Parallel programming of n-grams in C

Christian Zanzi
E-mail address
christian.zanzi@edu.unifi.it

Abstract

This report aims to demonstrate how the performance of a program, written to calculate the histogram containing the frequencies with which the n-grams appear inside a text, can improve with the parallelization of the code. In more detail, starting from a sequential program, we will implement parallel programming techniques to speed up the execution, analyzing how the Map-Reduce pattern and the OpenMP directives operate.

Future Distribution Permission

1. Introduction

The detection and the count of n-grams are particularly used in natural language processing. Analyzing the frequency with which a set of words or characters appears in a text can give information on the structure of it or can help to identify recurrent schemes or lexical constructs used by an individual to elaborate a discourse or written essay.
An n-gram is a sub-sequence of n elements belonging to a set. Specifically, a text can be considered as the set and, concerning the application examined, the literals, syllables, or words can be defined as the elements.
The program in this report is written for the detection of bigrams/trigrams of words and characters from a text, counting how many times they arise to obtain the relative histogram.

1.1. Corpus

To obtain the dataset on which we are doing the tests, we have considered the Herman Melville's book Moby Dick downloaded from The project Gutenberg site. The contents equivalent of 20 copies of the book have been used as a way to have a 4 million-word corpus. Then we have removed every special character and punctuation symbol from the dataset, changing every word to lowercase.

1.2. Developement metodologies

C is used as the programming language for the program's code, which is structured in the following phases:

- reading from an external file;
- preprocessing, the removal of punctuation and special characters;
- tokenization of the normalized words;
- detection of word/character n-grams;
- ordering of the n-grams;
- count of equal n-grams.

In this work, each word or each character divided from another one with a punctuation symbol or a space is considered as a single token.
For this problem, the design pattern called Map-Reduce was implemented for its applicability to the resolution phases, combined with the use of the OpenMP library.
OpenMP is an API that contains a set of directives

and can modify the behaviour of the program during runtime, executing parts of the code in parallel.
In addition, we used the SoA (structure of array) paradigm for the data memorization in memory for the purpose of improving the data access on SIMD architectures.
In C, there is no native structure that implements the hash map, useful to decrease the time of search and insert of n-grams, so we preferred to simplify the code using a structure of arrays containing the n-grams and the relative frequencies. In this way, in the parallel section, the allocation of the local structures will be in contiguous memory cells. A future implementation could consider the hash table to limit the time requested for the ordering that takes place in the reduction phase.

1.3. Performance evaluation

The speedup $\frac{t_{seq}}{t_{par}}$ measures the performance improvement obtained from the parallelization of the sequential code. Each program is executed 5 times, and the execution time is recorded with the *omp_get_time()* directive. The parallel program has been timed using 2, 4, 6, and 8 threads to verify that increasing the number of cores enhances the speedup.

1.4. Test environment

The program is runned on
Model name: MacBook Air
Chip: Apple M3
total number of Cores: 8 (prestazioni 4 ed efficienza 4)
Memory: 16 GB
using Clion as IDE.

2. Programs comparison

In this section, we will discuss the difference between the programs and their implementation.

2.1. Sequential program

The sequential program instantiates a struct called Histogram that contains an array for the n-grams and one for the relative frequencies. It begins by reading a text file, and after that, preprocesses it by replacing special and punctuation characters with spaces and bringing each word to lowercase. Successively, through the *tokenize()* function, we divide each token (word), storing them inside an array.
We create the n-grams and we initialize their frequencies using two for loops.

```
for (int i = 0; i < word_count - NGRAM_SIZE + 1;
    i++) {
        histogram->ngrams[i] = strdup(
    create_ngram(words, i));
        histogram->frequencies[i] = 1;
        histogram->size++;
    }

    for (int i = 0; i < word_count; i++) {
        for (int j = 0; j < (int)strlen(words[i])
     - NGRAM_SIZE + 1; j++) {
            char_histogram->ngrams[char_histogram
    ->size] = strdup(create_char_ngram(words[i],
    j));
            char_histogram->frequencies[
    char_histogram->size] = 1;
            char_histogram->size++;
        }
    }
```

Once the n-grams have been created and associated with the unit frequency, the array is ordered using the *qsort()* function. The ordering is needed to count the frequencies of the equal n-grams; indeed, in the *combine()* function, we sum the frequencies of the equal successive n-grams in the array, deleting the duplicates.

```
int combine(Histogram *h1, const Histogram *h2,
    int index) {
   for (int i = 0; i < h2->size; i++) {
      if (index != 0 && strcmp(h2->ngrams[i], h1
    ->ngrams[index - 1]) == 0) {
          h1->frequencies[index - 1] += h2->
    frequencies[i];
      } else {
          h1->ngrams[index] = strdup(h2->ngrams[i
    ]);
          h1->frequencies[index] = h2->frequencies
    [i];
          index++;
      }
   }
   return index;
}
```

The program prints the execution time at the end.

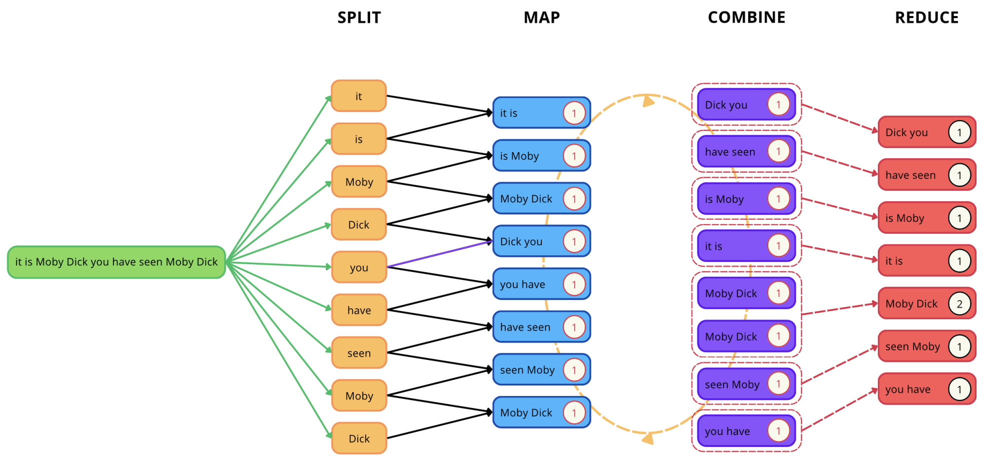

Figure 1. Functioning of the Map-Reduce design pattern in the computation of bigrams.

2.2. Parallel program

The parallel program starts with the same operation of reading and preprocessing as the sequential counterpart. After that, it includes a *#pragma omp parallel* section where the map phase takes place.

```
#pragma omp parallel num_threads(NUM_THREADS)
    default(none) shared(word_ngram,
    character_ngram, local_ngrams, local_c_ngrams
    , words, word_count) firstprivate(
    word_chunk_size, character_chunk_size)
```

With the clause *default(none)*, we manually identify the private and shared variables between the threads in the parallel section, *firstprivate()* defines the private variables that will have an initial value equal to the one held outside the parallel section.

At the beginning of the parallel section, the local histogram structures, which are private for each thread, are instantiated. They will keep track of the n-grams found in the iteration assigned to the specific thread.

```
int tid = omp_get_thread_num();
local_ngrams[tid] = create_histogram(
    word_chunk_size);
local_c_ngrams[tid] = create_histogram(
    character_chunk_size);

#pragma omp for
for (int i = 0; i < word_count - NGRAM_SIZE; i++)
     {
   local_ngrams[tid]->ngrams[local_ngrams[tid]->
    size] = strdup(create_ngram(words, i));
   local_ngrams[tid]->frequencies[local_ngrams[
    tid]->size] = 1;
   local_ngrams[tid]->size++;
}

#pragma omp for
for (int i = 0; i < word_count; i++) {
   for (int j = 0; j < (int)strlen(words[i]) -
    NGRAM_SIZE + 1; j++) {
      local_c_ngrams[tid]->ngrams[local_c_ngrams[
    tid]->size] = strdup(create_char_ngram(words[
    i], j));
      local_c_ngrams[tid]->frequencies[
    local_c_ngrams[tid]->size] = 1;
      local_c_ngrams[tid]->size++;
   }
}
```

We use the same two loops of the sequential version for the mapping, but with the *#pragma omp for* directive, every loop iteration is assigned and executed by a different thread that computes the instructions between the brackets.
After reordering the local result obtained by the map phase with the *qsort_r()* function, we enter the combine phase where the local n-grams histograms of partial results are computed, summing the frequencies of the equal n-grams.

```
int combine(Histogram *h1, const Histogram *h2,
    int index) {
   for (int i = 0; i < h2->size; i++) {
      if (index != 0 && strcmp(h2->ngrams[i], h1
    ->ngrams[index - 1]) == 0) {
         h1->frequencies[index - 1] += h2->
    frequencies[i];
      } else {
         h1->ngrams[index] = strdup(h2->ngrams[i
    ]);
        h1->frequencies[index] = h2->frequencies
    [i];
        index++;
      }
   }
   return index;
}
```

The reduction phase takes place in the critical section *#pragma omp critical* and its role is to take each n-gram of the thread's partial result and put it in the global and final histogram.
In the last phase, we check if the local n-gram is already inside the global structure, summing the frequencies if it already appears, or adding to it otherwise.

```
#pragma omp critical
{
   int index =  0;
   for (int i = 0; i < local_ngrams[tid]->size; i
    ++) {
      if (strcmp(local_ngrams[tid]->ngrams[i],
    word_ngram->ngrams[index]) == 0) {
         word_ngram->frequencies[index] +=
    local_ngrams[tid]->frequencies[i];
         index++;
      }
      else {
         word_ngram->ngrams[word_ngram->size] =
    strdup(local_ngrams[tid]->ngrams[i]);
         word_ngram->frequencies[word_ngram->size
    ] = local_ngrams[tid]->frequencies[i];
         word_ngram->size++;
      }
   }

   int c_index =  0;
   for (int i = 0; i < local_c_ngrams[tid]->size;
     i++) {
      if (strcmp(local_c_ngrams[tid]->ngrams[i],
    character_ngram->ngrams[c_index]) == 0) {
         character_ngram->frequencies[c_index] +=
     local_c_ngrams[tid]->frequencies[i];
         c_index++;
       }
      else {
         character_ngram->ngrams[character_ngram
    ->size] = strdup(local_c_ngrams[tid]->ngrams[
    i]);
         character_ngram->frequencies[
    character_ngram->size] = local_c_ngrams[tid
    ]->frequencies[i];
         character_ngram->size++;
      }
   }
}
```

3. Sperimental results

The execution times of the programs have been collected, considering, in addition to the sequential one, different versions of the parallel program that vary in the number of threads used in the computation. The results are shown in *Table 1*.

Version	Mean	Speedup
Sequential 2gram	3,814749	*
Sequential 3gram	3,939756	*
Parallel 2gram 2-thread	2,211058	1,725305
Parallel 2gram 4-thread	1,513071	2,521196094
Parallel 2gram 6-thread	1,284081	2,970800138
Parallel 2gram 8-thread	1,237904	3,081618581
Parallel 3gram 2-thread	2,254385	1,747596485
Parallel 3gram 4-thread	1,506942	2,614404883
Parallel 3gram 6-thread	1,309119	3,009470795
Parallel 3gram 8-thread	1,219341	3,231053495

Table 1. Speedup results.

We can notice how the time spent in the execution of the parallel version decreases with respect to the sequential version. Increasing the number of threads, the program rapidly computes the histogram. The maximum speedup obtained was not more than 3.5 times. This, following Amdahl's law, in which the fraction of the code that is not parallelizable has an impact, is limited by the work that can be executed in parallel.

4. Conclusion

Amdahl's law states that the speedup achievable from a P processor is $S_P = \frac{P}{(1+(P-1)\cdot f)}$, where f is the fraction of non-parallelizable code. With the speedup obtained from 8 threads in the 3-grams problem, using the inverse function, we can see that $f = 21\%$, which brings a limit to the maximum value that can be reached of 4.75.

The only two phases that have been parallelized are the map and combine, the reduce phase is not parallelizable. The ordering is a bottleneck for high-dimensional data and tends to slow down the process. The hash map can surely improve the performance. For the analysìzed data, the tokenization does not seem a bottleneck, but in future versions, the split phase in which the tokens are found can be made parallel.
In conclusion, we can infer that parallelizing a sequential program for the computation of a histogram of n-grams can improve execution perfor-

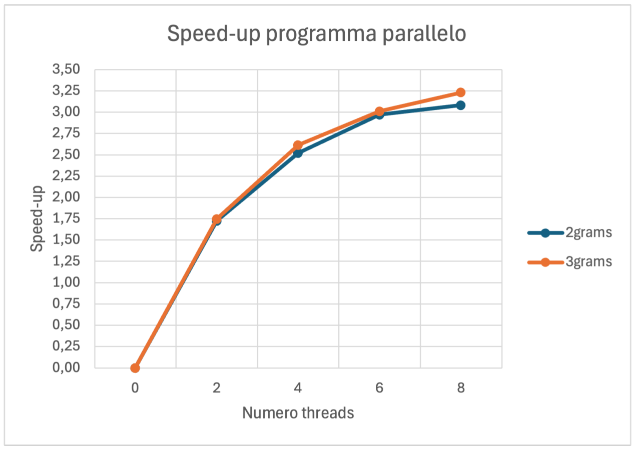

Figure 2. Risultati speedup sottoforma di grafi.

mance.