High Performance Computing
Semester: 2021/22-1

László Nagy, Gábor Vitrai

[1] University of Trento, TN, Italy https://www.unitn.it/
[2] Department of Information Engineering and Computer Science

Abstract. This document was created for the High Performance Computing for Data Science (HPC) Subject as a final - end of semester - assignment/project. The project consists of a data mining problem where the aim is to generate a frequency table from a text using parallel processing solutions. The project includes an optimized solution for file access, different solutions regarding the communication between processes, a hashing solution with a special derived datatype using the MPI library and a final comparison between the solutions.
The project is run on the High Performance Cluster of UniTN.

Keywords: HPC4DS · Hash · Frequency Table · MPI · Data Mining

1 Challenge, state of the art, motivation

A frequency table is a dictionary mapping each (word, sentiment) pair to its frequency. Since we don't have the sentiments for each line, we are computing the word frequency of a text. The aim of the project would be to optimally process a book. Why would this be useful in practice: In case we would have the label of each word in our dictionary, our frequency table could be used to create an excellent supervised learning solution. As this process can take a lot of time for bigger inputs, we would like to implement different solutions to see how the amount of processes and the number of communications in between processes effects the execution time in the case of such small elements (words).

```
dict = {
    {"word_1": 10},
    {"word_2": 6}
};
```

2 Problem analysis

In case we would like to generate a frequency table for (e.g.) reviews or tweets left by users, we need to compute millions of records, which despite consisting a small tasks, the required computational power increases exponentially as the number of words is getting bigger.

A couple of different tasks are needed to tackle this problem. First, a suitable input dataset has to be found, that is large enough to take advantage of parallel processing. Then this data has to be cleaned to be ready for the dictionary creation. Finally, a parallel processing implementation has to be devised, to go through different parts of the data at the same time.

2.1 Serial code

The serial solution was implemented first. It opens the file, goes through it line by line and for every word inside the file, it creates a record in the dictionary with an initial value of 1. For each additional time a word appears, this value gets increased by 1. When adding words to the dictionary, we leverage the hashing library to do this in a fast manner.
In the end, the dictionary is sorted based on the occurrences of the words. This sorting method is included in the library we are introducing later in this document.

```
program sequential
    set dictionary = [];
    open input_file;
    for each line in input_file:
        read line;
        for each word in line do:
            set word = line.split(' ');
            add word to dictionary;
    print execution_time;
    sort dictionary by word_occurrence;
    exit;
```

3 Steps towards parallelization

3.1 Pre-Processing

In order to create the table without any problem, we need to pre-process books. To take care of this, we decided to use python as it has powerful tools for text processing. We have created a script which can be parameterized.
The script needs to remove many different characters and symbols, for this we mostly used regex and replace methods. Dots, commas, different types of new line symbols were needed to remove. Afterwards the text can be cleaned. Content removed from the text:

- Stop Words (e.g.: I, the, of, he, her, it's, so, a , and...)
- Special Punctuation (e.g.: "..." or ". . . ")
- White spaces at the beginning or end of a sentence
- Special Characters (e.g.: ", ', /, (, [, *, -, —)
- Not Allowed Long words (e.g.: "aaaaaaaaaaRRREEEEEEEEEeeeeeeeeeee")
- Empty Lines
- Optional stemming is also available

3.2 Design of the parallel solution

Getting the Input All the processes have access to the input file, so instead of opening it with the master process and sending a section to each child processes, we decided to make a method which can open a specific part of the input. Input handling went through some iteration. First, we decided next to an algorithm [1] which was made to handle single line reading from a file, based on a single line number.
This method was modified to return multiple lines, so each process can read it's specific section of the file in an optimized way. To use the method, we need to specify the file path, the line numbers which in between we want to have the text returned, and the output buffer. With these parameters, the method will read chunks from the input to read and return the asked lines in an optimized way.

Creating the Frequency table As described earlier, we are creating a Frequency table, however for this, we need a dictionary data structure, which is not available in generic C, so we had to come up with a solution. Any custom C data structure can be stored in a hash table using **uthash**. The only component which was needed is a "UT_hash_handle" attribute to the structure, and after defining the key and value components, we could already use macros to store, retrieve or delete items from our hash table.

Custom MPI Data Type Having the custom data structure is amazing as it is fast and reliable. On the other hand, this datatype was not transferable with MPI Send and Recv, so we created the MPI version of the same data structure, this time, without the hashing handler. In order to make the MPI datatype transferable, we had to define type, size and memory attributes (offsets).

3.3 Implementation

Distributing the tasks In order to achieve parallel processing of the text input, we have to divide it between the tasks. We had to write this in such a way that we can run the program with any number of processes and it divides the input in the same way.
For this, we compute start and end positions for each process based their rank and we know that the length of their chunk needs to be the total number of lines divided by the number of processes.

Handling parallel communication One of the biggest bottlenecks in the program apart from the IO operations related to our input file, is how the communication between processes is handled. Increasing the number of communication operations between processes results in dramatic decrease in performance as we'll see in the benchmarking section.
In our implementation, each child process creates a local dictionary with the

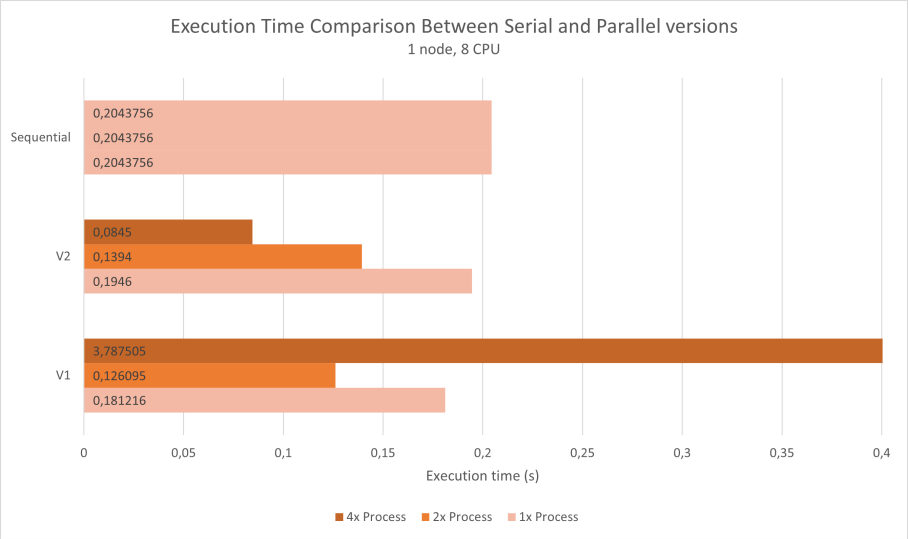
words from their chunk of input text. After this, they send the length of the dictionary to the main process so it knows how much communication it should expect next.
In **Version 1**, after this each child process send all elements of the local dictionary one by one, with an *MPI_Send* operation. We realized that as we started scaling the application, the exponentially increasing number of communications increased execution time in the same way. So in **Version 2**, we create an array out of the local dictionary elements and leveraging the fact that one can send multiple objects in a single *MPI_Send* call, we send it using only one of them. This results in the main process also having to do only one corresponding *MPI_Recv* operation.

Pseudo code

```
program parallel
    initialize MPI library;
    create custom datatype;
    master process:
        set master_dictionary = [];
        get first chunk of input;
        for each word in chunk do:
            add word to master_dictionary;
        receive local dictionary sizes from all child processes;
        for all child processes:
            receive local dictionary contents;
            add them to master_dictionary;
        compute elapsed execution time;
    child processes:
        set local_dict = [];
        get corresponding chunk of input;
        for each word in chunk do:
            add word to local_dict;
        send local dictionary size to master process;
        send local dictionary content to master process;
    exit;
```

3.4 Benchmark on the HPC@UniTrento cluster

Upon running the versions on the cluster provided us some interesting results. Looking at the comparison chart below, we can already see how the sequential solution is always run on a single process and how it compares to the run times of the parallel solutions. This chart was based on an average of results, running on different configurations. 8 cores provided us some delightful results.



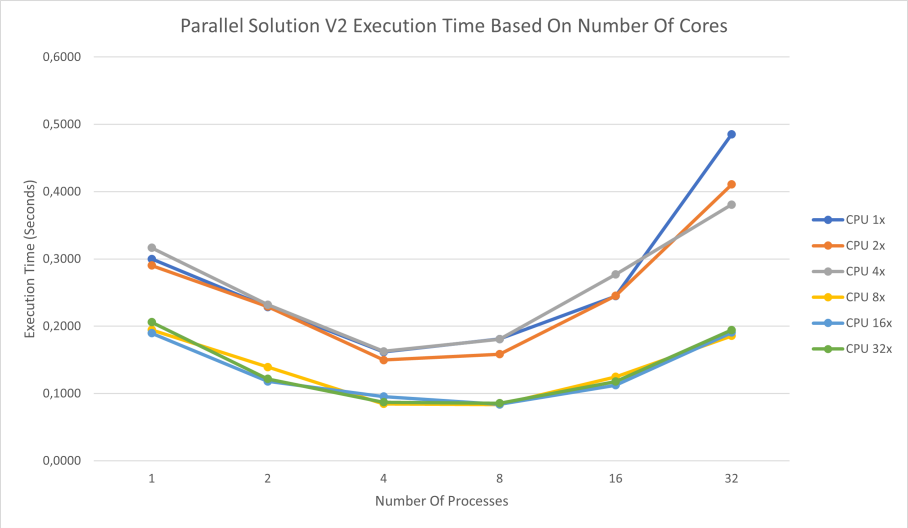
Experiences with the Sequential Solution The sequential solution did not bring a big surprise, it was running with constant speeds, for a small input of a ~1000 lines the run times were:

- 0,251478
- 0,266928
- 0,244632

This results in a rough average of **0,25** seconds.

Experiences with V1 For the first solution, we wanted to see how atomic communications affect the speed on a large scale. Version one of the implementation run on an average speed for a smaller input, similar to V2. On a big inputs (5 books together) however, version 1 got exponentially slower. This is the result of each word requiring a separate communication.
The implementation even got more slower as we increased the number of processes. The reason for this is, as we split the text into more sections, each unique word will generate a record in every process's dictionary, so potentially for a single word, more and more record will be created increasing the number of communications, and the time it takes in the end to merge the dictionaries.

Experiences with V2 The second implementation worked significantly better, as the number of Send - Recv pairs got reduced to the number of processes. This is a version where it can be seen, that upon running with 4 to 8 processes the parallel solution is actually faster then any other. As we further increased the number of processes, the speed started to get slower. With increasing the amount of used processors, this could be compensated, however based on the benchmarks, using 4 to 8 processes, resulted in the best performance.



Optimization experiments During development, we tried other approaches as well. With version 3, we tried to modify the I/O method to read a single line only, so every process could call this method for every line which it had to process. This modification resulted in making the whole application exponentially slower, so we discarded it.
In the case of Version 4, we tried introducing OpenMP to create a parallel for loop in the master process. In theory this would allow us to process the results from the children at them same time and not sequentially. Due to time constraints, we did not succeed with finalizing this optimization.
We had to overcome a memory overflow as well, which was caused by words longer then 25 characters. During processing books, apparently we have to take into consideration the creativity of the writers.

4 Final discussion

Sequential is stable V1 Is not scalable V2 is very nice

References

1. Optimized File Reading Source, https://www.rosettacode.org/wiki/Read_a_specific_line_from_a_file#C. Last accessed 28 Nov. 2021
2. The UTHASH Library, https://troydhanson.github.io/uthash/. Last accessed 13 Dec. 2021