PC-2022/23 Calcolo parallelo di bigrammi e trigrammi in Java

Alessandro Petricci
7017385
alessandro.petricci@stud.unifi.it

Abstract

An n-gram is a sequence of n contiguous elements in a text. This report will compare the sequential and parallel implementation of a program to compute n-grams with n=2 and n= 3. The programming language used will be Java.

Future Distribution Permission

1. Introduzione

An n-gram is, in general, a sequence of n conscutive symbols or words in a given text. For example, in the string ”Ciao Mondo!”, bigrams of letters (n-grams with n=2) are ”Ci”, ”ia”, ”ao”, ”o ”, ” M”, ”Mo”, ”on”, ”nd”, ”do”, ”o!” while the bigram of words will be the string itself since it’s made up of exactly two words. Statistical analysis of n-grams frequency is commonly used in many fields of study, e.g. cryptography, natural language processing and speech recognition. In the following report we will focus exclusively on bigram and trigrams of letters.

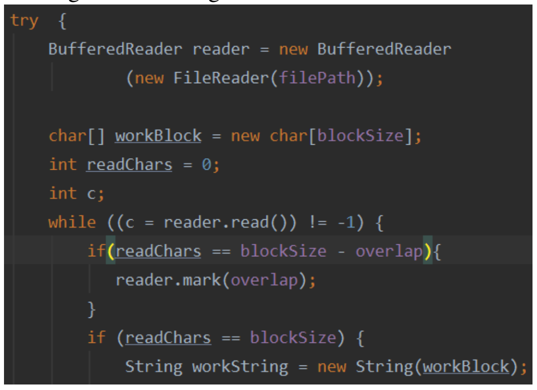
2. Implementation

Both the parallel and sequential version of the program take as input a txt file and return as output a dictionary containing either the bigrams or trigrams as keys and their count in the file as value. Two parameters can be set: one is called n and is the length of the n-gram to be computed, the other one is blockSize and is the number of characters read from the text file at once. A sequential program wouldn’t have problems reading first the entire file and then making computations but in the parallel version of the same program would want to split the data load evenly among all threads, so for consistency both the implementations let the user decide the block size.

2.1. Sequential version

The sequential version of the program uses a BufferedReader to read the file text in blocks of arbitrary length and then compute the n-grams in a cyclic way. This means one can specify how many characters they want to read per cycle. This is particularly useful since reading a file from memory is prone to be a bottleneck for the program performance and changing this parameter gives control on the granularity of the work done. As we will see later, reading few characters per cycle heavily harms the speed of computation. Since reading a text file in blocks would makes us lose the n-grams across two consecutive blocks, a marker is set in the appropriate position so that the pointer of the BufferedReader can be reset back to a position where no n-grams get skipped.

Figure 1. Loading a block of text from the file

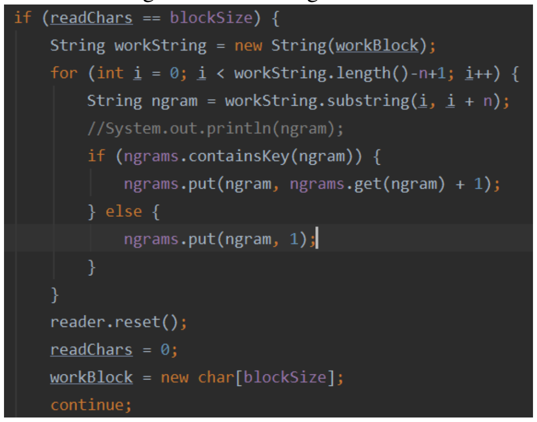


```
try  {
    BufferedReader reader = new BufferedReader
            (new FileReader(filePath));

    char[] workBlock = new char[blockSize];
    int readChars = 0;
    int c;
    while ((c = reader.read()) != -1) {
        if(readChars == blockSize - overlap){
           reader.mark(overlap);
        }
        if (readChars == blockSize) {
            String workString = new String(workBlock);
```

After reading the block of text the program cycles through the characters taking substring of length either 2 or 3 and adding them to an HashMap. At the end of the computation the HashMap contains all the n-grams of said block, the BufferedReader pointer is reset and the thread goes back to reading a new block.

Figure 2. Processing the block



```
if (readChars == blockSize) {
    String workString = new String(workBlock);
    for (int i = 0; i < workString.length()-n+1; i++) {
        String ngram = workString.substring(i, i + n);
        //System.out.println(ngram);
        if (ngrams.containsKey(ngram)) {
            ngrams.put(ngram, ngrams.get(ngram) + 1);
        } else {
            ngrams.put(ngram, 1);
        }
    }
    reader.reset();
    readChars = 0;
    workBlock = new char[blockSize];
    continue;
```

2.2. Parallel version

The parallel version of the program takes a multithreaded approach by using a producer consumer architecture. This decouples reading the file from processing it giving way to better performance.

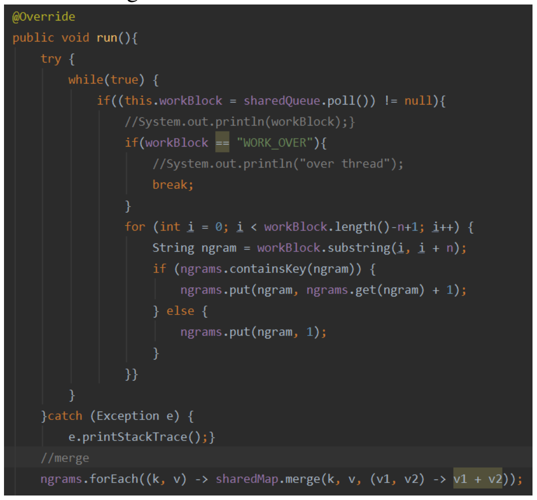
2.2.1 Producer thread

A ConcurrentLinkedQueue is shared among the producer and consumers threads to distribute work to the consumers. The producer, in a way akin to the sequential version of the program, reads a block of text and then enques it in the shared queue, still making sure to mark and reset the reader pointer as to not skip n-grams. This particular queue was chosen as it is a non blocking and thread-safe implementation of Queue. When accesses to the queue are infrequent, we can expect a better performance compared to a blocking queue. We will see how varying the size of the blocks read impacts on the performance of the program. When the whole file has been read the Producer thread enques a poison pill to signal Consumers to stop running.

2.2.2 Consumer thread

An ExecutorService with a FixedThreadPool is used to handle the Consumers. In their run method every thread polls the queue for a block to work on and then processes it in a similar way to the sequential version. There is a difference though in where partial data is saved. Each of the Consumer thread has a local HashMap where they initially save the computed n-grams. Before termination they merge their local HashMap with a shared ConcurrentHashMap. In this way access to a shared data structure is minimized, reducing contention and saving execution time.

Figure 3. Consumer's run method



```
@Override
public void run(){
    try {
        while(true) {
            if((this.workBlock = sharedQueue.poll()) != null){
                //System.out.println(workBlock);}
                if(workBlock == "WORK_OVER"){
                    //System.out.println("over thread");
                    break;
                }
                for (int i = 0; i < workBlock.length()-n+1; i++) {
                    String ngram = workBlock.substring(i, i + n);
                    if (ngrams.containsKey(ngram)) {
                        ngrams.put(ngram, ngrams.get(ngram) + 1);
                    } else {
                        ngrams.put(ngram, 1);
                    }
                }}
        }
    }catch (Exception e) {
        e.printStackTrace();}
    //merge
    ngrams.forEach((k, v) -> sharedMap.merge(k, v, (v1, v2) -> v1 + v2));
```

3. Considerations

1. The number of threads assigned to the tasks are 1 thread for the Producer and 8 threads for the Consumers. Since there's only one file to be read from there's no advantage in having more than 1 thread sharing access to the file to read it. The number of Consumers thread is instead machine dependent. A call to Runtime.getRuntime().availableProcessors() gives us the number of available processors. On the particular machine we run the program on there are 8 logical processors.

2. As said before, work sharing beetween Consumers and Producer is achieved through a non blocking queue. With big enough blocks to compute the queue is accessed in an infrequent manner so that there is minimal contention. In the next section we will see how changing the size of the blocks changes the performance of the program.

3. Each Consumer thread has its own private HashMap.This choiche reduces contention as there are far less access to the shared map.

4. Tests and performance

To test the application random txt files of increasingly bigger size have been generated on this site. The program will be tested on computing bigrams with different sized files and different sized blocks read from those file at once, both in the sequential and parallel version. Every result is averaged on 10 different computations. Tests were done on an Intel i5-8250U, a 4 core 8 threads processor. As we expected performance improves with block size for the parallel version of the program. A block size of 2 has dramatic effects on the run time, making the parallel version of the program even worse than its sequential version. With 500 or more characters for block we reach a plateau where run times are kinda similar. This happens because the shared queue gets polled very frequently for too little jobs and threads have to wait for their turn most of the time. Finding the right granularity for the specific computation is essential in parallel computing. Furthermore, even the sequential version timings are harmed by smaller blocks, even if not as much as the parallel version.

Table 1. Block size: 2 characters

	1MB	2MB	4MB	8MB	16MB
Sequential	0.243	0.431	0.820	1.499	2.705
Parallel	0.661	1.051	2.305	4.316	8.406
Speedup	0.367	0.410	0.355	0.347	0.321

Table 2. Block size: 50 characters

	1MB	2MB	4MB	8MB	16MB
Sequential	0.169	0.326	0.621	1.245	2.156
Parallel	0.125	0.171	0.257	0.441	0.773
Speedup	1.352	1.906	2.416	2.823	2.789

Table 3. Block size: 500 characters

	1MB	2MB	4MB	8MB	16MB
Sequential	0.186	0.372	0.673	1.238	2.100
Parallel	0.135	0.162	0.235	0.393	0.713
Speedup	1.377	2.296	2.863	3.150	2.945

Table 4. Block size: 5000 characters

	1MB	2MB	4MB	8MB	16MB
Sequential	0.178	0.337	0.592	1.109	2.133
Parallel	0.099	0.141	0.223	0.388	0.679
Speedup	1.797	2.390	2.654	2.858	3.141

Table 5. Block size: 10000 characters

	1MB	2MB	4MB	8MB	16MB
Sequential	0.208	0.341	0.662	1.112	2.126
Parallel	0.104	0.148	0.232	0.391	0.659
Speedup	2.000	2.304	2.853	2.843	3.226

For completeness here are the performance result of computing trigrams with a 500 characters block size. From results shown we can see how the best speedup reached is a little less than 3.3, with a noticeable dip down calculating trigrams. This could be related to the load put on the HashMaps having to store around 52^3 elements (26 letters, 52 because trigrams are case sensitive). Speedup still goes up with a bigger workload, since this means there are more jobs to parallelize.

Table 6. Block size: 500 characters

	1MB	2MB	4MB	8MB	16MB
Sequential	0.455	0.902	1.651	2.785	5.384
Parallel	0.31	0.485	0.816	1.261	2.256
Speedup	1.467	1.859	2.023	2.20	2.386