PC-2019/20

Niccolò Corsani
E-mail address
niccolo.corsani@stud.unifi.it

Abstract

The bigrams or n-grams are 2 or n sequences of letters adjacent to each other. The purpose of the software is to count the bigrams of texts, so as to make them a statistic. In fact, the software was designed in a linguistic perspective. The objective, in fact, is not only to count the bigrams, but to make sure that they are counted in the right way: not counting for example the bigrams that have a space in the middle, or where there are not useful characters like ".,:".

Future Distribution Permission

1. Introduction

The realized project has been tested on two different technologies: ”java” and ”c”. In both software, the producer-consumer paradigm was used. The producer aims to produce the right data for the consumer. Specifically, once a text file is obtained, the producer will divide it in to as many strings as many consumers there are. Finally, it synchronizes itself with the other consumers, to generate an initial parallelism between producer and consumer. In both the c and java implementations the producer is one thread, but it can be extended to a greater number of threads if necessary. However, the choice of only one producer is due to the low amount of data that this has to compute. Consumers perform computationally more expensive actions and therefore it has been chosen to perform them on multiple threads. The computer on which the software has been tested has 4 cores moreover the times where each Thread does not compute are short. So the best choice to make maximum use of the CPU can be 6 or 7 Threads or a number a litle bit higher than the number of ”core”.

This is a brief description of the behavior of the two implementations. The following goes to analyze the software code in detail.

2. implementation java

The java project was made with two different synchronization tools:

- Synchronized blocks , wait, notify.

-ConcurrentHashMap

The performances of the two cases were then tested.

2.1. Main

The main class takes care of managing the right calls of the FileUtility class, of starting the producer at the right time and of creating more consumer threads.

```
public static void main(String[] args)
    throws InterruptedException,
    IOException {

  InteractionUser iu = new
      InteractionUser();
```

```
String article =
    InteractionUser.readArticle("File");
int numberOfConsumers = 4;
List sharedListofSubstrings = new
    ArrayList<String>();
Consumer[] consumers = new
    Consumer[numberOfConsumers];
Producer producer = new
    Producer(article,
    numberOfConsumers,
    sharedListofSubstrings);

ConcurrentHashMap
    concurrentHashMapParallelo = new
    ConcurrentHashMap<String,
    AtomicInteger>();
ConcurrentHashMap
    concurrentHashMapSequenziale = new
    ConcurrentHashMap<String,
    Integer>();
ConcurrentHashMap
    concurrentHashMapParalleloTrigrams
    = new ConcurrentHashMap<String,
    AtomicInteger>();
ConcurrentHashMap
    concurrentHashMapSequenzialeTrigrams
    = new ConcurrentHashMap<String,
    Integer>();
producer.run();


Counter counterParallel = new
    Counter();
Counter counterSequential = new
    Counter();
counterSequential.setCounter(-1);

long inizioParallelo =
    System.currentTimeMillis();
for (int i = 0; i < numberOfConsumers;
    i++) {
  consumers[i] = new
      Consumer(sharedListofSubstrings,
      concurrentHashMapParallelo,
       concurrentHashMapParalleloTrigrams,
           inizioParallelo,
           counterParallel,
           numberOfConsumers);
  consumers[i].start();

}
producer.join();

for(int i = 0; i<numberOfConsumers;i++)
consumers[i].join();
```

In the exposed code the instances of ”ConcurrentHashMap” are observed, however the logic is the same also in the implementation with the ”HashMap”.

2.2. Producer

The main purpose of the producer is to divide the article into equal parts. To do this it implements a simple text division algorithm. Another important responsibility is to ensure the mutual exclusion of sharedListofSubstrings.

```
private void produceSubSequences(String
    subString) throws
    InterruptedException {
  {
    synchronized
        (sharedListofSubstrings) {
      sharedListofSubstrings.add(subString);
      sharedListofSubstrings.notifyAll();
    }
  }
}
  @Override
public void run() {

  this.DivideArticle();

}
public String[] DivideArticle() {
```

The produceSubSequences (subString) function has a ”synchronized” block inside. The purpose of the block is, in fact, to synchronize itself with the other consumers. Whenever the producer produces wake up consumers of the queue with notifyAll, then they will immediately compite for the string produced.

2.3. Consumer

As already said, the most computationally burdensome part is that of consumers. These perform the following actions sequentially:

-They take the substring from the list created by the producer.

-The bigrams and trigrams are calculated.

-They merge their respective maps into a single shared map.

In the implementation of what has been said, it has always been thought of reducing synchronized computations to a minimum and increasing parallel phases as much as possible.

In the image below the first of the 2 synchronizations is shown: the one with the producer.

```
    synchronized (sharedListofSubstrings) {

      try {
        while
            (sharedListofSubstrings.get(0)
        .equals(null)) {
          System.out.println("Thread: " +
              this.getName() + "is
              waiting for the producer to
              produce.");
          sharedListofSubstrings.wait();
          try {
            sharedListofSubstrings.wait();
          } catch (InterruptedException
              e) {
            e.printStackTrace();
          }

        }
      } catch (IndexOutOfBoundsException e)
```

For the calculation of the bigrams and trigrams there is a control so that they are all part of the same word: for example the string ”milk bread” will evaluate ”mi”, ”il”, ”lk”, ”br”, ”re” , ”ea”, ”ad”, but not ”kb” as it belongs to two different words. This makes the software more appropriate for a real comparison of how are grammatically different two languages.

In the image below there is the second synchronization, the one between consumers. In this case it can be observed that the concurrentHashMap manages everything. For mutual exclusion to take place, it is important to use only the methods provided by the ”java.util.concurrent.ConcurrentHashMap” package (putIfAbsent, merge).

```
        while ((bigram1 =
            subArticle.substring(counterBigram1
            + 0, counterBigram1 +
            2).toLowerCase())
             .contains(" ")) {
          counterBigram1++; //// procede
              a bigramma successivo
              saltando spazzi
        }

        if (concurrentHashMapBigrams.
        putIfAbsent(bigram1, 1) != null)
          concurrentHashMapBigrams
          .merge(bigram1, 1,
              Integer::sum);
```

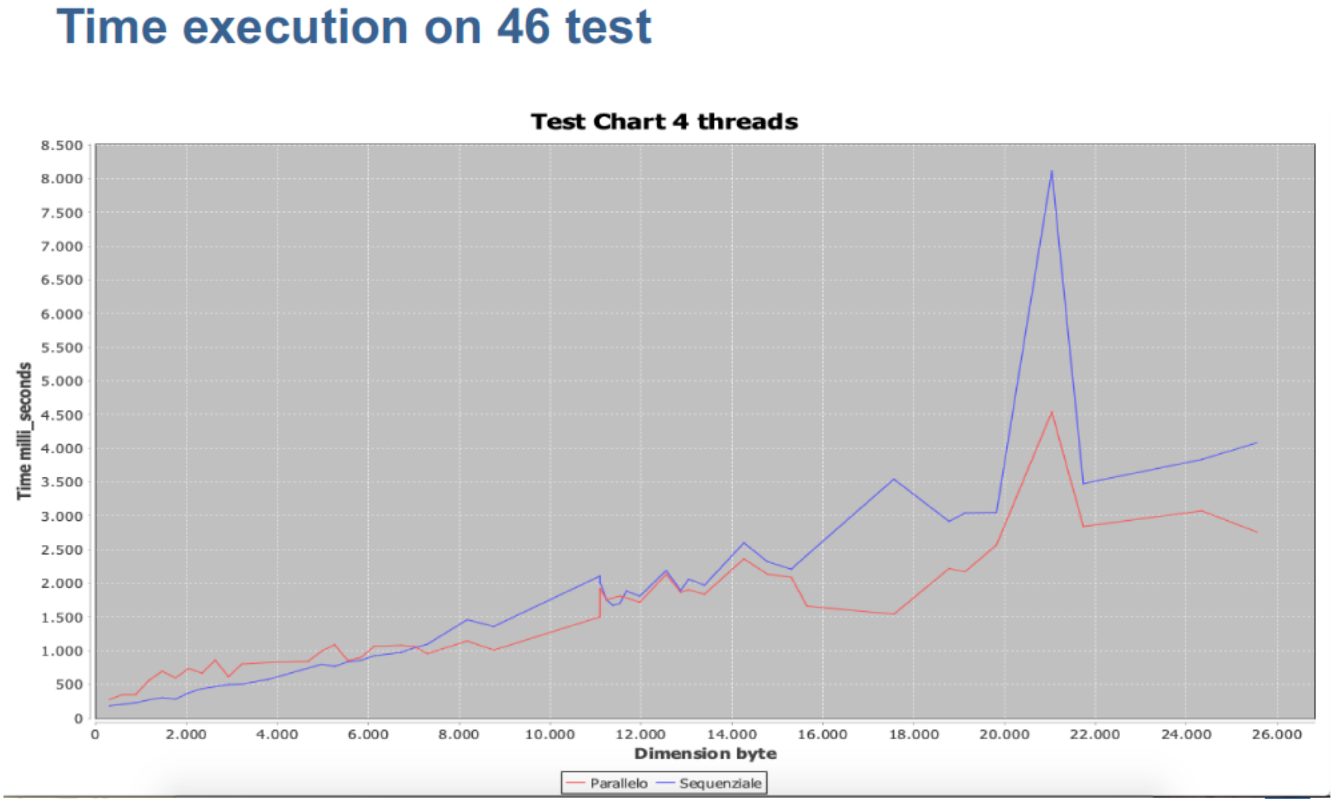
As regards the implementation without the ”Concurrent package”, mutual exclusion will be guaranteed by placing a ”Synchronized” block.

2.4. Conclusion

Before talking about the conclusions that emerged following the tests, a small note how these were made. Since the tests performed are many, it was thought to make this phase as automatic as possible. To achieve this, jars found online were added. Those that allowed to plot the loaded data in the form of a text file. These files were automatically created each time the software terminate the execution, noting the size of the analyzed file and the respective completion time of the process.

```
    File file = new
        File("FileIntero.txt");
    float byte_number =
        file.length()/1024;
    String dim =
        String.valueOf(byte_number);
    iu.append_Article(dim,time);
```

The image below shows the ”Execution Time-Dimension” graph.



Specifically, it is possible to observe the sequential and parallel execution speed as the size changes. The result shows a progressive decrease of the parallel case compared to the sequential one, the bigger the analyzed file. The test cases also covered files in the order of hundreds of megabytes. In this regard, it is useful to remember that the JVM provides a small memory heap and that prevents the software from running. To solve this problem, however, just put the ”-xmx1g” command before compilation, so that the heap is increased to 1 Gigabyte.

3. Implementation c

The software created in c starts from the data created by the Java application previously described. In fact, it is the Java program that takes care of saving the data in the correct folder for analysis. This choice will thus reduce the tasks of the new producer implemented in c. As for the tasks implemented in the consumer, these are practically the same as the software described before, the explanation of this will be more shorter.

3.1. Main

The main class takes care, as before, of calling the various file reading functions, of creating ”pthread” instances and managing them. Below the code with the most important points in the Main.

```
if (pthread_create(&threadId1, NULL,
    calculateBigrams, (void*) article1))
     {
  fprintf(stderr, "Error creating
      thread\n");
  return 1;
}
pthread_create(&threadId2, NULL,
    calculateBigrams, (void*) article2);
pthread_create(&threadId3, NULL,
    calculateBigrams, (void*) article3);
pthread_create(&threadId4, NULL,
    calculateBigrams, (void*) article4);
pthread_join(threadId1, NULL);
pthread_join(threadId2, NULL);
pthread_join(threadId3, NULL);
pthread_join(threadId4, NULL);
```

In order for threads to work, it is important to put ”-pthread” before running the ”GCC” compiler.

3.2. Producer

As already mentioned, the Producer aims to generate useful data for consumers. This, through an algorithm, first obtains the names of the files containing the previously created articles, then opens the respective files and saves them in a char pointer, one for each consumer.

```
void putFileInbuffer(char *article, int n)
    {
  DIR *dir;
  char *articlesName[30];
  int i = 0;
  int j = 0;
  int numberOfArticles = 0;
  struct dirent *ent;
  if ((dir = opendir("./src/Articles")) !=
      NULL) {
    /* print all the files and directories
        within directory */
    while ((ent = readdir(dir)) != NULL) {
      if (strcmp(ent->d_name, ".") &&
          ent->d_name != NULL) {
        if (strcmp("......", ent->d_name)
            == 1)
          continue;
        articlesName[i] =
            malloc(strlen(ent->d_name));
        strcpy(articlesName[i],
            ent->d_name);
        numberOfArticles++;
        i++;
        j++;
      }
    }
    closedir(dir);
  } else {
    int errnum = errno;
    fprintf(stderr, "Value of errno:
        %d\n", errno);
    perror("");
  }

  char *nameArticle;
  nameArticle = strcat("./src/Articles/",
      articlesName[n]);
  long length;
  FILE *f = fopen(nameArticle, "rb");

  if (f) {
    fseek(f, 0, SEEK_END);
    length = ftell(f);
    fseek(f, 0, SEEK_SET);
    fread(article, 1, length, f);
    fclose(f);

  }
  else{

    printf("\nErrore apertura file\n");
    perror("");
  }
```

Finally, the producer provides some utility functions such as: Print the list of items found on the files, support for string management. Since the producer carries out a light task computationally it was thought not to parallel it.

3.3. Consumer

The functional behavior of the consumer is like that in java. For the management of synchronization mutex semaphores were used.

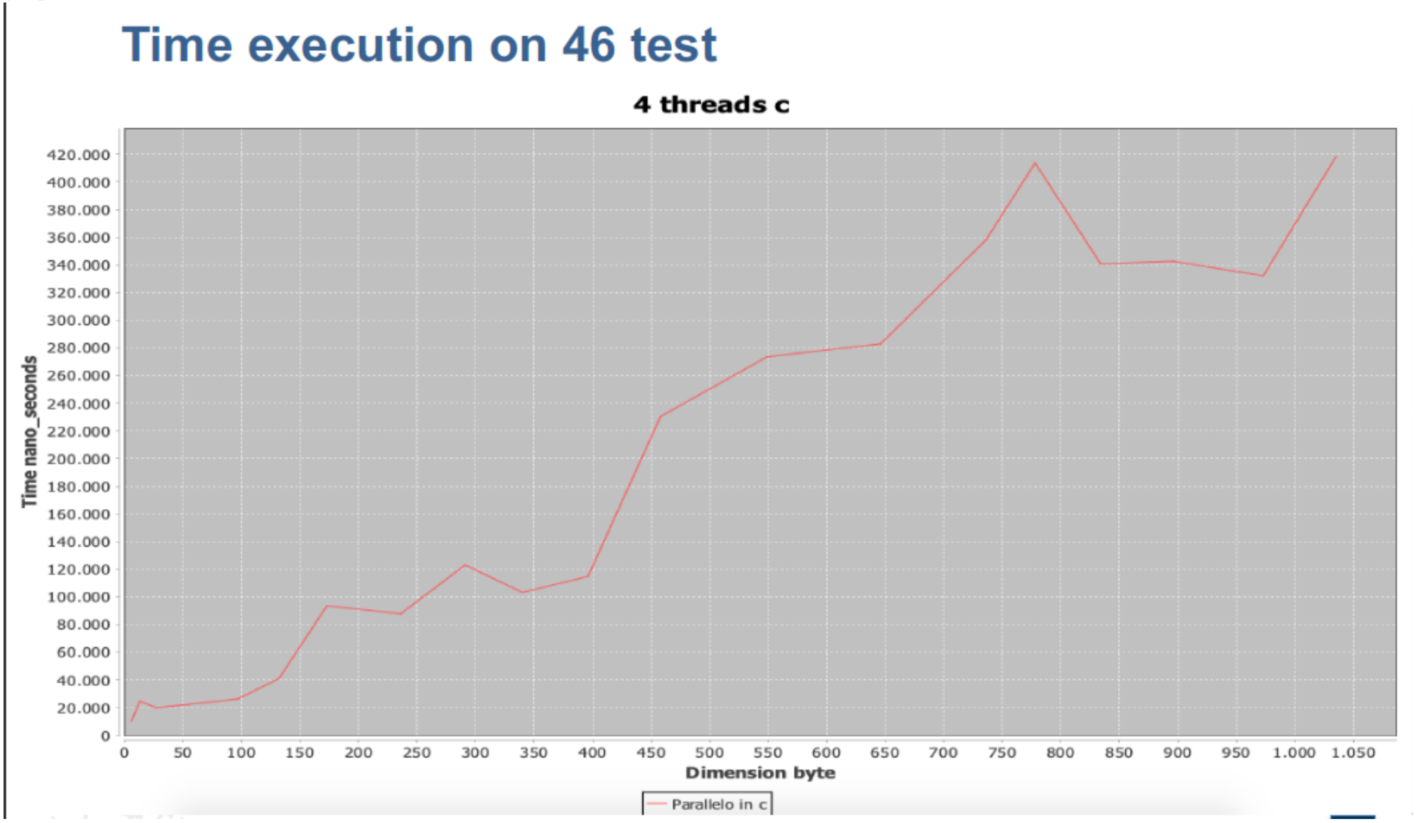
```
    pthread_mutex_lock(&mutex);
    // critical section
    if (UpdateIfPresent(bigram[0],
        bigram[1]) == 0) {
      ///// return 0 if not present
      strcpy(mapBigrams[counterMap].bigram,
          bigram);
      mapBigrams[counterMap].occurences =
          1;
      counterMap++;
      contatoreVerifica++;
    }
         // critical section
    pthread_mutex_unlock(&mutex);
```

The code shows how synchronization occurs between consumers when they check and possibly add a bigram to the map.

3.4. Conclusion

As a conclusion of the project it is reported a graph of the speed of execution.

The image below shows the ”Execution Time-Dimension” graph.



Time execution on 46 test