Report Levenshtein Distance
Parallel Programming for Machine Learning project

Chisci Marco
marco.chisci@edu.unifi.it

Abstract

This report explores the process of parallelizing a sequential implementation of the nearest word search algorithm using the Levenshtein Distance, with OpenMP. The report details the sequential and parallel implementations and presents experimental results analyzing execution times, speedups, the effects of different vocabulary sizes, word lengths, and number of thread using both real and generated vocabularies.

1. Introduction

This report aims to illustrate the work I conducted on parallelizing a previously sequential implementation of a simple nearest word search in a vocabulary using the Levenshtein Distance (or Edit Distance), utilizing OpenMP.

The following sections will contain:

- Section 2: the vocabularies used in the experiments
- Section 3: a description of the Levenshtein Distance and the nearest word searching algorithm
- Section 4: an in-depth explanation of both my sequential and parallel implementations of the previously mentioned algorithm.
- Section 5: a thorough report on the experiments conducted to measure the time required to complete the search algorithm under different configurations.
  The main goal of this section is to evaluate the potential speedup achieved by using the parallel version of the search.

2. Used Data

To create large vocabularies containing real words, I combined the vocabularies of the following languages in a single file (all_Languages.txt): English, French, Italian, Danish, Polish, Portuguese, Swedish, German and Spanish. To create even larger vocabularies, all_Languages.txt was duplicated multiple times (e.g. all_Languages2.txt contains all the words of all the previously cited languages twice).
These languages were selected due to their relatively large vocabulary sizes and shared alphabet.
The .txt files of each language was downloaded from: all words in all languages GitHub

To generate vocabularies of varying sizes and word lengths, *generate_random_string* and *generate_random_txt*, in 'generateData.h/cpp', can be used. These functions are able to generate a vocabulary with specified number of random strings. Each string is generated with a specified length by randomly selecting characters from the ranges 0-9, A-Z, and a-z.

Lastly, to evaluate how a vocabulary with numerous similar terms affects the search algorithm, the *generate_similar_txt* function creates a file containing a specified percentage of words that vary by just one character from the target string.

3. Levenshtein Distance

The Levenshtein distance between two strings is the minimum number of single-character edits (insertions, deletions or substitutions) needed to transform one string into the other.
Between two strings, a and b (with lengths $|a|$ and $|b|$, respectively), it is calculated as:

$$lev_{A,B}(i,j) = \begin{cases} |a| \quad \text{if} \quad |b| = 0 \\ |b| \quad \text{if} \quad |a| = 0 \\ lev_{A,B}(i-1,j-1) \quad \text{if} \quad a_i = b_j \\ 1 + min \begin{cases} lev_{A,B}(i-1,j) \\ lev_{A,B}(i,j-1) \quad \text{otherwise} \\ lev_{A,B}(i-1,j-1) \end{cases} \end{cases} \tag{1}$$

This definition creates an $|a|$x$|b|$ matrix where each

		M	O	N	K	E	Y
	0	1	2	3	4	5	6
M	1	0	1	2	3	4	5
O	2	1	0	1	2	3	4
N	3	2	1	0	1	2	3
E	4	3	2	1	1	1	2
Y	5	4	3	2	2	2	1

Figure 1. Edit distance matrix for ”monkey” and ”money” where the costs of insertion, deletion and substitution are each 1

element i, j represents the Levenshtein distance between the substrings a[0, ..., i] and b[0, ..., j].
If one of the strings is empty, the Levenshtein distance equals the length of the other string.
If two characters are identical, the distance remains unchanged.
Otherwise, the distance corresponds to the operation with the lowest cost (deletion, insertion, or substitution).
For example, the Levenshtein distance between ”monkey” and ”money” is 1 (just a deletion) Fig. 1.

I chose to implement a basic nearest word search in a vocabulary using the Levenshtein Distance.
The goal of the algorithm is to identify the word in the vocabulary that is most similar to the input string (e.g., given a misspelled English word, determine the closest match).

4. Code

4.1. Code structure

The files listed below, available in the GitHub repository, will be utilized in the upcoming experiments.

- **LevenshteinDistance.h/cpp**: a straightforward implementation of the algorithm that computes the distance between two words.
- **checkVocabularySeq.h/cpp**: the sequential definition and implementation for determining the closest word (or words) in a vocabulary and its distance from a given input string.
- **checkVocabularyPar.h/cpp**: two parallel versions of the vocabulary check using OpenMP.
- **testLevenshteinDistance.cpp**: the executable file used to compare the performances of the three different methods for checking various vocabularies and input strings.

4.2. LevenshteinDistance.h/cpp

This file includes two functions: *min*, a helper function that returns the smallest of three input integers, and *levenshteinDistance*.

The latter function constructs a table as described in Section 3 and returns the Levenshtein Distance, which is the value of the table’s bottom-right element. The function takes two strings and their lengths as inputs.

4.3. checkVocabularySeq.h/cpp

The only function here is the homonymous function *checkVocabularySeq*, that takes as input a string and a vocabulary (a vector of strings).

As the name suggests, it is a sequential implementation of the nearest word search.

After initializing the min_distance as the maximum possible value, a simple for loops calculates the Levenshtein distance between the input string and each word in the vocabulary.
If a smaller distance is found, min_distance is updated, the vector of nearest words is emptied and the word that was found to be closer to the input string is pushed inside it.
If the distance matches the current min_distance, the current vocabulary word is pushed inside the closest_words vector, allowing multiple words to be considered equally close to the input string.
If neither condition is met, no action is taken.

Finally, the function prints all the words from the vocabulary that are closest to the input string, along with their corresponding distance.

4.4. checkVocabularyPar.h/cpp

This file contains two different functions that implement two parallelized versions of the previously described sequential nearest word search: *checkVocabularyParCrit* and *checkVocabularyParPrivate*.

Both functions maintain the same initialization and printing steps but focus on parallelizing thread access to the for loop that processes the words.

They differ in their approach to synchronize the minimum distance and the vector that contains all the closest words to the targeted string.

checkVocabularyParCrit (Fig. 2) handles synchronization with a flush and an atomic write when updating the minimum distance, and a critical section when pushing

```
    // create the threads, start parallel section
#pragma omp parallel default(none) shared(min_distance, closest_strings, words) private(string_distance) \
firstprivate(string, display) num_threads(threads)
{   //parallelize the for loop
    #pragma omp for
    for(const auto & word : string.const.& : words){
        string_distance = levenshteinDistance( str1: string, length1: (int)string.length(), str2: word, length2: (int)word.length());
        // find the closest string
        // synchronize min_distance and closest_strings in all the threads
        if(string_distance<min_distance){
            #pragma omp flush(min_distance)
            #pragma omp atomic write
            min_distance = string_distance;
            #pragma omp flush(min_distance)
            #pragma omp critical
            {
            closest_strings.clear();
            closest_strings.push_back(word);
            }
        }
            // check if more than one string is as close to the target string
            // synchronize closest_strings in all the threads
        else if (string_distance == min_distance) {
            #pragma omp critical
            closest_strings.push_back(word);
        }
    }
```

Figure 2. checkVocabularyParCrit()

```
    // create the threads, start parallel section
#pragma omp parallel default(none) shared(words, closest_strings, min_distance) private(string_distance) \
    firstprivate(string, display) num_threads(threads)
    {   // each thread has its own local variables to search the closest words to the original string
        std::vector<std::string> local_closest_strings;
        int local_min_distance = std::numeric_limits<int>::max();
        //parallelize the for loop, not necessary to wait all the threads, they can start entering the critical section
        #pragma omp for nowait
        for(const auto & word : string.const.& : words){
            string_distance = levenshteinDistance( str1: string, length1: (int)string.length(), str2: word, length2: (int)word.length());
            // find the closest string
            if(string_distance<local_min_distance){
                local_min_distance = string_distance;
                local_closest_strings.clear();
                local_closest_strings.push_back(word);

            }
                // check if more than one string is as close to the target string
            else if (string_distance == local_min_distance) {
                local_closest_strings.push_back(word);
            }
        }
        // synchronize all the threads, find the real closest strings
        #pragma omp critical
        {
            if(local_min_distance < min_distance){
                min_distance = local_min_distance;
                closest_strings.clear();
                closest_strings.insert( position: closest_strings.end(), first: local_closest_strings.begin(), last: local_closest_strings.end());
            }
            else if(local_min_distance == min_distance){
                closest_strings.insert( position: closest_strings.end(), first: local_closest_strings.begin(), last: local_closest_strings.end());
            }
        }

        // make sure that all the thread compared their local_closest_strings before final print
        #pragma omp barrier
```

Figure 3. checkVocabularyParPrivate()

a new word into the shared closest words vector. This approach is efficient only if the critical section is accessed infrequently compared to the overall search (for instance, when most words in the vocabulary are quite different from the target string).

checkVocabularyParPrivate (Fig. 3) creates private copies of minimum distance and the closest words vector for each thread. Once each thread completes its search over its assigned vocabulary portion, the results collected by every thread are compared, and only the real closest words are pushed into the shared vector. This approach of privatization minimizes the need for synchronization among threads, requiring critical sections only equal to the number of threads.

4.5. testLevenshteinDistance.cpp

This is the executable file used to test the previously described functions.

testLevenshteinDistance calculates the average time,

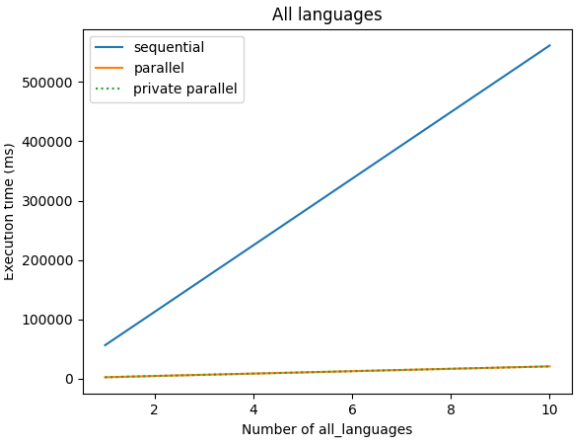

Figure 4. Execution time (ms) required to complete the search

across multiple iterations, that each *check Vocabulary* function requires finding an input string within a vocabulary, as well as the speedups of the parallel versions of the searching algorithm.

The results are stored on a txt file.

If the parallel vocabulary check finds the same closest word(s) and the same distance as the sequential one, given the same input string and vocabulary, it is correct.

5. Experiments and Results

All the following experiments were conducted on the SSH server 'papavero.dinfo.unifi' using CLion as the IDE and g++ as the compiler.

Unless stated otherwise, in each experiment the default number of threads is set to 64, and the target string is "Korrespondenzbankenabteilungen".

5.1. all_Languages.txt

The first experiment involves increasing the size of the vocabulary containing all the languages, as previously explained.

Fig. 4 shows the execution time required to complete the search (sequential, parallel with more critical sections and private parallel) as the vocabulary size grows.
We can see that both parallel versions significantly outperform the sequential version, particularly as the number of all_Languages increases.
Fig. 5 displays the speedup achieved by the two parallelized functions as the size of the vocabulary increases.

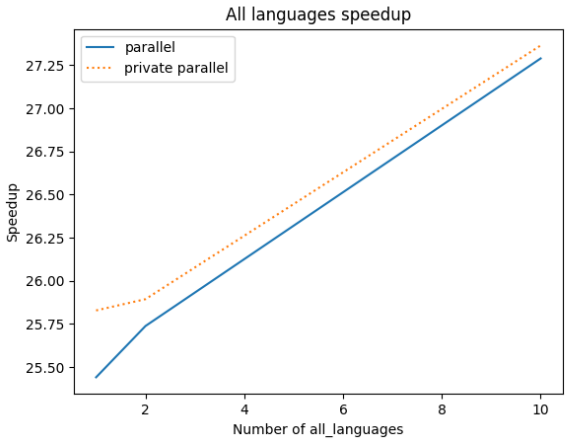

Figure 5. Speedup achieved by the two parallelized functions

	100000	1 Million	10 Millions
Parallel, dim=30	21.0918	22.6528	25.6444
Private, dim=30	24.5439	23.3495	26.1514
Parallel, dim=60	22.1895	22.8898	25.4873
Private, dim=60	23.9347	24.0446	25.5917

Table 1. Speedups obtained by changing the size of the randomly generated vocabulary and the word dimensions)

Once again, it is clear that both parallel versions are much faster than the sequential one, with the lowest speedup being 25.44. The private version of the searching algorithm is always faster than the other, probably due to the less frequent need to synchronize the different threads.

5.2. Different vocabulary sizes and word lengths

The Tab. 1 presents the speedups obtained by changing the size of the randomly generated vocabulary and the word dimensions for the two parallelized functions.

Three main trends are observable:

1. The private parallel version is always faster than the other one.

2. As the vocabulary size grows, so does the speedup. A larger vocabulary reduces the relative impact of thread initialization time on the overall execution, as more time is spent in the parallelized sections of the search algorithm.

3. Increasing word length results in similar speedups. This is likely because doubling the word length doesn't significantly affect execution time, unlike the previous scenario where the vocabulary size increased tenfold.

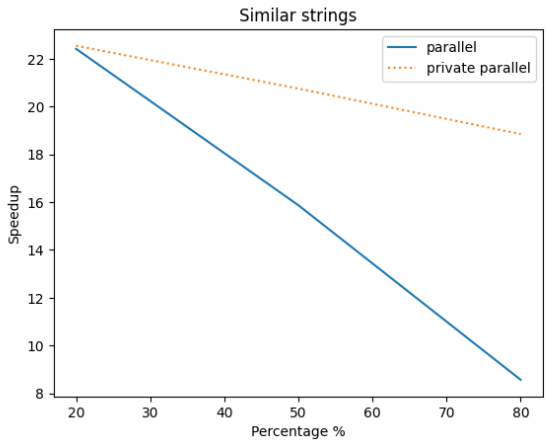

Figure 6. Speedup obtained as the percentage of 10 millions similar long strings (30 characters each) increases

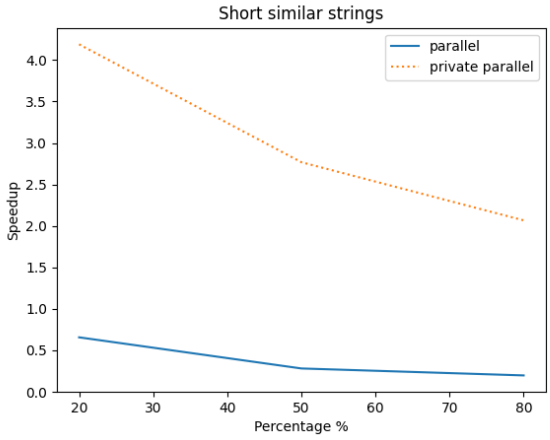

Figure 7. Speedup obtained as the percentage of 100 millions similar short strings (2 characters each) increases

5.3. Vocabularies with many similar words

This subsection examines the impact on the performance of the parallel search algorithm when the vocabulary contains varying percentages of words that are very similar (differing from the target string by only one character).

Fig. 6 shows the speedup obtained by the two parallelized search algorithms as the percentage of similar long strings (30 characters each) increases. As expected, the performance of the parallel version with multiple critical sections is significantly worse than the other one, since every new similar string found triggers a slow critical section.
The private parallel version is also affected by a higher percentage of similar strings due to the increased need for synchronization at the end of each thread's search.

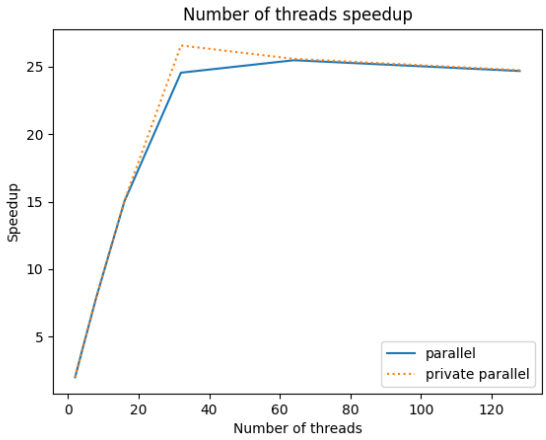

Figure 8

Fig. 7 presents a graph similar to the previous one, but in this case, the strings are only two characters long. As a result, the overall speedup for both algorithms is much lower, as the time needed to compute the Levenshtein distance is almost insignificant.
The parallel version with more critical sections performs even worse than the sequential one, since the execution is basically sequential but with the added overhead of thread creation, management, and synchronization.

5.4. Different number of threads

The final experiment simply changes the number of threads used for searching within a vocabulary of 10 million randomly generated strings, each 60 characters long.

In Fig. 8 we can clearly observe an almost linear increase in speedup as the number of threads increases up to 32. After that peak, the speedup levels off, even with more threads. This suggests that 32 threads are enough to provide enough work for all the CPU cores, and adding more threads only introduces additional overhead for thread creation, management, termination, and synchronization, which negates the benefits of a shorter execution time.