Sequential and Parallel implementation in JAVA of N-grams of words and characters

Mihail Teodor Gurzu
7060245
mihail.gurzu@stud.unifi.it

Abstract

This report describes the implementation of an application for computing and estimating N-grams in a set of text files, using the JAVA thread programming model for the parallel version. Are presented the results of a series of tests on the two implementations showing the differences between them in terms of computation time and speedup at vary of the number of files and threads for the computation of bigrams, trigrams, 5-grams and 10-grams of words and characters.

Future Distribution Permission

The author(s) of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.

1. Introduction

In the fields of computational linguistics and probability, an N-gram is a contiguous sequence of *N* items from a given sample of text or speech. The items can be phonemes, syllables, letters, words or base pairs according to the application. The N-grams typically are collected from a text or speech corpus [wikipedia].

2. Implementation

The application takes in input a set of files from a directory specified by the user and prepares the files one by one for the computation. This pre-process operation is common to both sequential and parallel implementation.

2.1. Pre-processing

First, the application requests the user to specify the directory containing all the files to be processed and stores the paths of all files contained in the directory in a linked list using the method *loadDatasets()*:

```
public static void loadDatasets(
        LinkedList<String> txtListMain,
        String directory)
```

The files paths are stored in *txtListMain* and then passed to the method *process_txt()* which prepares the data for the computation of words or characters:

```
public static char[] process_txt(
                String txt, String mode)
```

process_txt() reads the entire file and builds the *char[]* array for the computation replacing all the invalid characters. The method uses a *string* library method to accomplish it's task:

- *replaceAll()*: changes the specified invalid characters to a specific character

mode parameter specifies the computation of words or characters N-grams (if *mode = ”word”* the computation is intended for word N-grams, if *mode != ”word”* the computation is intended for characters N-grams):

- for *words*, the method objective is to separate the words by a point ’.’ such that the method for words N-gram computation is able to distinguish the words to process.

- for *characters*, the method objective is to eliminate all the invalid characters creating an array of subsequent characters.

This distinction is necessary since it is not possible to know in advance the length of a word. The resulting *char* array contains all the valid characters for computation, lower-case.

2.2. Sequential implementation

2.2.1 Data structure

The data structure for storing the N-grams is a *HashMap <String, Integer>* which stores the N-gram string and the relative number of occurrences in the input files. This data structure is suitable for the application since it provides constant time operations for input/output.

2.2.2 Computation

There have been implemented two methods, one for word computation and one for characters computation:

- *compute_chars()* takes in input the *char* array of subsequent characters and the *HashMap* to store the calculated N-grams and returns the final *HashMap*.

```
public static HashMap<String, Integer>
    compute_chars(char[] fileString,
    HashMap<String, Integer> hashMap,
    int N)
```

- *compute_words* takes in input the *char* array containing the words separated by '.' and the *HashMap* to store the N-grams, which will be returned by the method.

```
public static HashMap<String, Integer>
    compute_words(char[] fileString,
    HashMap<String, Integer> hashMap,
    int N)
```

We can note that the signatures differ only in the name of the method, but actually the *char* array is constructed differently in the preprocessing as described in section 2.1. Of course, the implementation of the two methods is very different since one has to compute with *items* of a fixed length (characters) and the other one with items of vary length (words).

2.3. Parallel implementation

The parallel version of the application is implemented using the JAVA thread programming model, dividing the data of each file in as many parts as the thread instances.

The implementation makes use of the *java.thread.concurrent* package providing the following features:

- *Future:* represents the result of an asynchronous operation.
- *Executor:* represents an object which executes provided tasks.
- *Executor Service:* is a complete solution for asynchronous processing. It manages an in-memory queue and schedules submitted tasks based on thread availability.

2.3.1 Data structure

The data structures used in the parallel implementation are:

- *HashMap<String, Integer>*, used as a private dictionary by each thread instance.
- *ConcurrentHashMap<String, Integer>*, representing the final dictionary which will contain the merge of each future representing each thread's instance private *HashMap*.

Since the *"merge"* operation is parallelized, a non-blocking data structure allowing concurrent write operations is needed. Hence the choice of *ConcurrentHashMap* since this data structure locks only the index which is being modified.

2.3.2 Computation

Before starting to iterate over the files in input, the application defines a *fixed thread pool* of the number of threads specified by the user and an array of *Futures*.

```
ArrayList<Future> futuresArray = new
    ArrayList<>();

ExecutorService executor = Executors.
    newFixedThreadPool(NUM_THREADS);
```

then, for each file, the executor submits *NUM_THREADS* threads as follows:

```
for (int i = 0; i < NUM_THREADS; i++)
{
   if(i == NUM_THREADS - 1)
      stop = fileLen -1;
   else
      stop = ((i + 1) * k) + (N - 1) - 1;
   Future f = executor.submit(new
       Par_thread(”t” + i, i * k, stop,
       file, N, MODE));
   futuresArray.add(f);
}
```

We can note that each *Par_thread* will return a *Future* for each processed file which is added in *futuresArray* for later retrieval of the results, once all the files have been processed:

```
for (Future<HashMap<String, Integer>> f :
    futuresArray)
{
   HashMap<String, Integer> tmp_dict = f.
       get();
   executor.execute(new merge_thread(
       tmp_dict, dict));
}
awaitTerminationAfterShutdown(executor);
```

Once all the files have been processed and all *Par_thread* have finished their computation, *futuresArray* will contain as many *HashMap* dictionaries as the (number of threads) * (number of files). Finally, the application merges these dictionaries using the *merge_thread* threads, in the *ConcurrentHashMap* global dictionary.

We have seen that there are two type of threads, *Par_thread* and *merge_thread*, and the way they are called in action is different too:

- *Par_thread* is called by *executor.submit* and returns a *Future*, thus it implements the *Callable* interface:

  ```
  public class Par_thread implements
      Callable<HashMap<String, Integer>>
  ```

- *merge_thread* is called by *executor.execute* and does not return anything since it's task is to merge the *Future* returned by a *Par_thread*. It implements the *Runnable* interface:

  ```
  public class merge_thread implements
      Runnable
  ```

Sequential merge for parallel version

There's also the possibility to configure the application to process the merge of the futures sequentially. This can be done by setting a variable accordingly:

```
for (Future<HashMap<String, Integer>> f :
    futuresArray) {
  HashMap<String, Integer> tmp_dict = f.get
      ();

  if(MERGE.equals(”PAR”))
  {
    executor.execute(new merge_thread(
        tmp_dict, dict));
  }
  else
    HashMerge(tmp_dict, dict);
}
```

Tests include the comparison between the parallel and sequential merge process.

No futures version

During testing process, it came out that the parallel version of the application using futures is very memory demanding. In fact, as we will see in the results of tests, using many files to compute a N-gram with N sufficiently big in relation with total input size (inversely proportional) the application runs out of *heap memory*. A solution is to increase the memory reserved for the *JVM*.

However it has been implemented another parallel version of the application, not requiring the use of futures. This version of the application uses a single global *ConcurrentHashMap <String, Integer>*. Instead of returning a future, the threads proceed with computing the N-grams, updating the global dictionary concurrently. Hence there's no need of using futures, these threads implement the runnable interface.

3. Test

Tests have been run on a:

- *Intel(R) Core(TM) i7-3632QM 2.20GHz, quad-core machine (8 logical processors)*
- *8 GB DDR3 1600 MHz RAM memory*

All versions and configurations of the application have been tested on datasets composed of *.txt* files from *gutemberg.org*. There have been used three datasets, respectively of 10 files (15.5 MB), 15 files (21.8 MB) and 42 files (45.4 MB). For each of above dataset the application has been run to compute *bigrams, trigrams, 5-grams* and *10-grams*, using *2, 4* and *8 threads*. Each combination above has been run for computing *word n-grams* and *character n-grams*:

```
public class test
{
  public static void main(String[] args)
  {
    String[] dirList = {”texts10”, ”texts15
        ”, ”texts42”};
    int[] N_gram = {2, 3, 5, 10};
    int[] num_threads = {2, 4, 8};

    for (String dir : dirList)
    {
      for(int n_gram : N_gram)
      {
        //compute SEQUENTIAL
        for(int th : num_threads)
        {
          //compute PARALLEL
        }
      }
    }
  }
}
```

4. Results

Here will be presented the results of the tests (averaged on 10 iterations). The results are divided in two sections, one for computing *word N-grams* and one for computing *character N-grams*. Each section contains the comparison between the sequential and parallel version of the application in terms of computational time and speedup, for each parallel implementation:

- futures and sequential merge
- futures and parallel merge
- no futures

4.1. Word n-grams

Before looking at the results, is mandatory to say that the parallel version of the *word n-gram computation* code has an extra initial part than the sequential version, since the application has to manage the indexes of all threads involved. In particular is shown below the initial *start* and *stop* index manipulation of threads in *compute_words()* method:

```
public HashMap<String, Integer>
    compute_words(char[] fileString, HashMap
    <String, Integer> hashMap)
{
  if(start != 0)
  {
    while (fileString[(int) start] != '.')
      start -= 1;
    start += 1;
  }

  stop -= (N - 3);
  for (int i = 0; i < N - 1; i++)
  {
    if(stop < fileString.length)
    {
      while (stop < fileString.length &&
          fileString[(int) stop] != '.')
        stop += 1;
      stop += 1;
    }
  }
  stop -= 1;

  if (stop > fileString.length)
  stop = fileString.length;
  ...
}
```

The threads are invoked with *start* and *stop* indexes as shown in section 2.3.2, which is ok for *character n-gram* computation since the processed elements have all length 1, hence this index *”adjustment”* is needed for *word n-gram* computation since different words may have different lengths. The only thing we know about the elements in *word n-gram* computation is that they’re separated by *’.’*

That being said, we can take a look at the tests results for each of the three datasets used and for each parallel version of the application (results written in **bold** represent the best result obtained at vary of the number of threads for each parallel version and those written in **red** represent the best result between the three parallel versions of the application):

texts10

SEQ	Sequential	2 thread		4 thread		8 thread	
merge	time	time	speedup	time	speedup	time	speedup
2-gram	2200.0	1478.1	1.48	**1431.3**	**1.53**	1454.6	1.51
3-gram	2695.1	1932.8	1.39	**1812.4**	**1.48**	1932.8	1.39
5-gram	3200.0	2148.3	1.48	**2100.0**	**1.52**	2140.5	1.49
10-gram	4298.3	2578.1	1.66	**2334.3**	**1.84**	2343.7	1.83

PAR	Sequential	2 thread		4 thread		8 thread	
merge	time	time	speedup	time	speedup	time	speedup
2-gram	2192.1	1479.6	1.48	1448.4	1.51	**1434.3**	**1.52**
3-gram	2751.4	1901.4	1.44	**1798.4**	**1.52**	1807.8	1.52
5-gram	3312.3	2137.5	1.54	1999.9	1.65	**1929.6**	**1.71**
10-gram	4303.0	2623.3	1.64	**2295.3**	**1.87**	2334.3	1.84

NO	Sequential	2 thread		4 thread		8 thread	
future	time	time	speedup	time	speedup	time	speedup
2-gram	2199.9	**1184.3**	**1.85**	1192.2	1.84	1256.2	1.75
3-gram	2736.0	1649.9	1.65	**1565.6**	**1.74**	1642.2	1.66
5-gram	3243.6	1840.6	1.76	**1770.3**	**1.83**	1879.5	1.72
10-gram	4304.5	2248.5	1.91	**1935.9**	**2.22**	2063.9	2.08

texts15

SEQ	Sequential	2 thread		4 thread		8 thread	
merge	time	time	speedup	time	speedup	time	speedup
2-gram	3018.6	2582.7	1.16	**2098.4**	**1.43**	2175.0	1.38
3-gram	3876.8	2621.8	1.47	**2554.6**	**1.51**	2693.6	1.43
5-gram	5254.5	3384.2	1.55	**3182.7**	**1.65**	3296.8	1.59
10-gram	6977.9	3948.3	1.76	**3498.4**	**1.99**	3917.1	1.78

PAR	Sequential	2 thread		4 thread		8 thread	
merge	time	time	speedup	time	speedup	time	speedup
2-gram	3010.8	2045.2	1.47	**2001.5**	**1.50**	2029.5	1.48
3-gram	3850.7	3023.4	1.27	2573.3	1.49	**2562.5**	**1.50**
5-gram	5165.4	3307.6	1.56	**3107.7**	**1.66**	3098.4	1.66
10-gram	6846.7	4446.7	1.53	**3854.6**	**1.77**	3692.1	1.85

NO	Sequential	2 thread		4 thread		8 thread	
future	time	time	speedup	time	speedup	time	speedup
2-gram	2909.2	1676.5	1.73	**1634.3**	**1.78**	1821.8	1.59
3-gram	3792.1	**2178.2**	**1.74**	2225.0	1.70	2460.8	1.54
5-gram	4999.8	**2978.1**	**1.67**	3184.2	1.57	3265.6	1.53
10-gram	6763.8	3782.8	1.78	**3662.3**	**1.84**	3814.0	1.77

texts42

SEQ	Sequential	2 thread		4 thread		8 thread	
merge	time	time	speedup	time	speedup	time	speedup
2-gram	6376.4	**4517.0**	**1.41**	4528.1	1.40	4706.4	1.35
3-gram	9906.1	7402.9	1.33	7034.2	1.40	**6835.7**	**1.44**
5-gram	15165.2	9529.4	1.59	**8875.0**	**1.70**	8918.7	1.70
10-gram	20110.2	10821.6	1.85	9796.6	2.05	**9632.8**	**2.08**

PAR	Sequential	2 thread		4 thread		8 thread	
merge	time	time	speedup	time	speedup	time	speedup
2-gram	6562.3	4567.1	1.43	**4493.7**	**1.46**	4672.1	1.40
3-gram	9790.3	**6432.7**	**1.52**	6699.8	1.46	6671.7	1.46
5-gram	15190.1	8963.8	1.69	**8815.8**	**1.72**	9115.3	1.66
10-gram	19960.2	10867.0	1.83	**9443.5**	**2.11**	10181.1	1.96

NO	Sequential	2 thread		4 thread		8 thread	
future	time	time	speedup	time	speedup	time	speedup
2-gram	6615.8	**3838.9**	**1.72**	4701.5	1.40	3942.0	1.67
3-gram	9685.6	**6167.0**	**1.57**	6345.0	1.52	6603.0	1.46
5-gram	15363.6	**9293.8**	**1.65**	9762.1	1.57	10376.2	1.48
10-gram	18925.9	13071.8	1.44	**12844.9**	**1.47**	13659.0	1.38

It comes out that the best results are obtained predominantly using *4 threads*, computing with the *no future* parallel version of the application.

4.2. Character n-grams

The parallel version of *compute_chars()* is similar to the sequential version. Here follows the test results:

texts10

SEQ	Sequential	2 thread		4 thread		8 thread	
merge	time	time	speedup	time	speedup	time	speedup
2-gram	2221.9	1142.4	1.94	**1111.6**	**1.99**	1228.4	1.80
3-gram	2426.6	1236.7	1.96	**1189.1**	**2.04**	1357.0	1.78
5-gram	3911.4	2456.6	1.59	**2074.4**	**1.88**	2156.6	1.81
10-gram	10360.7	7089.2	1.46	**6991.5**	**1.48**	7575.8	1.36

PAR	Sequential	2 thread		4 thread		8 thread	
merge	time	time	speedup	time	speedup	time	speedup
2-gram	2304.7	1149.8	2.00	**1138.2**	**2.02**	1234.3	1.86
3-gram	2493.7	1273.5	1.95	**1250.7**	**1.99**	1395.3	1.78
5-gram	3879.5	2460.9	1.57	**2164.1**	**1.79**	2345.2	1.65
10-gram	11347.3	8399.1	1.35	**7865.1**	**1.44**	8859.2	1.28

NO	Sequential	2 thread		4 thread		8 thread	
future	time	time	speedup	time	speedup	time	speedup
2-gram	2190.6	1168.7	1.87	**1132.8**	**1.93**	1203.1	1.82
3-gram	2376.5	1203.0	1.97	**1146.9**	**2.07**	1235.8	1.92
5-gram	3857.8	1904.6	2.02	**1504.6**	**2.56**	1517.2	2.54
10-gram	10409.2	6901.4	1.50	7002.9	1.48	**6771.6**	**1.53**

texts15

SEQ	Sequential	2 thread		4 thread		8 thread	
merge	time	time	speedup	time	speedup	time	speedup
2-gram	3023.7	**1453.2**	**2.08**	1533.3	1.97	1719.6	1.75
3-gram	3452.5	1632.7	2.11	**1554.7**	**2.22**	1830.1	1.88
5-gram	5450.3	3297.7	1.65	**2878.3**	**1.89**	3427.4	1.59
10-gram	18651.6	**12088.2**	**1.54**	12181.8	1.53	13401.7	1.39

PAR	Sequential	2 thread		4 thread		8 thread	
merge	time	time	speedup	time	speedup	time	speedup
2-gram	3187.4	**1569.1**	**2.03**	1635.9	1.94	1803.0	1.76
3-gram	3451.0	**1687.5**	**2.04**	1704.6	2.02	1954.6	1.76
5-gram	5410.7	**3384.3**	**1.59**	3535.8	1.53	3892.6	1.38
10-gram	19763.7	**12797.2**	**1.54**	12797.3	1.54	14465.2	1.36

NO	Sequential	2 thread		4 thread		8 thread	
future	time	time	speedup	time	speedup	time	speedup
2-gram	3049.9	**1528.1**	**1.99**	1548.4	1.96	1685.8	1.80
3-gram	3545.1	1604.8	2.20	**1568.7**	**2.25**	1731.1	2.04
5-gram	5545.1	2742.1	2.02	**2021.8**	**2.74**	2110.9	2.62
10-gram	17951.2	11909.0	1.50	**11543.4**	**1.55**	12287.1	1.46

texts42

SEQ	Sequential	2 thread		4 thread		8 thread	
merge	time	time	speedup	time	speedup	time	speedup
2-gram	6431.0	**3073.0**	**2.09**	3330.6	1.93	3822.8	1.68
3-gram	7320.3	**3281.0**	**2.23**	3565.5	2.05	5646.8	1.29
5-gram	11817.1	8158.2	1.44	**8103.5**	**1.45**	9337.9	1.26
10-gram							

PAR	Sequential	2 thread		4 thread		8 thread	
merge	time	time	speedup	time	speedup	time	speedup
2-gram	6229.5	**3165.3**	**1.96**	3419.2	1.82	3849.4	1.61
3-gram	7244.6	**3421.8**	**2.11**	3783.6	1.91	5408.5	1.33
5-gram	11622.4	8456.1	1.37	**8291.9**	**1.40**	9557.9	1.21
10-gram							

NO	Sequential	2 thread		4 thread		8 thread	
future	time	time	speedup	time	speedup	time	speedup
2-gram	6348.3	**3151.4**	**2.01**	3273.3	1.93	3595.2	1.76
3-gram	7432.9	**3223.3**	**2.30**	3376.4	2.20	3646.6	2.03
5-gram	11743.4	5921.6	1.98	**4381.2**	**2.68**	4704.5	2.49
10-gram	58768.8	43520.9	1.35	**43466.3**	**1.35**	45915.8	1.27

We see that on *character n-gram* computation the *no future* version of the parallel application remains the optimal one. The majority of the best results are obtained using 2 or 4 threads.