Parallel bigrams and trigrams

Midterm project for Parallel Computing course

Carlo Ceccherelli

Università degli Studi di Firenze

Table of contents

Introduction

Goal

Write a program that counts the number of **bigrams** and **trigrams** of both **charachters** and **words** in a given text.
Implement two versions:

- Sequential
- Parallel using **OpenMP**

Analyse execution times and *speedup* reachable when applying parallel programming.

N-Grams

An **n-gram** is a contiguous sequence of n *items* from a given text.
An item can be a charachter, a word, a phrase...
In this project: computing the occurrencies of 4 types of grams:

- bigrams of charachters
- trigrams of charachters
- bigrams of words
- trigrams of words

Main idea

Split the program in two parts:

1. counting bigrams/trigrams of charachters
2. counting bigrams/trigrams of words

1° part: iterates over a `string` containing the whole text.
2° part: iterates over a `vector` of words obtained from text string.

Between the two parts: text **tokenization** to generate the vector of words

Main idea

Once made this distinction, the main algorithm is this:

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

Implementation

Sequential version

Implements the **Algorithm 1**, using `unordered_map<string,int>` to store occurrencies of bigrams/trigrams of chars/words.
Tokenize the string `text` filling a buffer `word` and appending it to the vector `words` when encountering a space.

```
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;
```

Parallel version

Implemented with **OpenMP**, a shared memory API for parallel programming.
Includes a set of directives to instruct the compiler on how to parallelize the program.

Main idea is to split the `text` string and `words` vector in chunks to be assigned to threads $\rightarrow$ two pitfalls:

1. counting bigrams/trigrams at chunk boundaries
2. concurrent update of the maps by different threads

Parallel version

1: counting bigrams/trigrams at chunk boundaries

Replace the `#pragma omp for` with manual assignment of text chunks to threads:

- threads overlap in reading text to avoid missing grams located at chunk boundaries
- the last text chunk is handled to avoid reading over the text length

Parallel version

```
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]++; }
}
...//other 3 loops counting other grams
#pragma omp barrier
//reduction phase
```

Parallel version

2: concurrent update of the maps by different threads

For each task (bigrams/trigrams of chars/words) declare outside the parallel region a **global map** and an **array of maps**:

```
unordered_map<string, int> bcFreqReduction;
unordered_map<string, int> bcFreq[nthreads];
```

- each thread `i` counts n-grams on its chunk and updates its "private" map `bcFreq[i]` with **no need for synchronization**
- the array of maps will be reduced in `bcFreqReduction` after threads have synchronized on a **barrier**

Parallel version

The reduction phase can be splitted in four **sections** due to the tasks independence.

```
...
#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;
        //other 3 sections reducing other type of grams
    }
}
```

Parallel version

Parallelize also the **text tokenization**.

Before the parallel region → compute an array `pos` from which threads pick their working chunk, splitting when there's a space.

In the parallel region → each thread `i` fills a buffer `word` with valid chars and appends to its private vector `words[i]` when it finds a space.

After the parallel region → private vectors are reduced inside `wordsReduction`.

```
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]++;
}
```

Parallel version

```
#pragma omp parallel 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);
}
```

Parallel version

Use of `resize()` + `insert()` instead of `push_back()` to avoid multiple memory reallocations

```
...//end of parallel region
//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());
}
```

Tests and results

Tests setup

Tests executed on a laptop with 16GB RAM and an i5-1135G7 quad core processor clocked at 2.4GHz.
Time measurements are done using `chrono::high_resolution_clock`

- execution times are measured 10 for each configuration an then averaged
- a configuration is identified by a number of threads and a text size

Speedup is evaluated as $S = \frac{t_s}{t_p}$ where t_s indicates the execution time of the sequential version and t_p the execution time of the parallel version

Time measurements

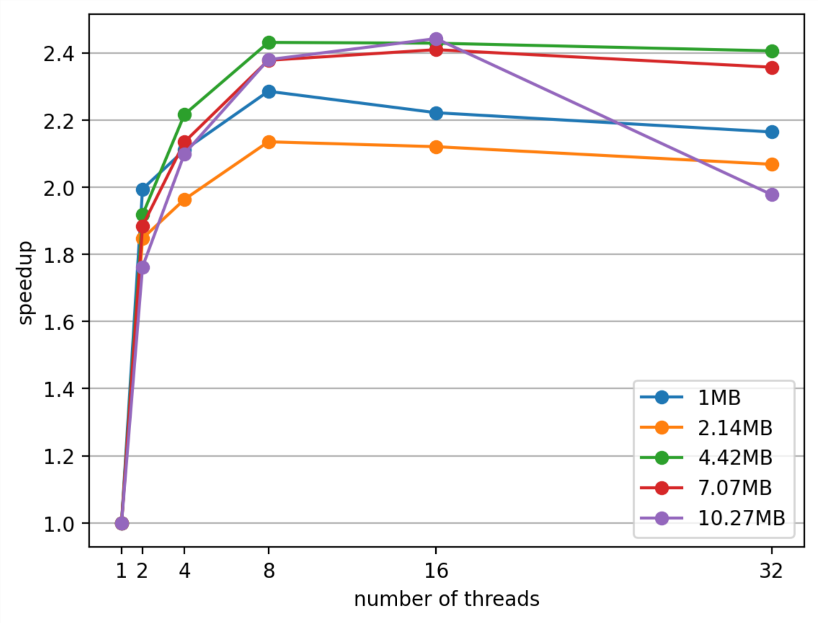
1° test set: no sections in reduction phase and sequential tokenization

2° test set: use of sections in reduction phase and parallel tokenization

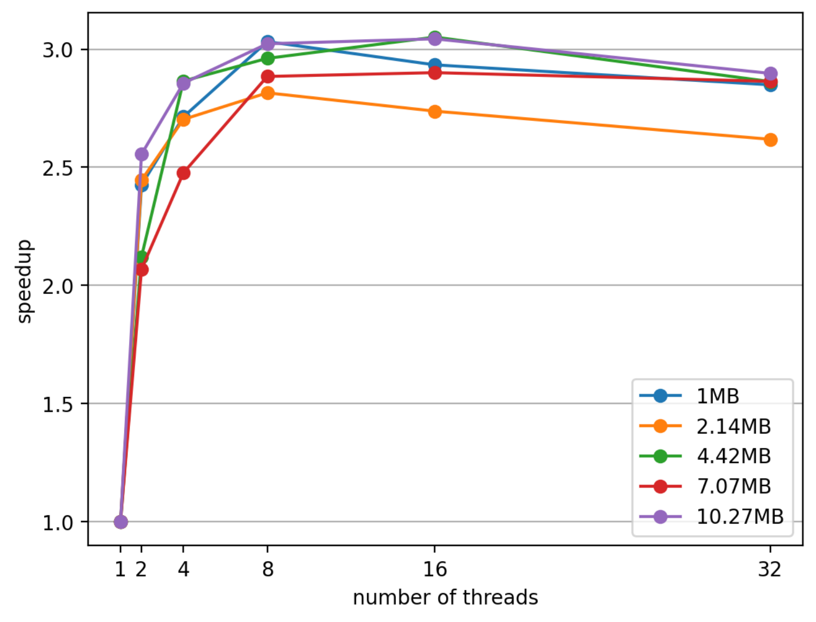
text size	seq time	par time (4 thr)	par time (8 thr)
1MB	0.37s	0.17s	0.16s
2.14MB	0.75s	0.38s	0.35s
4.42MB	1.79s	0.81s	0.74s
7.07MB	2.78s	1.30s	1.17s
10.27MB	4.12s	1.96s	1.73s

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

Speedup - 1st test set



Speedup - 2nd test set



Thank you for your attention!