_node*> left_w;
  std::vector<ff_node*> right_w;

  for (uint64_t i = 0; i < left_workers; i++)
    left_w.push_back(new Reader(filenames, num_lines, left_workers));

  for (uint64_t i = 0; i < right_workers; i++)
    right_w.push_back(new Tokenize(umap_results[i]));

  a2a.add_firstset(left_w, 1);
  a2a.add_secondset(right_w);

  if (a2a.run_and_wait_end() < 0) {
    error("running a2a\n");
    return -1;
  }

  ffTime(STOP_TIME);
  auto time1 = ffTime(GET_TIME) / 1000;
  // start the time
  ffTime(START_TIME);

  if (right_workers > 1) {
    for (uint64_t id = 1; id < right_workers; id++)
      for (auto i = umap_results[id].begin(); i != umap_results[id].end(); i++)
        umap_results[0][i->first] += i->second;
  }

  // sorting in descending order
  ranking rank;
  rank.insert(umap_results[0].begin(), umap_results[0].end());

  ffTime(STOP_TIME);
  auto time2 = ffTime(GET_TIME) / 1000;

  .....
}
```

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 78, it can be seen that the business logic of the program is realized through the all-to-all building block, which is capable of emulating a farm if the number of L-Workers is equal to one. L-Workers are realized through the multi-output `Reader` node and R-Workers with the multi-input `Tokenize` node. L-Workers use the on-demand scheduling policy, line 88. Each R-Worker, which will then be a thread, stores the number of words in its own local umap. After all-to-all termination, the various umaps are merged into a single umap, line 101, and then sorted, line 109.

The *Reader* node (L-Worker), line 5, is responsible for reading a set of files assigned to it using a block distribution. For each file, it reads `num_lines` rows, storing them on the heap, until the end of the file is reached, and for each set of rows sends the pointer to them to an R-Worker via the function *ff_send_out()*.

The *Tokenize* node (R-Worker), line 49, tokenizes each row in the input vector and stores the number of words encountered in the local umap.

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 machine (spmcluster.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:

```
g++ -std=c++17 -O3 -march=native -I ./fastflow
    -DNO_DEFAULT_MAPPING Word-Count-par.cpp -o Word-Count-par
```

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 R-Workers and L-Workers changes for the *spmcluster.unipi.it* cluster, which has a maximum of 40 logical cores. For L-Workers the values used are 1 and 8 so as to appreciate the difference between *farm* and *all-to-all*.

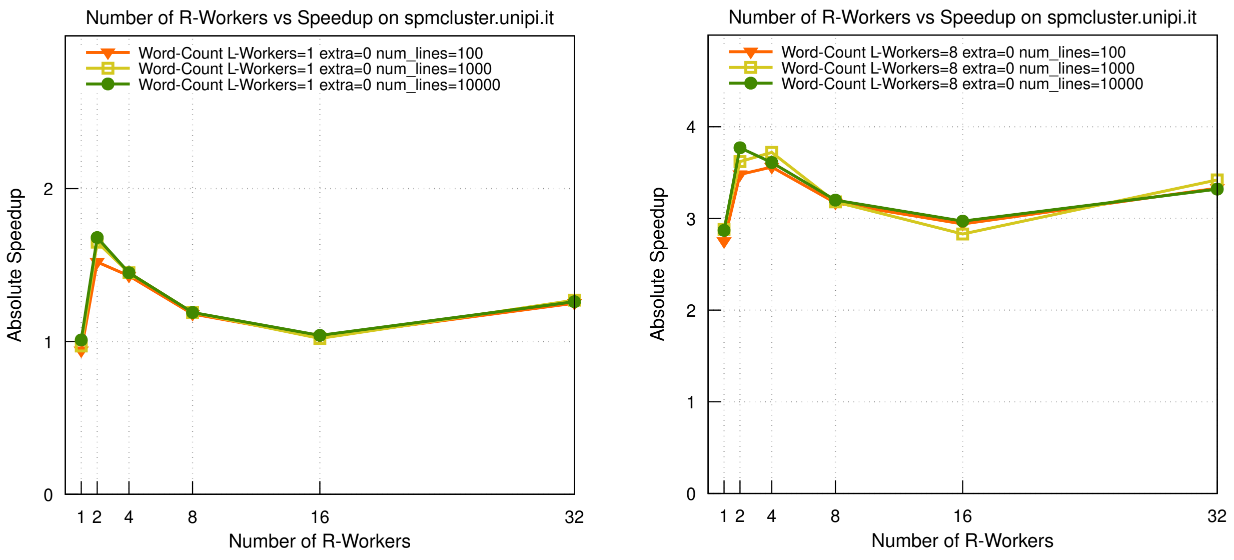

Figure 1: Absolute Speedup vs Number of R-Workers performed on spmcluster.unipi.it. ($extraworkXline = 0$). `Left`: L-Workers = 1 (farm). `Right`: L-Workers = 8 (all-to-all).

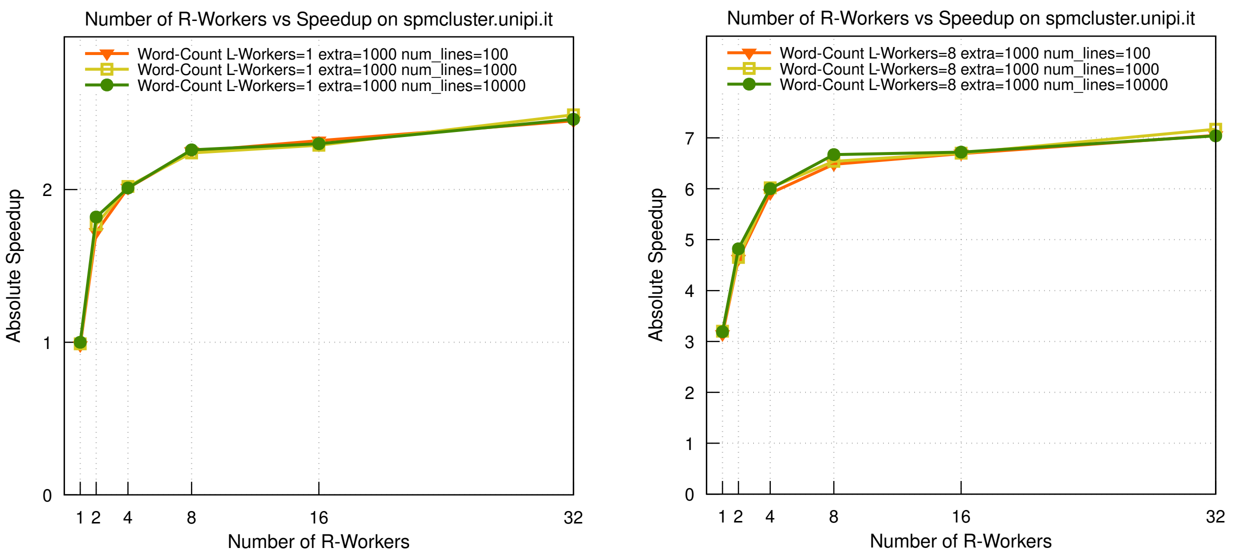

Figure 2: Absolute Speedup vs Number of R-Workers performed on spmcluster.unipi.it. ($extraworkXline = 1000$). `Left`: L-Workers = 1 (farm). `Right`: L-Workers = 8 (all-to-all).

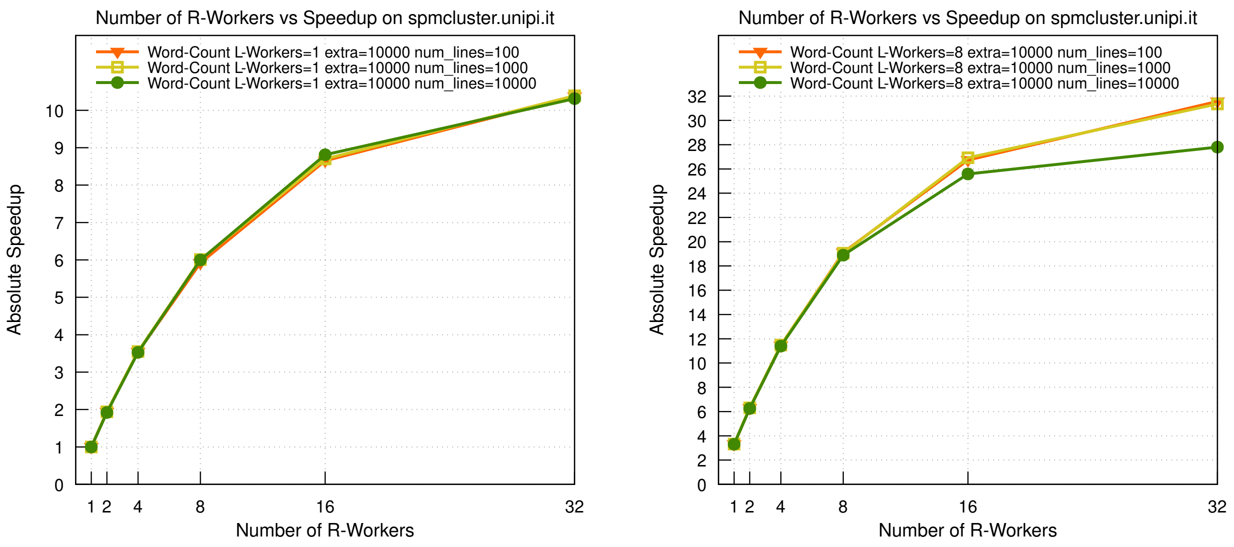

Figure 3: Absolute Speedup vs Number of R-Workers performed on spmcluster.unipi.it. ($extraworkXline = 10000$). `Left`: L-Workers = 1 (farm). `Right`: L-Workers = 8 (all-to-all).

In the graphs above, it can be seen that as the *extraworkXline* parameter increases the speedup also increases, this is because the problem is mainly memory-bound as has already been observed in the version made with OpenMP. The speedup achieved with the farm is comparable with the OpenMP version. Instead, it is possible to observe that with all-to-all the speedup is significantly higher. It is therefore evident how increasing the degree of parallelism for L-Workers the execution time found is greatly reduced. In conclusion, the version built through FastFlow achieves higher performance than the version built with OpenMP tasks.

3.1 CPU affinity

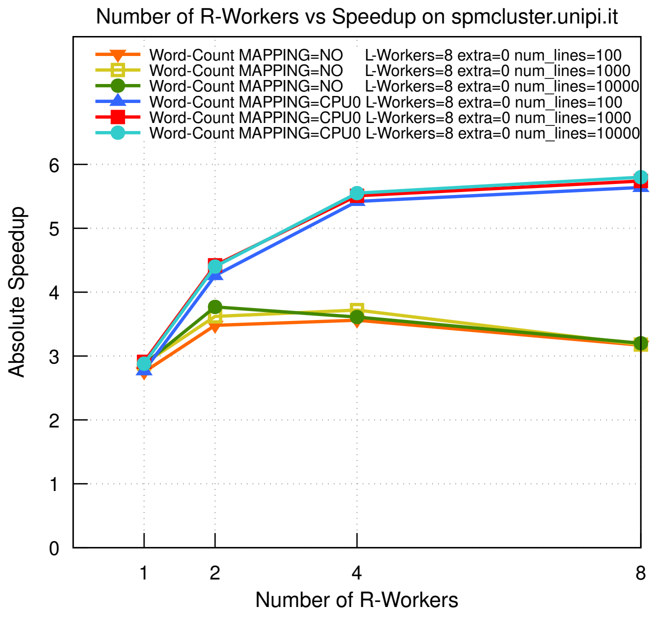

Figure 4: Absolute Speedup vs Number of R-Workers performed on spmcluster.unipi.it. ($extraworkXline$ = 0). L-Workers = 8 (all-to-all). Same test performed with thread mapping disabled and on a single CPU.

FastFlow (like OpenMP) provides the ability to control the placement of threads on CPU cores. FastFlow as a default behavior pins the threads by extracting the id of the cores from the string `FF_MAPPING_STRING`. The test in Figure 4, was performed by disabling the default mapping of FastFlow and instead taking advantage of the linux `taskset` command, this is simply because it is easier to use and it is dynamic:

```
taskset -c 0:38:2 ./Word-Count-par filelist.txt [leftWorkers] [rightWorkers]
                                  [lines] [extraworkXline] [topk] [showresults]
```

If you want to achieve the same result using FastFlow mapping the `ff_MAPPING_STRING` string for the cluster *spmcluster.unipi.it* is as follows:

```
"0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39"
```

Figure 4, shows the difference between disabled mapping and instead placing threads in a single CPU of the *spmcluster.unipi.it* cluster, which can host 20 logical cores in a single CPU. As can be seen by increasing the number of threads, the operating system is more likely to place them in different CPU. In fact, the speedup obtained is the highest with mapping enabled and with 16 threads (8 L-Workers + 8 R-Workers). In conclusion, placing the threads in the same CPU reduces latency times because you take advantage of memory hierarchy and faster communications within the same CPU.

4 Conclusions

To summarize, the version implemented with FastFlow achieves better performance than the previous version built with OpenMP tasks, due to greater support for parallel structured programming. In addition, good use of thread mapping can lead to significant improvements.