Name: Andrew Hayes
Student ID: 21321503
E-mail: a.hayes18@universityofgalway.ie

CT414

2025–03–24

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.

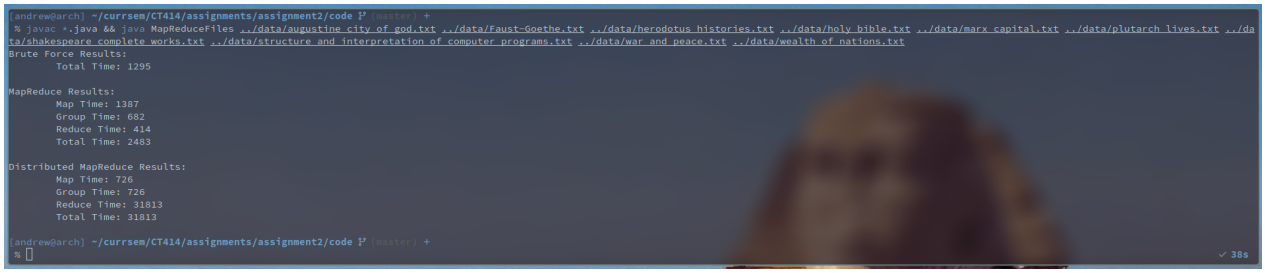

Figure 1: Baseline results for my list of files (in milliseconds)

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.

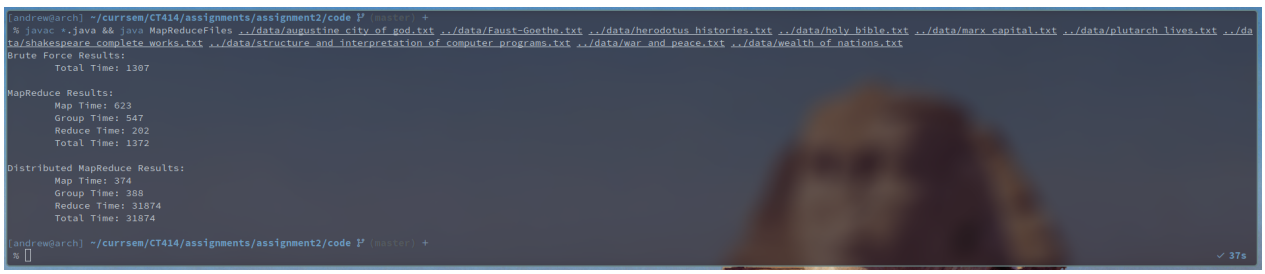

Figure 2: Baseline results with `ArrayList` update (in milliseconds)

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. Heatmaps that contain results pertaining only to the map phase and the reduce phase, as well as a table of the results from the CSV file can be found in the Appendix.

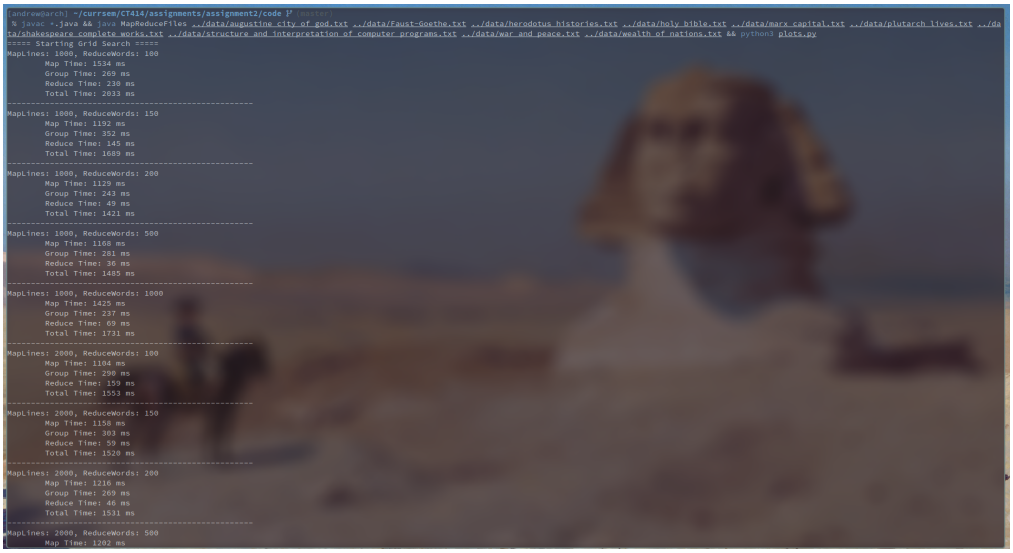

Figure 3: Running the grid-search and plotting the results

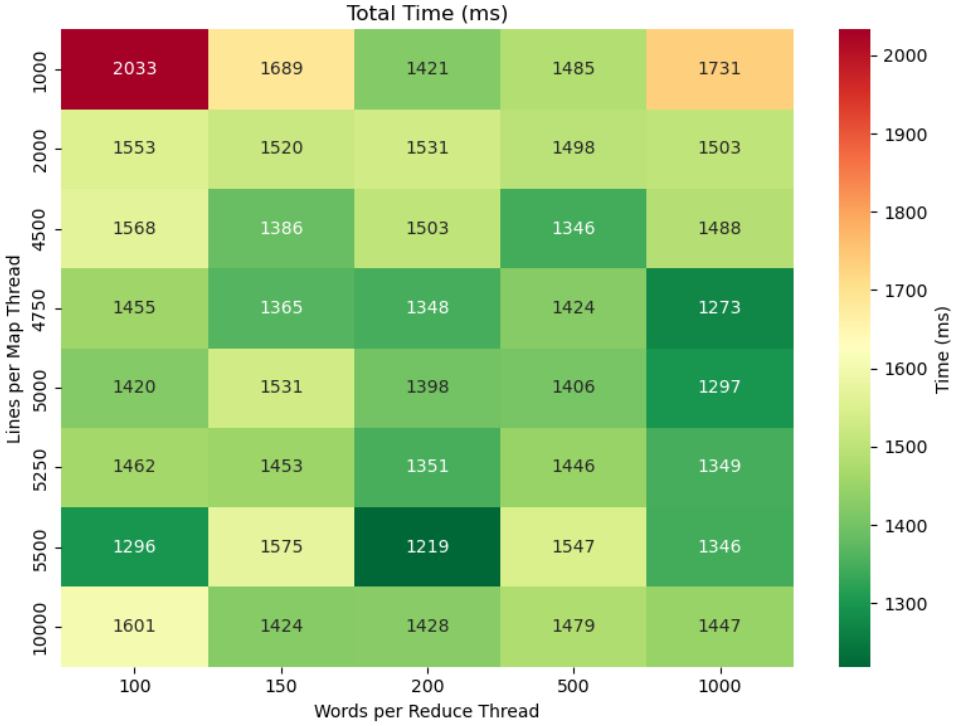

Figure 4: Heatmap of total time taken by each parameter combination

As can be seen from the heatmap above, the best result achieved by the distributed MapReduce approach was 1,219 millisec-

onds with the parameter combination of 200 words per reduce thread and 5,500 lines per map thread. Not only is this an improvement of over 2600% ($\frac{31874}{1219} \times 100 = 2614.766$), but it beats the original brute force result of 1,307 milliseconds and the non-distributed MapReduce result of 1,372 milliseconds. We can tell that this isn't a fluke, as this is consistent with other neighbouring values in the heatmap, with similar times of 1,296 milliseconds, 1,273 milliseconds, etc.

What I find most interesting about these results is that they are so similar to the brute force results; while the fully-optimised & tuned distributed MapReduce does beat brute force, it only does so by a narrow margin. I have a few ideas as to why this might be the case: it may be due to the brute force approach being more suitable to CPU caching due to its sequential approach, or due to the high thread overhead running it on the CPU of a single laptop, but I think that the main reason is that the dataset isn't actually very big. While 10 large books may seem to be quite a lot of data, MapReduce was created to deal with petabytes of data, not a few megabytes. I would hypothesise that the real performance benefits of distributed MapReduce would only become clear if the testing was repeated on at least a couple gigabytes of data.

4 Appendix: Source Code

```
import java.util.*;
import java.io.*;

public class MapReduceFiles {

  private static final String CSV_FILE = "performance_results.csv";

  public static void main(String[] args) {
    if (args.length < 1) {
      System.err.println("Usage: java MapReduceFiles file1.txt file2.txt ... fileN.txt");
      return;
    }

    Map<String, String> input = new HashMap<>();
    try {
      for (String filename : args) {
        input.put(filename, readFile(filename));
      }
    } catch (IOException ex) {
      System.err.println("Error reading files: " + ex.getMessage());
      ex.printStackTrace();
      return;
    }

    int[] mapSizes = {1000, 2000, 4500, 4750, 5000, 5250, 5500, 10000};
    int[] reduceSizes = {100, 150, 200, 500, 1000};

    System.out.println("===== Starting Grid Search =====");

    try (PrintWriter writer = new PrintWriter(new FileWriter(CSV_FILE))) {
      writer.println("MapLines,ReduceWords,MapTime,GroupTime,ReduceTime,TotalTime");

      for (int mapSize : mapSizes) {
        for (int reduceSize : reduceSizes) {
          runDistributedMapReduce(input, mapSize, reduceSize, writer);
        }
      }

    } catch (IOException e) {
      System.err.println("Error writing to CSV file: " + e.getMessage());
    }

    System.out.println("===== Grid Search Complete =====");
    System.out.println("Results saved to: " + CSV_FILE);
  }

```

```
public static void runDistributedMapReduce(Map<String, String> input, int linesPerMapThread, int
↪  wordsPerReduceThread, PrintWriter csvWriter) {
  final Map<String, Map<String, Integer>> output = new HashMap<>();

  // MAP Phase
  long mapStartTime = System.currentTimeMillis();
  List<MappedItem> mappedItems = Collections.synchronizedList(new ArrayList<>());

  final MapCallback<String, MappedItem> mapCallback = new MapCallback<>() {
    public synchronized void mapDone(String file, List<MappedItem> results) {
      mappedItems.addAll(results);
    }
  };

  List<Thread> mapCluster = new ArrayList<>();
  for (Map.Entry<String, String> entry : input.entrySet()) {
    final String file = entry.getKey();
    final String[] lines = entry.getValue().split("\\r?\\n");

    for (int i = 0; i < lines.length; i += linesPerMapThread) {
      int end = Math.min(i + linesPerMapThread, lines.length);
      final List<String> chunk = new ArrayList<>();
      for (int j = i; j < end; j++) {
        chunk.addAll(splitLongLine(lines[j]));
      }

      Thread t = new Thread(() -> map(file, chunk, mapCallback));
      mapCluster.add(t);
      t.start();
    }
  }

  for (Thread t : mapCluster) {
    try {
      t.join();
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }
  }

  long mapTotalTime = System.currentTimeMillis() - mapStartTime;

  // GROUP Phase
  long groupStartTime = System.currentTimeMillis();
  Map<String, List<String>> groupedItems = new HashMap<>();
  for (MappedItem item : mappedItems) {
    groupedItems.computeIfAbsent(item.getWord(), k -> new ArrayList<>()).add(item.getFile());
  }
  long groupTotalTime = System.currentTimeMillis() - groupStartTime;

  // REDUCE Phase
  long reduceStartTime = System.currentTimeMillis();
  final ReduceCallback<String, String, Integer> reduceCallback = (word, result) -> {
    synchronized (output) {
      output.put(word, result);
    }
  };

  List<Thread> reduceCluster = new ArrayList<>();
  List<Map<String, List<String>>> reduceChunks = new ArrayList<>();
  Map<String, List<String>> currentChunk = new HashMap<>();
```

```
    int count = 0;

    for (Map.Entry<String, List<String>> entry : groupedItems.entrySet()) {
      currentChunk.put(entry.getKey(), entry.getValue());
      count++;
      if (count >= wordsPerReduceThread) {
        reduceChunks.add(currentChunk);
        currentChunk = new HashMap<>();
        count = 0;
      }
    }
    if (!currentChunk.isEmpty()) reduceChunks.add(currentChunk);

    for (final Map<String, List<String>> chunk : reduceChunks) {
      Thread t = new Thread(() -> {
        for (Map.Entry<String, List<String>> entry : chunk.entrySet()) {
          reduce(entry.getKey(), entry.getValue(), reduceCallback);
        }
      });
      reduceCluster.add(t);
      t.start();
    }

    for (Thread t : reduceCluster) {
      try {
        t.join();
      } catch (InterruptedException e) {
        throw new RuntimeException(e);
      }
    }

    long reduceTotalTime = System.currentTimeMillis() - reduceStartTime;
    long totalTime = mapTotalTime + groupTotalTime + reduceTotalTime;

    // Print & Log
    System.out.println("MapLines: " + linesPerMapThread + ", ReduceWords: " + wordsPerReduceThread);
    System.out.println("\tMap Time: " + mapTotalTime + " ms");
    System.out.println("\tGroup Time: " + groupTotalTime + " ms");
    System.out.println("\tReduce Time: " + reduceTotalTime + " ms");
    System.out.println("\tTotal Time: " + totalTime + " ms");
    System.out.println("-----------------------------------------------------");

    csvWriter.printf("%d,%d,%d,%d,%d,%d%n",
            linesPerMapThread, wordsPerReduceThread,
            mapTotalTime, groupTotalTime, reduceTotalTime, totalTime);
    csvWriter.flush();
  }

  public static void map(String file, List<String> lines, MapCallback<String, MappedItem> callback) {
    List<MappedItem> results = new ArrayList<>();
    for (String line : lines) {
      String[] words = line.trim().split("\\s+");
      for (String word : words) {
        word = word.replaceAll("[^a-zA-Z]", "").toLowerCase();
        if (!word.isEmpty()) {
          results.add(new MappedItem(word, file));
        }
      }
    }
    callback.mapDone(file, results);
  }
```

```
168
169   public static void reduce(String word, List<String> list, ReduceCallback<String, String, Integer>
      ↪  callback) {
170     Map<String, Integer> reducedList = new HashMap<>();
171     for (String file : list) {
172       reducedList.put(file, reducedList.getOrDefault(file, 0) + 1);
173     }
174     callback.reduceDone(word, reducedList);
175   }
176
177   public interface MapCallback<E, V> {
178     void mapDone(E key, List<V> values);
179   }
180
181   public interface ReduceCallback<E, K, V> {
182     void reduceDone(E e, Map<K, V> results);
183   }
184
185   private static class MappedItem {
186     private final String word;
187     private final String file;
188
189     public MappedItem(String word, String file) {
190       this.word = word;
191       this.file = file;
192     }
193
194     public String getWord() {
195       return word;
196     }
197
198     public String getFile() {
199       return file;
200     }
201
202     @Override
203     public String toString() {
204       return "[\"" + word + "\",\"" + file + "\"]";
205     }
206   }
207
208   private static String readFile(String pathname) throws IOException {
209     File file = new File(pathname);
210     StringBuilder fileContents = new StringBuilder((int) file.length());
211     Scanner scanner = new Scanner(new BufferedReader(new FileReader(file)));
212     String lineSeparator = System.getProperty("line.separator");
213
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);
```

```
      if (splitAt <= 0) splitAt = 80;
      result.add(line.substring(0, splitAt));
      line = line.substring(splitAt).trim();
    }
    if (!line.isEmpty()) result.add(line);
    return result;
  }
}
```

Listing 1: `MapReduceFiles.java`

```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

df = pd.read_csv('performance_results.csv')

def save_heatmap(metric, title, filename):
    pivot = df.pivot(index='MapLines', columns='ReduceWords', values=metric)
    plt.figure(figsize=(8, 6))
    sns.heatmap(
        pivot,
        annot=True,
        fmt="d",
        cmap="RdYlGn_r",
        cbar_kws={'label': 'Time (ms)'}
    )
    plt.title(title)
    plt.ylabel("Lines per Map Thread")
    plt.xlabel("Words per Reduce Thread")
    plt.tight_layout()
    plt.savefig(filename)
    plt.close()
    print(f"Saved: {filename}")

save_heatmap('TotalTime', 'Total Time (ms)', '../latex/images/total_time_heatmap.png')
save_heatmap('MapTime', 'Map Time (ms)', '../latex/images/map_time_heatmap.png')
save_heatmap('ReduceTime', 'Reduce Time (ms)', '../latex/images/reduce_time_heatmap.png')
```

Listing 2: `plots.py`

Map Lines	Reduce Words	Map Time (ms)	Group Time (ms)	Reduce Time (ms)	Total Time (ms)
1000	100	1534	269	230	2033
1000	150	1192	352	145	1689
1000	200	1129	243	49	1421
1000	500	1168	281	36	1485
1000	1000	1425	237	69	1731
2000	100	1104	290	159	1553
2000	150	1158	303	59	1520
2000	200	1216	269	46	1531
2000	500	1202	260	36	1498
2000	1000	1202	264	37	1503
4500	100	1145	252	171	1568
4500	150	1089	238	59	1386
4500	200	995	294	214	1503
4500	500	863	295	188	1346
4500	1000	1183	250	55	1488
4750	100	1016	267	172	1455
4750	150	1078	228	59	1365
4750	200	1041	260	47	1348
4750	500	1110	278	36	1424
4750	1000	975	263	35	1273
5000	100	1069	277	74	1420
5000	150	1253	224	54	1531
5000	200	874	306	218	1398
5000	500	1118	252	36	1406
5000	1000	1006	244	47	1297
5250	100	1165	225	72	1462
5250	150	1008	272	173	1453
5250	200	1054	250	47	1351
5250	500	1134	275	37	1446
5250	1000	1027	280	42	1349
5500	100	976	249	71	1296
5500	150	1062	285	228	1575
5500	200	882	290	47	1219
5500	500	1254	257	36	1547
5500	1000	999	308	39	1346
10000	100	1167	360	74	1601
10000	150	1093	270	61	1424
10000	200	1075	296	57	1428
10000	500	1161	279	39	1479
10000	1000	1154	255	38	1447

Table 1: Results written to `performance_results.csv`

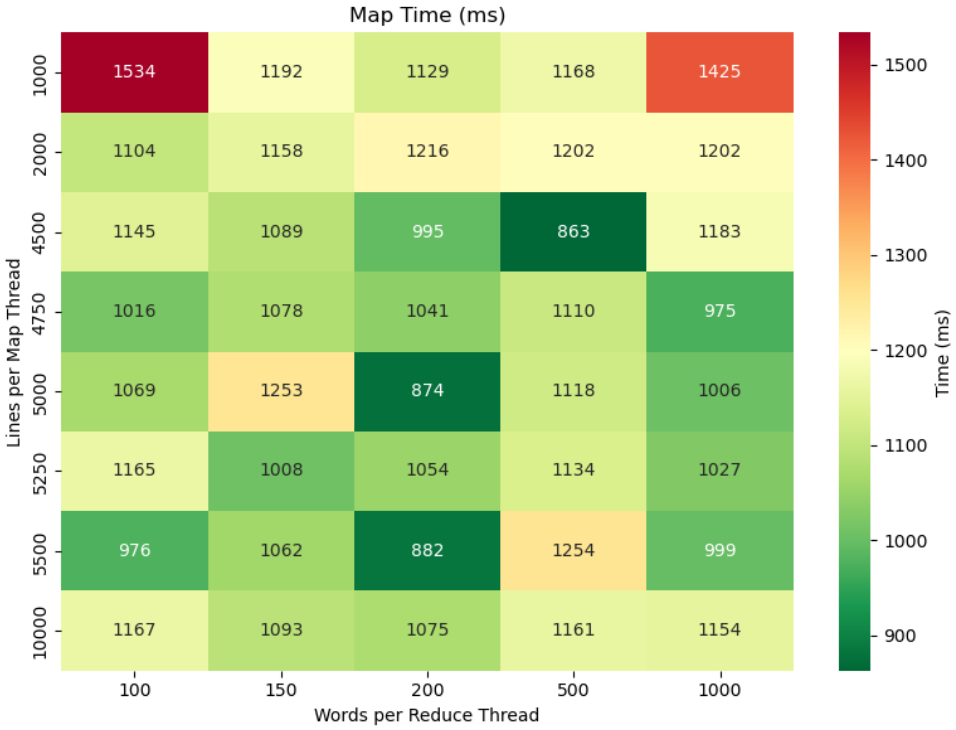

Figure 5: Heatmap of time taken during the map phase by each parameter combination

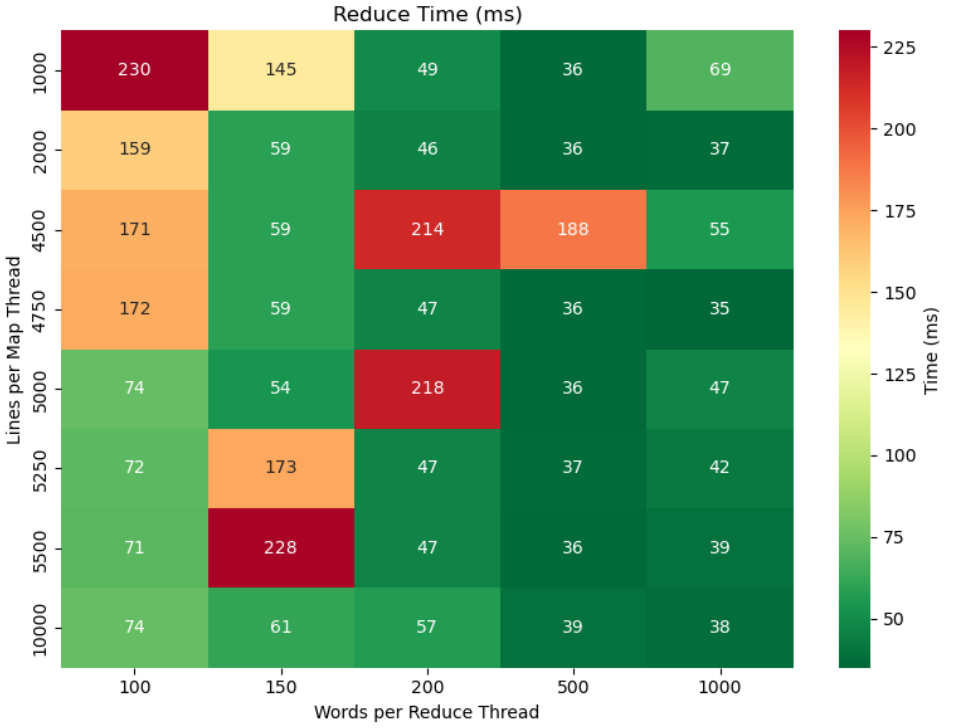

Figure 6: Heatmap of time taken during the reduce phase by each parameter combination