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.

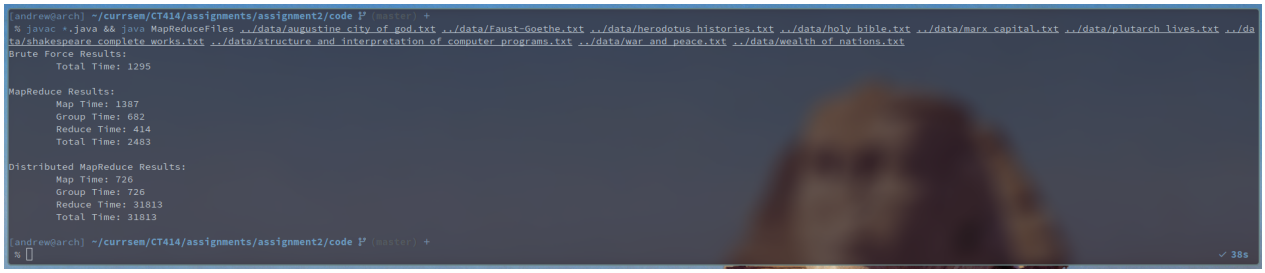

Figure 1: Baseline results for my list of files (in milliseconds)

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.

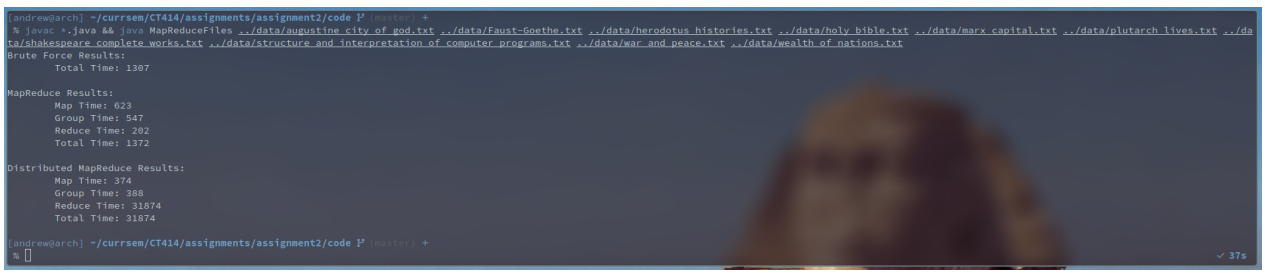

Figure 2: Baseline results with `ArrayList` update (in milliseconds)

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.

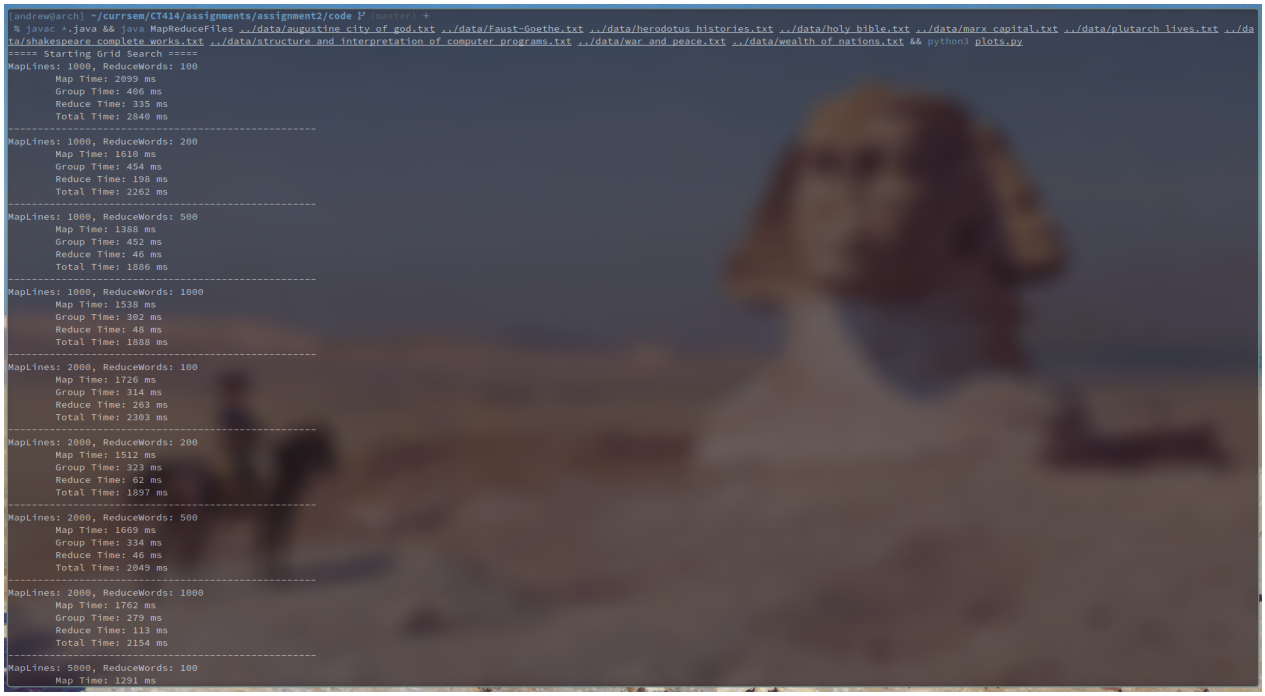

Figure 3: Running the grid-search and plotting the results

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`

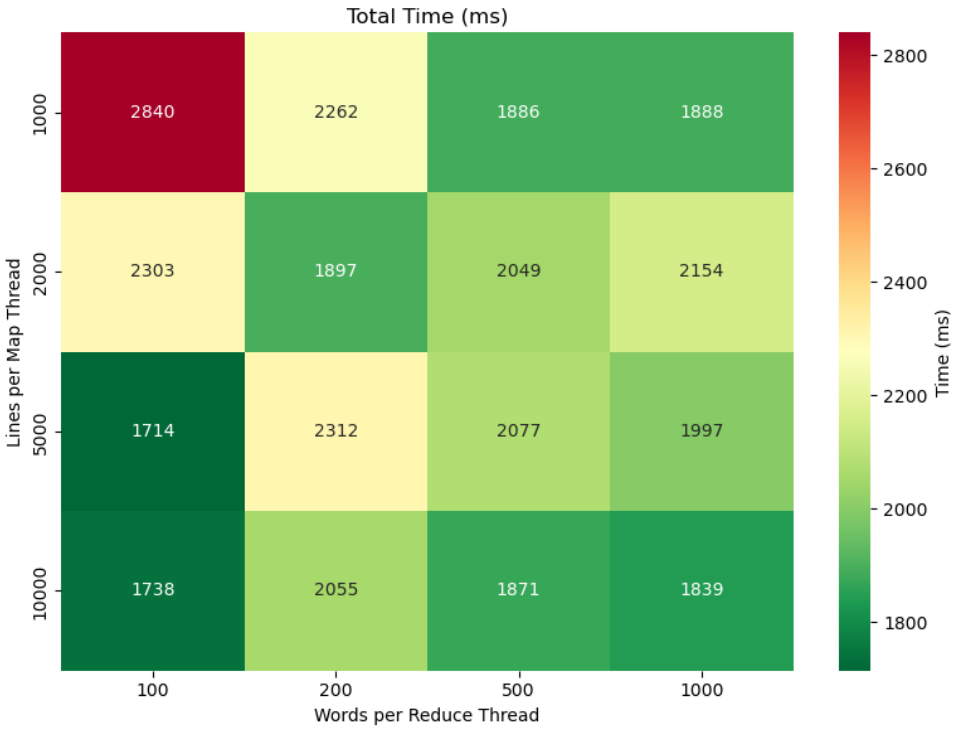

Figure 4: Heatmap of total time taken by each parameter combination

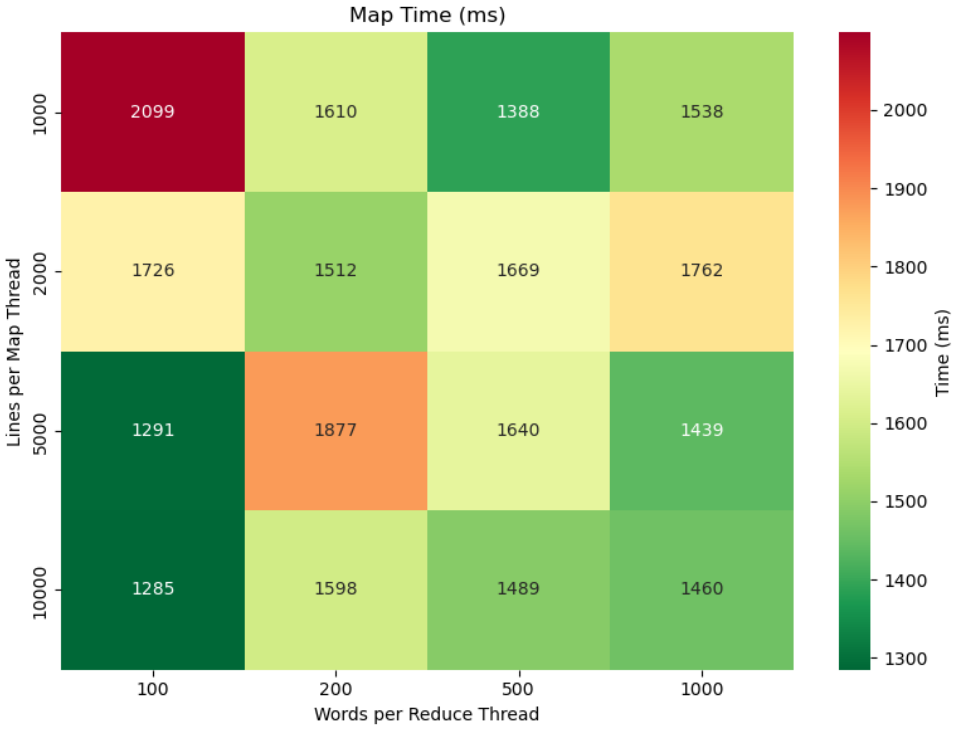

Figure 5: Heatmap of time taken during the map phase by each parameter combination

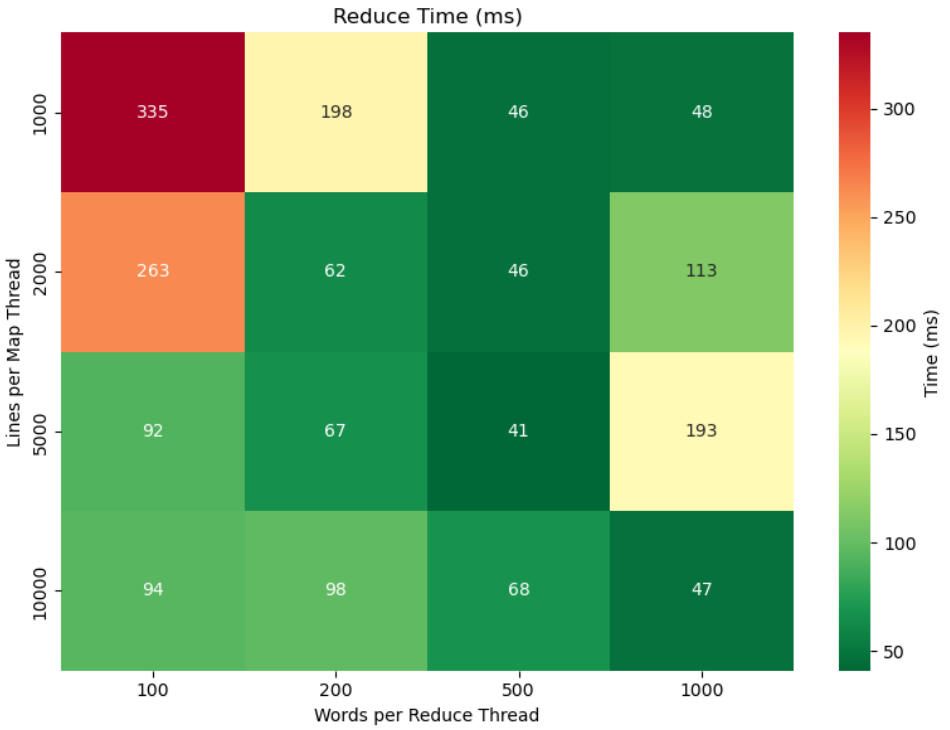

Figure 6: Heatmap of time taken during the reduce phase by each parameter combination

4 Appendix: Source Code

```
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, 5000, 10000};
    int[] reduceSizes = {100, 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();
  }
```

```
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>
    ↪  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
```

```
    try {
      while (scanner.hasNextLine()) {
        fileContents.append(scanner.nextLine()).append(lineSeparator);
      }
      return fileContents.toString();
    } finally {
      scanner.close();
    }
  }

  private static List<String> splitLongLine(String line) {
    List<String> result = new ArrayList<>();
    while (line.length() > 80) {
      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`

```
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`