UNIVERSITÀ
DEGLI STUDI
FIRENZE

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

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

19.01.2021

UNIVERSITÀ DEGLI STUDI FIRENZE

Introduction

- in general
  - an N-gram is a contiguous sequence of N items from a text or speech.
  - items can be phonemes, syllables, letters, words or base pairs according to the application
- in this application
  - computing N-gram given a set of *.txt* files from *gutemberg.org*
  - items are *word* and *characters*

UNIVERSITÀ DEGLI STUDI FIRENZE

Implementation

UNIVERSITÀ DEGLI STUDI FIRENZE

Pre-processing

```
public static void loadDatasets(LinkedList<String> txtListMain, String directory)
{
  try (DirectoryStream<Path> stream = Files.newDirectoryStream(Paths.get(directory)))
  {
    for (Path path : stream)
    {
      if (!Files.isDirectory(path))
      {
        txtListMain.add("./" + directory + "/" + path.getFileName().toString());
      }
    }
  } catch (IOException e)
    {
      e.printStackTrace();
    }
    // System.out.println("Loaded " +  txtListMain.size() + " files \n");
}
```

```
public static char[] process_txt(String txt, String mode) {
        Path path = Paths.get(txt);
        try {
            Stream<String> lines = Files.lines(path);
            char[] filestring = null;
            if(mode.equals("word")) {
                filestring = (lines.collect(Collectors.joining(" ")))
                        .replaceAll("[SIMBOLS_TO_REPLACE]+", ".").toCharArray();
            }else {
                filestring = (lines.collect(Collectors.joining(" ")))
                        .replaceAll("[SIMBOLS_TO_REPLACE]+", "").toCharArray();
            }

            for(int i = 0; i < filestring.length - 1; ++i) {
                if (Character.isUpperCase(filestring[i])) {
                    filestring[i] = Character.toLowerCase(filestring[i]);
                }
            }
            return filestring;
        }
        catch (IOException e) {
            System.out.println(e);
            System.exit(1);
            return null;
        }
    }
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Sequential implementation

- once available the *txtList* containing all files paths, for each file:
  - call *process_txt()* to generate the *char[]* accordingly
  - compute either *word* n-grams or *char* n-grams
- the data structure used in this version is a
  - *HashMap* <*String, Integer*> which is suitable for the application since it provides constant time operations for input/output
    - *String* stores the calculated n-gram
    - *Integer* stores the number of occurrences of the relative n-gram in all input files

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

        int i = 0;
        String key;
        while (i < fileString.length) {
            StringBuilder builder = new StringBuilder();
            int j = 0;
            int k = 0;
            int count = 0;
            key = null;
            while(j < N){

                char tmp;
                if(i < fileString.length)
                    tmp = fileString[i];
                else tmp = '.';

                if (tmp == '.'){
                    if(i < fileString.length)
                        k = i - count;
                    else {
                        if(j != N - 1)
                            j += N;
                        k = i;
                    }
                    i = i+1;
                    j = j+1;
                    builder.append("␣");
                    ...
```

```
                                        ...
                    if(N > 2) {
                        if (j == 1)
                            count = N - 2;
                     }else
                        count = 0;
                }else{
                    i = i+1;
                    builder.append(tmp);
                    count ++;
                }
                if(j == N)
                    key = builder.toString();
            }
            if(key != null) {
                if (!hashMap.containsKey(key)) {
                    hashMap.put(builder.toString(), 1);
                } else if (hashMap.containsKey(key)) {
                    hashMap.put(builder.toString(), hashMap.get(key) + 1);
                }
            }
            if(N != 1)
                i = k;
        }
        return hashMap;
    }
```

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

        for(int i = 0; i < fileString.length - N + 1; ++i) {
            StringBuilder builder = new StringBuilder();

            for(int j = 0; j < N; ++j) {
                builder.append(fileString[i + j]);
            }

            String key = builder.toString();

            if (!hashMap.containsKey(key)) {
                hashMap.put(builder.toString(), 1);
            }
            else if (hashMap.containsKey(key)) {
                hashMap.put(builder.toString(), hashMap.get(key) + 1);
            }
        }

        return hashMap;
    }
```

```
public static HashMap<String, Integer> iterate_txt(LinkedList<String> txtList,
      HashMap<String, Integer> dict, String MODE, int N) {

        //System.out.println("Computing " + N + "-grams of " + MODE);

        while(!txtList.isEmpty()) {

                String txtName = txtList.poll();
                char[] file = pre_process.process_txt(txtName, MODE);
                if(MODE.equals("word"))
                    compute_words(file, dict, N);
                else
                    compute_chars(file, dict, N);

        }
        return dict;
    }
```

Parallel implementation

- data of each file divided in as many parts as the thread number.
- parallel version implemented using 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.

UNIVERSITÀ DEGLI STUDI FIRENZE

Data structure

- The data structures used in the parallel implementation are
  - *HashMap<String, Integer>*, used as a private dictionary for each thread.
  - *ConcurrentHashMap<String, Integer>*, representing the final dictionary which will contain all the results from all threads after a merge of each future.

UNIVERSITÀ DEGLI STUDI FIRENZE

Computation

```
public static ConcurrentHashMap<String, Integer> iterate_txt(LinkedList<String>
      txtList, ConcurrentHashMap<String, Integer> dict, String MODE, int N, int
      NUM_THREADS) {

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

        ExecutorService executor = Executors.newFixedThreadPool(NUM_THREADS);

        while (!txtList.isEmpty()) {

            String txtName = txtList.poll();
            char[] file = pre_process.process_txt(txtName, MODE);

            int fileLen = file.length;
            double k = Math.floor(fileLen / NUM_THREADS);
            double stop;
            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);
            }
        }
        ...
```

```
        ...
        try {

            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);
            }
            awaitTerminationAfterShutdown(executor);

        } catch (Exception e) {
            System.out.println(e);
        }
        return dict;
    }
```

Par_thread

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

    public int N;
    public String MODE;

    private double start, stop;
    private String id;
    private HashMap<String, Integer> thread_dict;
    private char[] fileString;

    StringBuilder builder;

    public Par_thread(String id, double start, double stop, char[] fileString, int N,
           String MODE){...}

    public HashMap<String, Integer> call() {

        if(MODE.equals("word"))
            compute_words(fileString, thread_dict);
        else
            compute_chars(fileString, thread_dict);

        return thread_dict;
    }
```

UNIVERSITÀ DEGLI STUDI FIRENZE

Par_thread index adjustment

```
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;
```

UNIVERSITÀ DEGLI STUDI FIRENZE

merge_thread

```
public class merge_thread implements Runnable {

    private ConcurrentHashMap<String, Integer> final_dict;
    private HashMap<String, Integer> dict;

    public merge_thread(HashMap<String, Integer> dict, ConcurrentHashMap<String,
          Integer> final_dict){
        this.dict = dict;
        this.final_dict = final_dict;
    }

    public void run(){

        for (HashMap.Entry<String, Integer> entry : dict.entrySet()) {

            int newValue = entry.getValue();
            String key = entry.getKey();

            if(final_dict.putIfAbsent(key, newValue) != null) {
                final_dict.computeIfPresent(key, (k, val) -> val + newValue);
            }
        }
    }
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

No Futures Version

```
public static ConcurrentHashMap<String, Integer> iterate_txt(LinkedList<String>
      txtList, ConcurrentHashMap<String, Integer> dict, String MODE, int num_bigrams,
       int NUM_THREADS) {

        ExecutorService executor = Executors.newFixedThreadPool(NUM_THREADS);

        while (!txtList.isEmpty()) {

            String txtName = txtList.poll();
            char[] file = pre_process.process_txt(txtName, MODE);

            int fileLen = file.length;
            double k = Math.floor(fileLen / NUM_THREADS);
            double stop;
            for (int i = 0; i < NUM_THREADS; i++) {
                if(i == NUM_THREADS - 1)
                    stop = fileLen -1;
                else
                    stop = ((i + 1) * k) + (num_bigrams - 1) - 1;

                executor.execute(new Par_thread_no_future("t" + i, i * k, stop, file,
                      dict, num_bigrams, MODE));
            }
        }
        awaitTerminationAfterShutdown(executor);
        return dict;
    }
```

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*
- 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 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*:

UNIVERSITÀ DEGLI STUDI FIRENZE

Results

- the results are an average on 10 iterations.
- results represent the comparison of *computation time* and *speedup* between the sequential version and each of the parallel versions:
  - futures and sequential merge
  - futures and parallel merge
  - no futures

UNIVERSITÀ DEGLI STUDI FIRENZE

Test structure

```
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
        }
      }
    }
  }
}
```

Results

Word n-grams

texts10

SEQ merge	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 merge	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 future	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 merge	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 merge	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 future	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 merge	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 merge	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 future	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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

Results

Character n-grams

texts10

SEQ merge	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 merge	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 future	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 merge	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 merge	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 future	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 merge	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 merge	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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 future	Sequential time	2 thread time	2 thread speedup	4 thread time	4 thread speedup	8 thread time	8 thread 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