

Assignment 2: MapReduce

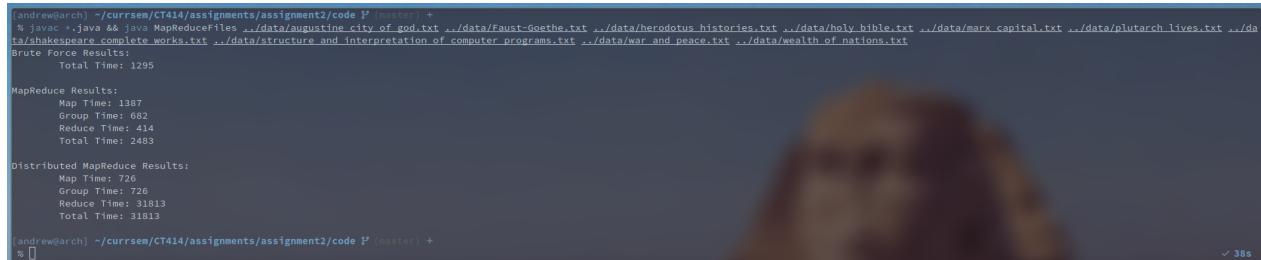
1 Set-Up

To obtain large text files to test the program with, I downloaded the first 10 long books I could think of from archive.org in txt file form. These were:

1. The Bible;
2. *War & Peace* by Leo Tolstoy;
3. Plutarch's *Lives*;
4. Herodotus' *Histories*;
5. *City of God* by Augustine of Hippo;
6. *Faust* by Goethe;
7. *Wealth of Nations* by Adam Smith;
8. *Capital* by Karl Marx;
9. The complete works of William Shakespeare;
10. *Structure & Interpretation of Computer Programs* by Harold Abelson & Gerald Jay Sussman.

2 Baseline Results

I modified the code to measure & output the time taken by each approach, in milliseconds. I also added timing for the different phases of the two MapReduce implementations, timing the map time, group time, and reduce time separately.



```
[andrewarch] ~/currsen/CT414/assignments/assignment2/code $ ^ + % java -jar MapReduceFiles ..../data/augustine_city_of_god.txt ..../data/Faust-Goethe.txt ..../data/herodotus_histories.txt ..../data/holy_bible.txt ..../data/marx_capital.txt ..../data/plutarch_lives.txt ..../data/shakespeare_complete_works.txt ..../data/structure_and_interpretation_of_computer_programs.txt ..../data/war_and_peace.txt ..../data/wealth_of_nations.txt
Brute Force Results:
Total Time: 1295

MapReduce Results:
Map Time: 1387
Group Time: 682
Reduce Time: 414
Total Time: 2483

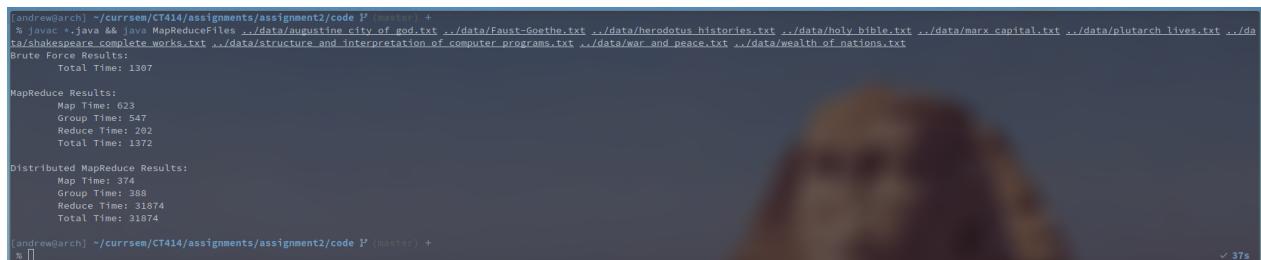
Distributed MapReduce Results:
Map Time: 726
Group Time: 726
Reduce Time: 31813
Total Time: 31813

[andrewarch] ~/currsen/CT414/assignments/assignment2/code $ ^ (MASTER) + % [ ] ✓ 38s
```

Figure 1: Baseline results for my list of files (in milliseconds)

As can be seen from the above terminal screenshot, the brute force approach performed best with no modifications, followed by the non-distributed MapReduce, followed by the distributed MapReduce; this is to be expected, as the brute force approach is the simplest & requires the fewest iterations over the data and no complex data structures. The non-distributed MapReduce requires more intermediate data structure and more iterations over the data. Finally, the non-optimised version of the distributed MapReduce is the slowest because it spawns a thread for each word in the dataset, causing massive stress on the CPU and memory.

I also updated the code to use ArrayLists rather than LinkedLists to reduce memory overhead and have faster traversal.



```
[andrewarch] ~/currsen/CT414/assignments/assignment2/code $ ^ (MASTER) + % java -jar MapReduceFiles ..../data/augustine_city_of_god.txt ..../data/Faust-Goethe.txt ..../data/herodotus_histories.txt ..../data/holy_bible.txt ..../data/marx_capital.txt ..../data/plutarch_lives.txt ..../data/shakespeare_complete_works.txt ..../data/structure_and_interpretation_of_computer_programs.txt ..../data/war_and_peace.txt ..../data/wealth_of_nations.txt
Brute Force Results:
Total Time: 1387

MapReduce Results:
Map Time: 623
Group Time: 547
Reduce Time: 202
Total Time: 1372

Distributed MapReduce Results:
Map Time: 374
Group Time: 388
Reduce Time: 31874
Total Time: 31874

[andrewarch] ~/currsen/CT414/assignments/assignment2/code $ ^ (MASTER) + % [ ] ✓ 37s
```

Figure 2: Baseline results with ArrayList update (in milliseconds)

As can be seen from the above terminal screenshot, this has no affect on the brute force results (besides slight variance due to background processes running on my laptop) as this approach did not use `LinkedLists` anyway. The non-distributed MapReduce approach was significantly faster due to the faster iteration and lower memory overhead. The distributed MapReduce saw significant improvements in the map & group phases, but these were dwarfed by the still greatly inefficient reduce phase.

3 Testing the Updated Code

After implementing the requested changes in steps 2–6 of the assignment specification, I then implemented a grid-search function which tested a range of values for the number of lines of text per map thread and the number of words per reduce thread. The results of this grid-search were exported to a CSV file for analysis. I then wrote a Python script to visualise the parameter combinations using heatmaps.



```
[laptop:~/course/CT414/assignments/assignment2/code]$ % java && java MapReduceFiles ../../data/augustine_city_of_god.txt ../../data/Faust-Goethe.txt ../../data/herodotus_histories.txt ../../data/holy_bible.txt ../../data/marx_capital.txt ../../data/plutarch_lives.txt ../../data/shakespeare_complete_works.txt ../../data/structure_and_interpretation_of_computer_programs.txt ../../data/war_and_peace.txt ../../data/wealth_of_nations.txt && python3 plots.py
===== Starting Grid Search =====
MapLines: 1000, ReduceWords: 100
    Map Time: 2099 ms
    Group Time: 406 ms
    Reduce Time: 335 ms
    Total Time: 2840 ms
-----
MapLines: 1000, ReduceWords: 200
    Map Time: 1610 ms
    Group Time: 454 ms
    Reduce Time: 198 ms
    Total Time: 2262 ms
-----
MapLines: 1000, ReduceWords: 500
    Map Time: 1388 ms
    Group Time: 452 ms
    Reduce Time: 46 ms
    Total Time: 1886 ms
-----
MapLines: 1000, ReduceWords: 1000
    Map Time: 1538 ms
    Group Time: 302 ms
    Reduce Time: 48 ms
    Total Time: 1888 ms
-----
MapLines: 2000, ReduceWords: 100
    Map Time: 1726 ms
    Group Time: 314 ms
    Reduce Time: 263 ms
    Total Time: 2303 ms
-----
MapLines: 2000, ReduceWords: 200
    Map Time: 1512 ms
    Group Time: 323 ms
    Reduce Time: 62 ms
    Total Time: 1897 ms
-----
MapLines: 2000, ReduceWords: 500
    Map Time: 1669 ms
    Group Time: 334 ms
    Reduce Time: 46 ms
    Total Time: 2049 ms
-----
MapLines: 2000, ReduceWords: 1000
    Map Time: 1762 ms
    Group Time: 313 ms
    Reduce Time: 113 ms
    Total Time: 2154 ms
-----
MapLines: 5000, ReduceWords: 100
    Map Time: 1291 ms
    Group Time: 331 ms
    Reduce Time: 92 ms
    Total Time: 1714 ms
-----
MapLines: 5000, ReduceWords: 200
    Map Time: 1877 ms
    Group Time: 368 ms
    Reduce Time: 67 ms
    Total Time: 2312 ms
-----
MapLines: 5000, ReduceWords: 500
    Map Time: 1640 ms
    Group Time: 396 ms
    Reduce Time: 41 ms
    Total Time: 2077 ms
-----
MapLines: 5000, ReduceWords: 1000
    Map Time: 1439 ms
    Group Time: 365 ms
    Reduce Time: 193 ms
    Total Time: 1997 ms
-----
MapLines: 10000, ReduceWords: 100
    Map Time: 1285 ms
    Group Time: 359 ms
    Reduce Time: 94 ms
    Total Time: 1738 ms
-----
MapLines: 10000, ReduceWords: 200
    Map Time: 1598 ms
    Group Time: 359 ms
    Reduce Time: 98 ms
    Total Time: 2055 ms
-----
MapLines: 10000, ReduceWords: 500
    Map Time: 1489 ms
    Group Time: 314 ms
    Reduce Time: 68 ms
    Total Time: 1871 ms
-----
MapLines: 10000, ReduceWords: 1000
    Map Time: 1460 ms
    Group Time: 332 ms
    Reduce Time: 47 ms
    Total Time: 1839 ms
```

Figure 3: Running the grid-search and plotting the results

Map Lines	Reduce Words	Map Time (ms)	Group Time (ms)	Reduce Time (ms)	Total Time (ms)
1000	100	2099	406	335	2840
1000	200	1610	454	198	2262
1000	500	1388	452	46	1886
1000	1000	1538	302	48	1888
2000	100	1726	314	263	2303
2000	200	1512	323	62	1897
2000	500	1669	334	46	2049
2000	1000	1762	279	113	2154
5000	100	1291	331	92	1714
5000	200	1877	368	67	2312
5000	500	1640	396	41	2077
5000	1000	1439	365	193	1997
10000	100	1285	359	94	1738
10000	200	1598	359	98	2055
10000	500	1489	314	68	1871
10000	1000	1460	332	47	1839

Table 1: Results written to `performance_results.csv`

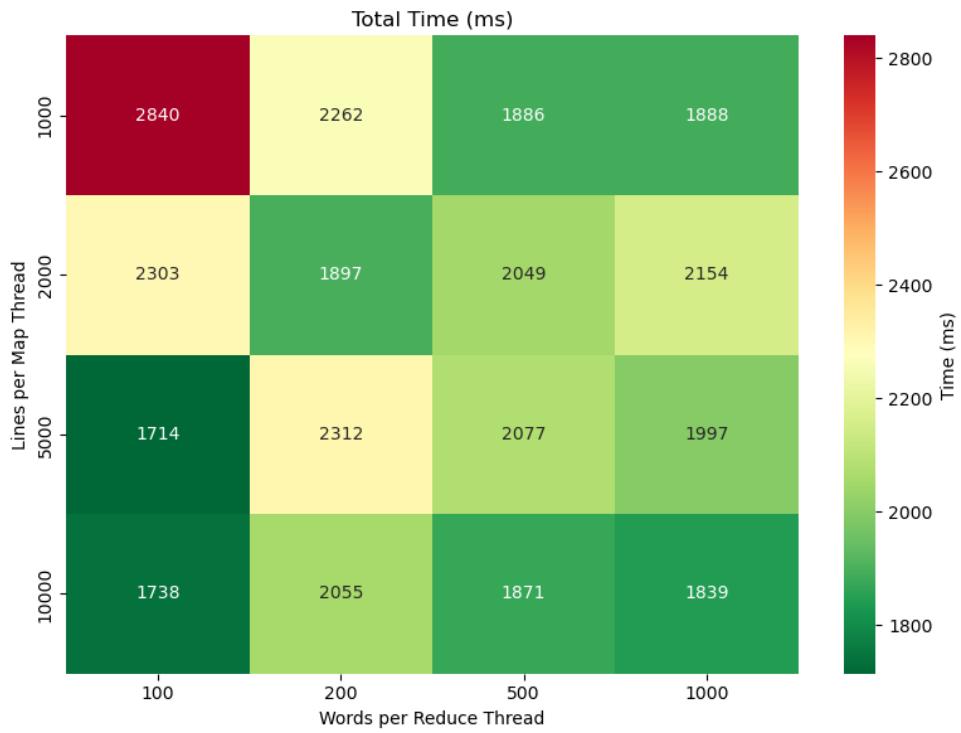


Figure 4: Heatmap of total time taken by each parameter combination

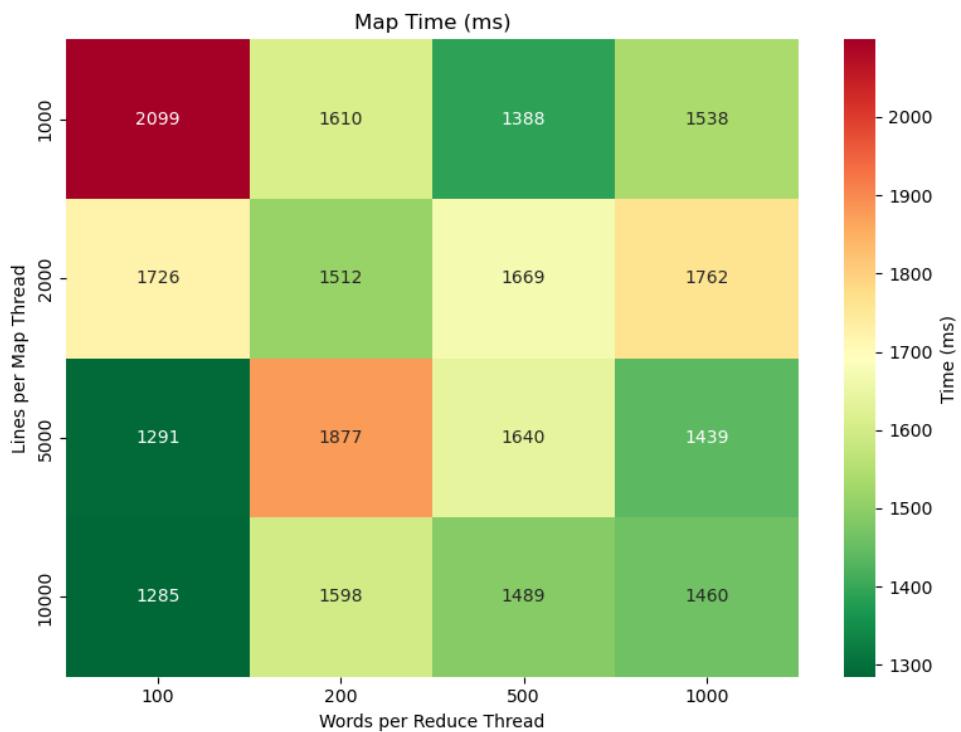


Figure 5: Heatmap of time taken during the map phase by each parameter combination

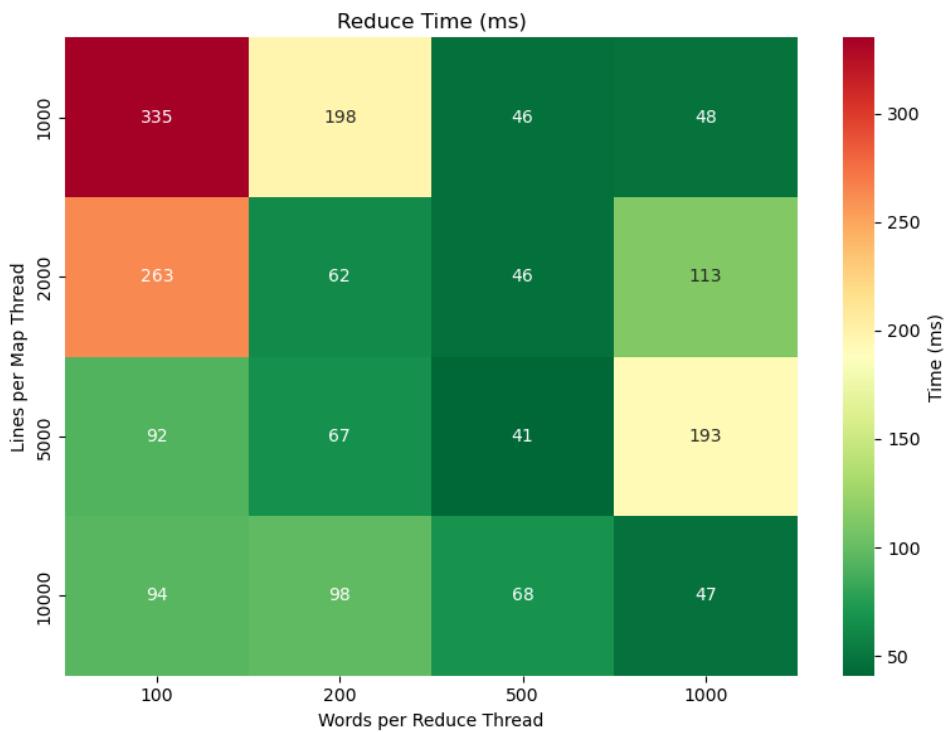


Figure 6: Heatmap of time taken during the reduce phase by each parameter combination

4 Appendix: Source Code

```

1 import java.util.*;
2 import java.io.*;
3
4 public class MapReduceFiles {
5
6     private static final String CSV_FILE = "performance_results.csv";
7
8     public static void main(String[] args) {
9         if (args.length < 1) {
10             System.err.println("Usage: java MapReduceFiles file1.txt file2.txt ... fileN.txt");
11             return;
12         }
13
14         Map<String, String> input = new HashMap<>();
15         try {
16             for (String filename : args) {
17                 input.put(filename, readFile(filename));
18             }
19         } catch (IOException ex) {
20             System.err.println("Error reading files: " + ex.getMessage());
21             ex.printStackTrace();
22             return;
23         }
24
25         int[] mapSizes = {1000, 2000, 5000, 10000};
26         int[] reduceSizes = {100, 200, 500, 1000};
27
28         System.out.println("===== Starting Grid Search =====");
29
30         try (PrintWriter writer = new PrintWriter(new FileWriter(CSV_FILE))) {
31             writer.println("MapLines,ReduceWords,MapTime,GroupTime,ReduceTime,TotalTime");
32         }

```

```

33     for (int mapSize : mapSizes) {
34         for (int reduceSize : reduceSizes) {
35             runDistributedMapReduce(input, mapSize, reduceSize, writer);
36         }
37     }
38
39 } catch (IOException e) {
40     System.err.println("Error writing to CSV file: " + e.getMessage());
41 }
42
43 System.out.println("===== Grid Search Complete =====");
44 System.out.println("Results saved to: " + CSV_FILE);
45 }
46
47 public static void runDistributedMapReduce(Map<String, String> input, int linesPerMapThread, int
48 ← wordsPerReduceThread, PrintWriter csvWriter) {
49     final Map<String, Map<String, Integer>> output = new HashMap<>();
50
51     // MAP Phase
52     long mapStartTime = System.currentTimeMillis();
53     List<MappedItem> mappedItems = Collections.synchronizedList(new ArrayList<>());
54
55     final MapCallback<String, MappedItem> mapCallback = new MapCallback<>() {
56         public synchronized void mapDone(String file, List<MappedItem> results) {
57             mappedItems.addAll(results);
58         }
59     };
60
61     List<Thread> mapCluster = new ArrayList<>();
62     for (Map.Entry<String, String> entry : input.entrySet()) {
63         final String file = entry.getKey();
64         final String[] lines = entry.getValue().split("\\r?\\n");
65
66         for (int i = 0; i < lines.length; i += linesPerMapThread) {
67             int end = Math.min(i + linesPerMapThread, lines.length);
68             final List<String> chunk = new ArrayList<>();
69             for (int j = i; j < end; j++) {
70                 chunk.addAll(splitLongLine(lines[j]));
71             }
72
73             Thread t = new Thread(() -> map(file, chunk, mapCallback));
74             mapCluster.add(t);
75             t.start();
76         }
77     }
78
79     for (Thread t : mapCluster) {
80         try {
81             t.join();
82         } catch (InterruptedException e) {
83             throw new RuntimeException(e);
84         }
85     }
86
87     long mapTotalTime = System.currentTimeMillis() - mapStartTime;
88
89     // GROUP Phase
90     long groupStartTime = System.currentTimeMillis();
91     Map<String, List<String>> groupedItems = new HashMap<>();
92     for (MappedItem item : mappedItems) {
93         groupedItems.computeIfAbsent(item.getWord(), k -> new ArrayList<>()).add(item.getFile());

```

```

93     }
94     long groupTotalTime = System.currentTimeMillis() - groupStartTime;
95
96     // REDUCE Phase
97     long reduceStartTime = System.currentTimeMillis();
98     final ReduceCallback<String, String, Integer> reduceCallback = (word, result) -> {
99         synchronized (output) {
100             output.put(word, result);
101         }
102     };
103
104     List<Thread> reduceCluster = new ArrayList<>();
105     List<Map<String, List<String>>> reduceChunks = new ArrayList<>();
106     Map<String, List<String>> currentChunk = new HashMap<>();
107     int count = 0;
108
109     for (Map.Entry<String, List<String>> entry : groupedItems.entrySet()) {
110         currentChunk.put(entry.getKey(), entry.getValue());
111         count++;
112         if (count >= wordsPerReduceThread) {
113             reduceChunks.add(currentChunk);
114             currentChunk = new HashMap<>();
115             count = 0;
116         }
117     }
118     if (!currentChunk.isEmpty()) reduceChunks.add(currentChunk);
119
120     for (final Map<String, List<String>> chunk : reduceChunks) {
121         Thread t = new Thread(() -> {
122             for (Map.Entry<String, List<String>> entry : chunk.entrySet()) {
123                 reduce(entry.getKey(), entry.getValue(), reduceCallback);
124             }
125         });
126         reduceCluster.add(t);
127         t.start();
128     }
129
130     for (Thread t : reduceCluster) {
131         try {
132             t.join();
133         } catch (InterruptedException e) {
134             throw new RuntimeException(e);
135         }
136     }
137
138     long reduceTotalTime = System.currentTimeMillis() - reduceStartTime;
139     long totalTime = mapTotalTime + groupTotalTime + reduceTotalTime;
140
141     // Print & Log
142     System.out.println("MapLines: " + linesPerMapThread + ", ReduceWords: " + wordsPerReduceThread);
143     System.out.println("\tMap Time: " + mapTotalTime + " ms");
144     System.out.println("\tGroup Time: " + groupTotalTime + " ms");
145     System.out.println("\tReduce Time: " + reduceTotalTime + " ms");
146     System.out.println("\tTotal Time: " + totalTime + " ms");
147     System.out.println("-----");
148
149     csvWriter.printf("%d,%d,%d,%d,%d%n",
150                     linesPerMapThread, wordsPerReduceThread,
151                     mapTotalTime, groupTotalTime, reduceTotalTime, totalTime);
152     csvWriter.flush();
153 }

```

```

154
155 public static void map(String file, List<String> lines, MapCallback<String, MappedItem> callback) {
156     List<MappedItem> results = new ArrayList<>();
157     for (String line : lines) {
158         String[] words = line.trim().split("\\s+");
159         for (String word : words) {
160             word = word.replaceAll("[^a-zA-Z]", "").toLowerCase();
161             if (!word.isEmpty()) {
162                 results.add(new MappedItem(word, file));
163             }
164         }
165     }
166     callback.mapDone(file, results);
167 }
168
169 public static void reduce(String word, List<String> list, ReduceCallback<String, String, Integer>
170 ← callback) {
171     Map<String, Integer> reducedList = new HashMap<>();
172     for (String file : list) {
173         reducedList.put(file, reducedList.getOrDefault(file, 0) + 1);
174     }
175     callback.reduceDone(word, reducedList);
176 }
177
178 public interface MapCallback<E, V> {
179     void mapDone(E key, List<V> values);
180 }
181
182 public interface ReduceCallback<E, K, V> {
183     void reduceDone(E e, Map<K, V> results);
184 }
185
186 private static class MappedItem {
187     private final String word;
188     private final String file;
189
190     public MappedItem(String word, String file) {
191         this.word = word;
192         this.file = file;
193     }
194
195     public String getWord() {
196         return word;
197     }
198
199     public String getFile() {
200         return file;
201     }
202
203     @Override
204     public String toString() {
205         return "[" + word + ", " + file + "]";
206     }
207 }
208
209 private static String readFile(String pathname) throws IOException {
210     File file = new File(pathname);
211     StringBuilder fileContents = new StringBuilder((int) file.length());
212     Scanner scanner = new Scanner(new BufferedReader(new FileReader(file)));
213     String lineSeparator = System.getProperty("line.separator");

```

```

214     try {
215         while (scanner.hasNextLine()) {
216             fileContents.append(scanner.nextLine()).append(lineSeparator);
217         }
218         return fileContents.toString();
219     } finally {
220         scanner.close();
221     }
222 }
223
224 private static List<String> splitLongLine(String line) {
225     List<String> result = new ArrayList<>();
226     while (line.length() > 80) {
227         int splitAt = line.lastIndexOf(' ', 80);
228         if (splitAt <= 0) splitAt = 80;
229         result.add(line.substring(0, splitAt));
230         line = line.substring(splitAt).trim();
231     }
232     if (!line.isEmpty()) result.add(line);
233     return result;
234 }
235 }
```

Listing 1: MapReduceFiles.java

```

1 import pandas as pd
2 import seaborn as sns
3 import matplotlib.pyplot as plt
4
5 df = pd.read_csv('performance_results.csv')
6
7 def save_heatmap(metric, title, filename):
8     pivot = df.pivot(index='MapLines', columns='ReduceWords', values=metric)
9     plt.figure(figsize=(8, 6))
10    sns.heatmap(
11        pivot,
12        annot=True,
13        fmt="d",
14        cmap="RdYlGn_r",
15        cbar_kws={'label': 'Time (ms)'})
16    )
17    plt.title(title)
18    plt.ylabel("Lines per Map Thread")
19    plt.xlabel("Words per Reduce Thread")
20    plt.tight_layout()
21    plt.savefig(filename)
22    plt.close()
23    print(f"Saved: {filename}")
24
25 save_heatmap('TotalTime', 'Total Time (ms)', '../latex/images/total_time_heatmap.png')
26 save_heatmap('MapTime', 'Map Time (ms)', '../latex/images/map_time_heatmap.png')
27 save_heatmap('ReduceTime', 'Reduce Time (ms)', '../latex/images/reduce_time_heatmap.png')
```

Listing 2: plots.py