PC-2022/23 midterm: Parallel bigrams and trigrams

Carlo Ceccherelli
E-mail address
carlo.ceccherelli@stud.unifi.it

Abstract

In this project we will see two version of a program that counts the frequencies of bigrams and trigrams of both letters and words in a text file: a sequential version and a parallel version with OpenMP. We will compare the performance of the two versions showing the speedup that can be reached using a parallel computing approach.

1. Introduction

An *ngram* [1] is a sequence of *n* letters or words, *e.g.* in the word "table" the bigrams of letters are "ta", "ab", "bl", "le". The goal of the project is to read one or more text files (retrieved from `https://www.gutenberg.org/`) and compute four dictionaries containing respectively the number of occurrencies of bigrams of letters, trigrams of letters, bigrams of words and trigrams of words.

1.1. Main idea

I decided to split the problem in two parts: counting bigrams and trigrams of letters and counting bigrams and trigrams of words; the first part of the program will iterate over a string containing the whole text in exam while the second part will need an array of strings representing all the words in the text. Once we made this distinction, the algorithms for counting letters or words are basically the same.

The algorithm for counting ngrams of letters (shown in 1) uses two loops, the outer one iterates over the text string while the inner one fills a buffer with n consecutive letters and updates the dictionary.

Algorithm 1

Require: $n =$ number of grams, $text$
  for $i = 0$ **to** $text.length - (n - 1)$ **do**
    $buf =$ ""
    for $j = 0$ **to** n **do**
      $buf.append(text[i + j])$
    end for
    $dict[buf] + +$
  end for

2. Implementation

We will now see the implementation details of both version, poning the focus on the parallelization of the problem using OpenMP and the challenges it brings. In both versions the data structure used to represent the four dictionaries is an `std::unordered_map<string,int>`; in order to compute both bigrams/trigrams of letters and words, we need a string `text`, extracted from some text files in a folder, and an array of strings `words` computed by two functions, one sequential (2) and one parallel (5)

2.1. Sequential version

In the sequential version, i simply follow the algorithm 1 and compute the dictionaries with four for loops, in listing 1 i show only the loops responsible for counting the bigrams of letters and words (the loops for counting

Listing 1 loops for counting letters and words bigrams

```
for(int i = 0; i < textLength-1; i++)
{
    bcBuf = "";
    for(int j = 0; j < 2; j++)
    {
        cc = text[i+j];
        if(isalpha(cc))
            bcBuf += cc;
    }
    if(bcBuf.size() == 2)
        bcFreq[bcBuf]++;
}

for(int i = 0; i < wordsLength-1; i++)
{
    for(int j = 0; j < 2; j++)
    {
        bwBuf[j] = words[i+j];
    }
    bwFreq[bwBuf[0] + " " + bwBuf[1]]++;
}
```

trigrams are basically the same). To compute the `words` vector (in other words to tokenize the text into words) i use the function `sequentialTokenizeWords(text)` showed in listing 2.

2.2. Parallel version with OpenMP

The main idea behind the parallelization of the sequential version is to divide the `text` string and `words` array in chunks to be assigned to threads, once each thread has its own chunk, it can iterate over it and count bigrams and trigrams. Using this approach we have to deal with two pitfalls:

1. concurrent update of the dictionaries by different threads running in parallel
2. counting bigrams/trigrams at chunk boundaries

To handle the first issue, i decided to declare an array of maps and a ”global” map for every task (meaning for computation of bigrams of letters, trigrams of letters ecc.), *e.g.* for bigrams of letters, there will be `bcFreq[nthreads]` which will be reduced (as shown in listing 4) in

Listing 2 sequential tokenization of text in a vector of words

```
vector<string>
↪  sequentialTokenizeWords(string& text)
{
    vector<string> words;
    string word;
    for(char c : text)
    {
        if(isalpha(c) || isspace(c))
        {
            if (isspace(c))
            {
                if(!word.empty())
                {
                    words.push_back(word);
                    word.clear();
                }
            }
            else
            {
                word += c;
            }
        }
    }
    return words;
}
```

`bcFreqReduction`; in this way threads can run in parallel with no need for synchronization or atomics and only in the end merge their local maps into a global one. Notice that the reduction phase can be splitted in four sections (i show only one to save space) because the four maps are independent of each other and can be reduced in parallel.

To handle the second issue the most efficient idea came to mind was to replace the built in `#pragma omp for` with manual assignment of text chunks to threads. Again considering only bigrams of letters, in listing 3 it’s shown this assignment and the loop that counts the bigrams (derived from the pseudocode 1); in this way we can see that threads overlaps in reading the charachters from `text` without missing any bigram and handle the last chunk so as to avoid reading over the text length.

In the parallel version i decided to parallelize also the tokenization of the text in words, so in this version to compute the `words` vector i call

Listing 3 letters bigrams counting

```
#pragma omp parallel default(none)
↪  shared(text, textLength, bcFreq)
{
    int tid = omp_get_thread_num();
    int nth = omp_get_num_threads();
    int chunkc = textLength/nth;
    int bcstart;
    int bcend;
    bcstart = tid*chunkc;
    if(tid == nth-1)
        bcend = textLength-1;
    else
        bcend = (tid+1)*chunkc;

    for(int i = bcstart; i < bcend; i++)
    {
        bcBuf = "";
        for(int j = 0; j < 2; j++)
        {
            cc = text[i+j];
            if(isalpha(cc))
                bcBuf += cc;
        }
        if(bcBuf.size() == 2)
            bcFreq[tid][bcBuf]++;
    }
    ...
    #pragma omp barrier
    //reduction phase
```

Listing 4 reduction phase for letters bigrams

```
    ...
    #pragma omp barrier
    //reduction phase
    for(int i = 0; i < nth; i++)
    {
        #pragma omp sections
        {
            #pragma omp section
            for(auto b : bcFreq[i])
                bcFreqReduction[b.first] +=
↪   b.second;
        }
    }
}//end of parallel region
```

the function `parallelTokenizeWords()` showed in listing 5. In this function i compute, before entering the parallel region, an array `pos` containing the start and end positions of the text that the threads will pick to determine the chunk of text they will be working on; this array of positions is computed to split the text exactly where there is a space, so the threads won't need synchronization during their text chunk tokenization. In the parallel region each thread picks its start and end indexes and evaluates each charachter `c`, filling a buffer `word` if c is valid and appending the word to its private vector `words[tid]` if c is a space; note that in this case, instead of using `push_back()` to append words, a combination of `resize()` and `insert()` is chosen to avoid multiple memory reallocations and to improve performance. Outside the parallel region the private vectors are reduced in a global vector `wordsReduction` which is returned by the function.

3. Tests and results

Finally we will see how the two versions perform and how much speedup the parallel version can bring. Tests are executed on a laptop with 16GB RAM and an i5-1135G7 quad core processor clocked at 2.4GHz. Time measurements in the code are done using `chrono::high_resolution_clock` and include the portion of code relative to tokenization of the text string and computation of bigrams/trigrams of letters/words (the initialization of the text string from reading the files is omitted); the code is measured 10 times for each configuration (number of threads, text dimension) to evaluate the average execution time. We will see the results relative to two code configuration: the first one avoid the use of sections in the reduction phase and uses the sequential tokenization; the second one instead leverage parallelism even in these tasks. In table 1 and Fig 1 are reported execution times and speedup of sequential and parallel versions for the first configuration on different text sizes with 4 and 8 threads; In table 2 and Fig 2 are reported execution times of sequential and parallel versions for the second configuration on different text sizes with 4 and 8 threads. Speedup

Listing 5 parallel tokenization of text in a vector of words

```
vector<string>
↪ parallelTokenizeWords(string& text, int
↪ textLength, int nthreads)
{
    int pos[nthreads+1];
    pos[0] = 0;
    pos[nthreads] = textLength;
    for(int i = 1; i < nthreads; i++)
    {
        pos[i] = i * (textLength/nthreads);
        while(!isspace(text[pos[i]-1]))
            pos[i]++;
    }
    vector<string> wordsReduction;
    vector<string> words[nthreads];
    #pragma omp parallel default(none)
↪   shared(text, pos, words,
↪   wordsReduction)
    {
        int tid = omp_get_thread_num();
        int start = pos[tid];
        int end = pos[tid+1];
        string word;
        int j = 0;
        char c;
        for(int i = start; i < end; i++)
        {
            c = text[i];
            if(isspace(c))
                if(!word.empty())
                {

↪   words[tid].resize(words[tid].size() +
↪   word.size());

↪   words[tid][j].insert(words[tid][j].end(),
↪   word.begin(), word.end());
                    j++;
                    word.clear();
                }
            else if(isalpha(c))
                word += c;
        }
        words[tid].resize(j);
    }
    //reduction phase
    for(int i = 0; i < nthreads; i++)
    {

↪   wordsReduction.resize(wordsReduction.size()
↪   + words[i].size());

↪   wordsReduction.insert(wordsReduction.end(),
↪   words[i].begin(), words[i].end());
    }
    return wordsReduction;
}
```

is evaluated by the formula

$$S = \frac{t_s}{t_p} \tag{1}$$

where t_s indicates the execution time of the sequential version and t_p the execution time of the parallel version.

We can see how the second configuration, which exploit parallelism even in text tokenization and maps reduction, show better execution times i.e. higher speedup.

text size	seq time	par time (4 thr)	par time (8 thr)
1MB	0.37s	0.17s	0.16s
2.14MB	0.75s	0.38s	0.35s
4.42MB	1.79s	0.81s	0.74s
7.07MB	2.78s	1.30s	1.17s
10.27MB	4.12s	1.96s	1.73s

Table 1. Execution times of sequential and parallel version for first configuration (no sections in reduction phase and sequential text tokenization)

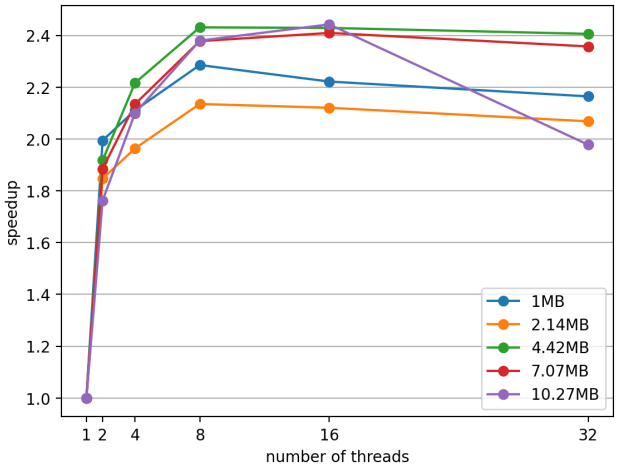

Figure 1. Speedup of the parallel version for first configuration

References

[1] Wikipedia contributors. N-gram — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=N-gram&oldid=1143917543, 2023. [Online; accessed 7-April-2023].

text size	seq time	par time (4 thr)	par time (8 thr)
1MB	0.37s	0.14s	0.12s
2.14MB	0.75s	0.28s	0.27s
4.42MB	1.79s	0.62s	0.60s
7.07MB	2.78s	1.12s	0.96s
10.27MB	4.12s	1.44s	1.36s

Table 2. Execution times of sequential and parallel version for second configuration (with sections in reduction phase and parallel text tokenization)

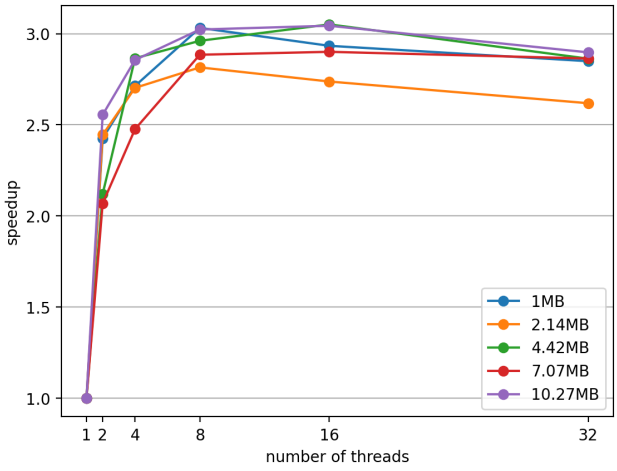

Figure 2. Speedup of the parallel version for second configuration