Distributed Systems Assignment – Map Reduce

Stephen Murphy – 17410394

1. Text files used are scripts of Alice in Wonderland, As You Like It, and Paradise Lost, copied multiple times over, containing 20000, 40000 and 20000 lines of text, respectively.

2. Remove Punctuation and numbering by using regex “\p{punct}” and “\d+” to replace all punctuation and numbers with nothing.

```
for(String word: words) {
    //Remove Punctuation from words
    String filteredWord = word.replaceAll("\\p{Punct}", "");
    //Remove numbers from words
    filteredWord = filteredWord.replaceAll("\\d+", "");
    results.add(new MappedItem(filteredWord, file));
}
```

3. Measure time taken for each approach to complete by using nanoTime at the start and end of each approach and calculating the difference, before converting to milliseconds. Example shown below.

```
            // APPROACH #3: Distributed MapReduce
            {
                long startTime3 = System.nanoTime();
    }
}
//System.out.println(output);
long elapsedTime3 = System.nanoTime() - startTime3;
System.out.println("Approach 3 Execution Time: " + elapsedTime3/1000000 + " ms");
```

4. Edited program to take extra argument linesPerThread that sets the number of lines of text that are allocated to each thread in the mapping phase. Code now creates an array of strings groupedLines that divides the data based on the number of total lines divided by the linesPerThread value. One new thread is then created for each string in groupedLines.

```
Iterator<Map.Entry<String, String>> inputIter = input.entrySet().iterator();
while(inputIter.hasNext()) {
    Map.Entry<String, String> entry = inputIter.next();
    final String file = entry.getKey();
    final String contents = entry.getValue();

    double lines = 1;
    Matcher m = Pattern.compile("\n").matcher(contents);
    while (m.find()) {
        lines ++;
    }

    String[] groupedLines = contents.split("\n", (int)Math.ceil(lines/linesPerThread));
    for(String s : groupedLines) {
        //System.out.println("Starting thread (File: " + file + ")");
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                map(file, s, mapCallback);
            }
        });
        mapCluster.add(t);
        t.start();
    }
}
```

5. Adds extra argument wordsPerThread which sets the number of words that are allocated to each thread in the reducing phase. List threadEntries is created to hold multiple entry objects to pass to a thread. Reduce method is edited to handle multiple entries at a time. Callback is also edited to handle a list of words and reduced lists being sent to reduceDone method.

```
final ReduceCallback<String, String, Integer> reduceCallback = new ReduceCallback<String, String, Integer>() {
    @Override
    public synchronized void reduceDone(List<String> words, List<Map<String, Integer>> reducedListGroup) {
        for(int i=0;i<words.size();i++) {
            String k = words.get(i);
            Map<String, Integer> v = reducedListGroup.get(i);
            output.put(k, v);
        }
    }
};

List<Thread> reduceCluster = new ArrayList<Thread>(groupedItems.size());

Iterator<Map.Entry<String, List<String>>> groupedIter = groupedItems.entrySet().iterator();

//Creates entries list and iterates over each entry to add to list
List<Map.Entry<String, List<String>>> entries = new ArrayList<Map.Entry<String, List<String>>>();
while(groupedIter.hasNext()) {
    Map.Entry<String, List<String>> entry = groupedIter.next();
    entries.add(entry);
}
```

```java
//Calculates number of threads needed based on how many words per thread user requests and the amount of entries
double threads = (int)Math.ceil(entries.size() / wordsPerThread);

//Divides up entries to different threads and starts the threads
for(int i =0;i<threads;i++) {
    List<Map.Entry<String, List<String>>> threadEntries = new ArrayList<Map.Entry<String, List<String>>>();
    for(int j=(int)(i*wordsPerThread);j<(i+1)*wordsPerThread;j++) {
        threadEntries.add(entries.get(j));
        //for last thread, entries left may be less than wordsPerThread, stops when max has been reached
        if(j == entries.size() - 1) {
            break;
        }
    }

    //Starts thread and sends group of entries to be reduced
    Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
            reduce(threadEntries, reduceCallback);
        }
    });
    reduceCluster.add(t);
    t.start();
}
```

Output Example

```
C:\Users\steph\OneDrive - National University of Ireland, Galway\Year_4\Distributed Systems and co-op Computing\MapReduc
e>java MapReduceFiles file1.txt file2.txt file3.txt 500 400
Approach 1 Execution Time: 134 ms
Approach 2 Execution Time: 65 ms
Approach 3 Execution Time: 161 ms
```

Testing of different linesPerThread and wordsPerThread values

Second java class used that iterates over different values for each argument and runs the Map Reduce algorithm for each value. The results are output to file times.txt. Results are shown below.

Lines per Thread (map)	Words per Thread (reduce)	Method 1 Time (ms)	Method 2 Time (ms)	Method 3 Time (ms)
1	1	280	199	9997
1	101	152	215	8419
1	201	117	149	8089
1	301	141	173	8137
1	401	127	168	10009
1	501	108	158	9431
1	601	212	202	9093
1	701	112	146	8559
1	801	113	155	7908
1	901	110	153	7838
1001	1	111	149	1577
1001	101	123	177	428
1001	201	114	146	467
1001	301	112	160	270
1001	401	285	157	392

1001	501	109	163	253
1001	601	111	151	246
1001	701	114	151	535
1001	801	112	147	247
1001	901	111	154	339
2001	1	121	153	1563
2001	101	122	181	457
2001	201	117	228	492
2001	301	116	157	310
2001	401	110	149	250
2001	501	110	152	254
2001	601	114	152	237
2001	701	114	158	252
2001	801	110	149	258
2001	901	115	157	264
3001	1	119	161	1407
3001	101	123	320	566
3001	201	111	161	266
3001	301	113	161	275
3001	401	115	154	513
3001	501	114	153	241
3001	601	118	161	254
3001	701	113	147	256
3001	801	120	153	255
3001	901	113	153	249
4001	1	111	152	1494
4001	101	122	156	566
4001	201	119	155	270
4001	301	111	153	247
4001	401	111	162	263
4001	501	111	152	540
4001	601	118	148	267
4001	701	114	157	668
4001	801	110	150	333
4001	901	117	150	328
5001	1	119	150	1480
5001	101	121	159	330
5001	201	113	140	257
5001	301	114	155	251
5001	401	114	222	258
5001	501	112	150	238
5001	601	110	147	243
5001	701	112	159	251
5001	801	114	172	240

5001	901	112	155	239
6001	1	115	150	1479
6001	101	118	157	307
6001	201	123	159	238
6001	301	109	163	257
6001	401	110	148	232
6001	501	116	158	317
6001	601	112	158	585
6001	701	113	274	480
6001	801	118	153	279
6001	901	117	151	254
7001	1	115	154	1572
7001	101	118	164	469
7001	201	111	148	237
7001	301	114	149	261
7001	401	108	151	392
7001	501	113	158	249
7001	601	123	153	242
7001	701	112	147	253
7001	801	110	225	242
7001	901	112	161	268
8001	1	112	153	1410
8001	101	118	153	364
8001	201	113	149	458
8001	301	200	152	258
8001	401	111	151	242
8001	501	113	150	248
8001	601	113	153	255
8001	701	109	148	236
8001	801	111	148	267
8001	901	110	150	246
9001	1	109	156	1482
9001	101	126	156	290
9001	201	130	152	256
9001	301	115	153	251
9001	401	112	154	488
9001	501	111	149	291
9001	601	113	151	261
9001	701	110	157	250
9001	801	113	155	273
9001	901	110	152	241

Conclusions

The addition of extra words and lines per thread has an improvement on the map reduce algorithm as seen from the results above. However, the brute force and non-multithreaded still perform the task quicker than the multithreaded algorithm. It is possible that with further improvements to the algorithm or with larger input files that approach 3 could be faster than approach 1 and 2.