# A Concurrent Implementation of MapReduce

CS5600 FINAL PROJECT

DECEMBER 16, 2021

BY: MITCHELL LAWSON

# MOTIVATION

- Redemption for the dreaded File Systems assignment
- Redemption from the concurrent Hash Table assignment
- Learn more about concurrency
- Attempt something hard and very interesting
- Prepare for Scalable Distributed Systems → MapReduce, created by engineers at Google for large scale parallel data processing
  - It makes programming on large-scale clusters easier for developers by alleviating the worries of managing parallelism, crashes and other issues associated with clusters of machines.

# PROJECT DETAILS

- Project is defined in the OSTEP repository:
  - https://github.com/remzi-arpacidusseau/ostep-projects/tree/master/concurrency-mapreduce.
- A simplification that is made for this project as suggested by the outline given is creating MapReduce for a single machine. There are 3 learning objectives for this personal end of term project.
  1. Learn about and understand the nature of the MapReduce paradigm.
  2. Implement a correct and efficient version of MapReduce that utilizes threads.
  3. Gain more experience and familiarity creating concurrent code.
- Will be used/implemented in the context of word counts

# WELL… WHAT ACTUALLY IS MAPREDUCE?

- MapReduce can perform distributed and parallel computations using large datasets across a large number of nodes
- A MapReduce job usually splits the input datasets and then process each of them independently by the Map tasks in a completely parallel manner… BUT in my implementation this is done in a concurrent manner

# THINK OF THE FOLLOWING COMPONENTS

1. Mapper
   - Some logic is applied to ”n” data blocks spread across different node.
   - The data blocks in my implementation are words and the nodes are files.
2. Shuffle and Sort
   - The output of the mapper then gets sorted
   - Repeated values get removed and the values get grouped together based on keys
   - This was done using quick sort and a concurrent hash table/ map
3. Reducer
   - The output from the previous step is used in this phase.
   - “Consolidates” the outputs and creates the finalized product.
   - Counts the words in my case.
4. Combiner (Optional)
   - Provides performance increase by combining inputs from the same node.

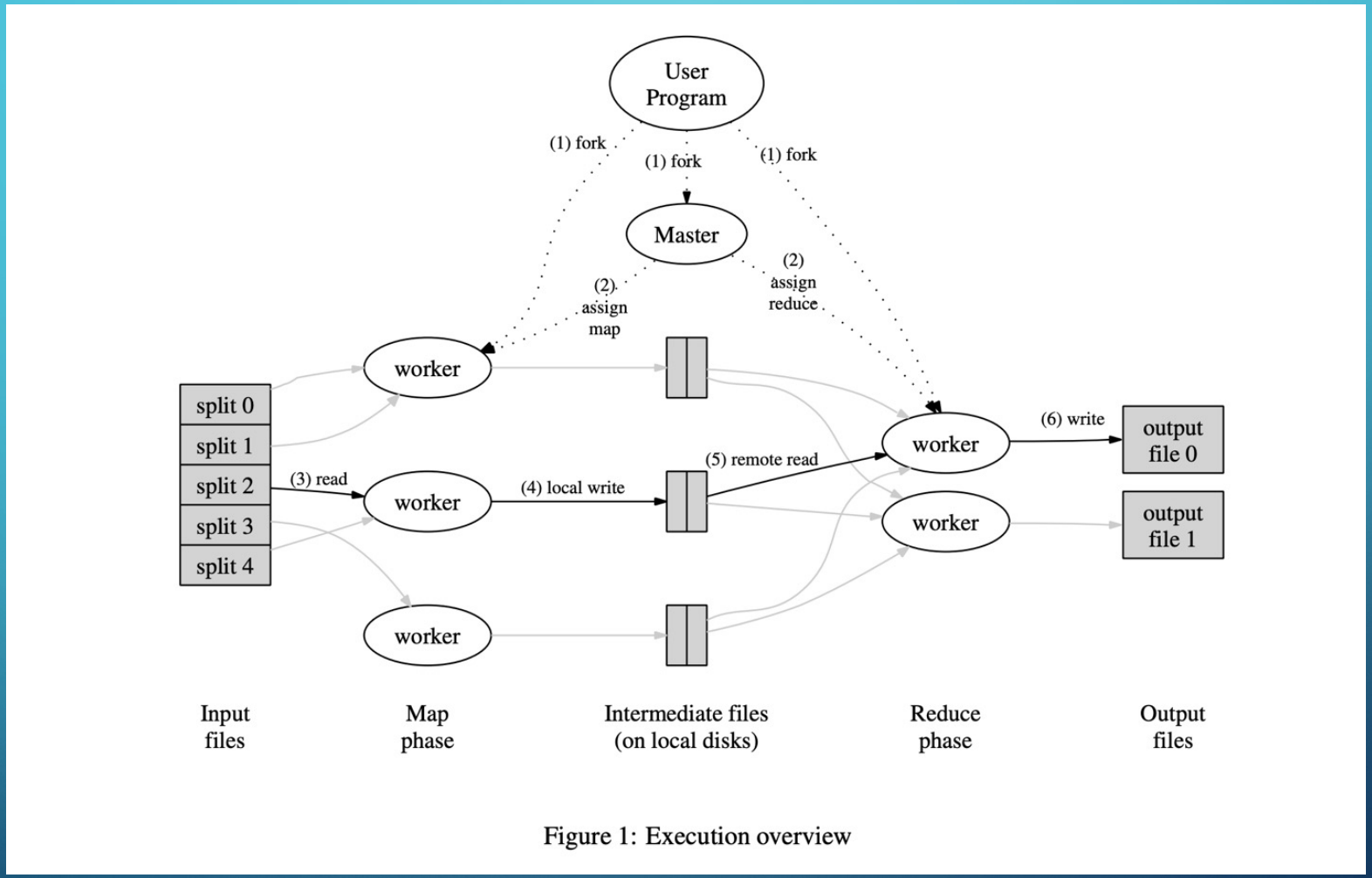


Figure 1: Execution overview

# THE PROVIDED INTERFACE

Concurrent_MapReduce – mapreduce.h

```h
#ifndef __mapreduce_h__
#define __mapreduce_h__

// Different function pointer types used by MR
typedef char *(*Getter)(char *key, int partition_number);
typedef void (*Mapper)(char *file_name);
typedef void (*Reducer)(char *key, Getter get_func, int partition_number);
typedef unsigned long (*Partitioner)(char *key, int num_partitions);

// External functions: these are what you must define
void MR_Emit(char *key, char *value);

unsigned long MR_DefaultHashPartition(char *key, int num_partitions);

void MR_Run(int argc, char *argv[],
        Mapper map, int num_mappers,
        Reducer reduce, int num_reducers,
        Partitioner partition);

#endif // __mapreduce_h__

```

# WORD COUNTING

```
12  void Map(char *file_name) {
13      FILE *fp = fopen(file_name, mode: "r");
14      assert(fp != NULL);
15
16      char* line = NULL;
17      size_t size = 0;
18      while (getline(&line, &size, fp) != -1) {
19          char *token, *dummy = line;
20          while ((token = strsep(&dummy, delim: " \t\n\r")) != NULL) {
21              MR_Emit(token, value: "1");
22          }
23      }
24
25      fclose(fp);
26  }
27
28  void Reduce(char *key, Getter get_next, int partition_number) {
29      int count = 0;
30      char *value;
31      value = get_next(key, partition_number);
32      while (value != NULL) {
33          count++;
34          value = get_next(key, partition_number);
35      }
36      printf("%s %d\n", key, count);
37  }
38
39  int main(int argc, char *argv[]) {
40      MR_Run(argc, argv, Map, num_mappers: 10, Reduce, num_reducers: 10, MR_DefaultHashPartition);
41  }
42
```

# TESTS

- There are many different areas of my implementation that I wanted to test, however due to time constraints I had to narrow it down to two sets of testing involving…
- Ensuring the correct output
  1. 1 file containing non-repeating words.
  2. 1 file containing repeating and non-repeating words.
  3. Multiple files containing repeating, non-repeating words and overlapping words.
- The timing of different file contents with my Reducer
  1. Best Case: 1 file all the same word
  2. Average Case: 1 file with a real-world example from free published text.
  3. Worst Case: All different words.

# RESULTS & ANALYSIS

Timing tests:

1. Best Case: 1 file all the same word.
2. Average Case: 1 file with a real-world example from free published text.
3. Worst Case: All different words.

| Test | Time (s) |
|---|---|
| **1 (Best)** | 0.025456, 0.026044, 0.024939 |
| **2 (Average)** | 0.223773, 0.199620, 0.192057 |
| **3 (Worst)** | 1.093060, 0.953125, 0.912240 |

- The timing of different file contents with my Reducer
  1. Best Case: 3 files, 10 mapping threads
  2. Average Case: 3 files, 3 mapping threads
  3. Worst Case: 3 files, 2 mapping threads

| Test | Time (s) |
|---|---|
| **1 (Best)** | 1.005772 |
| **2 (Average)** | 1.018589 |
| **3 (Worst)** | 1.031590 |

- The timing of different file contents with my Reducer
  1. Best Case: 3 files, 2 reducing threads
  2. Average Case: 3 files, 3 reducing threads
  3. Worst Case: 3 files, 10 reducing threads

| Test | Time (s) |
|---|---|
| **1 (Best)** | 0.530489 |
| **2 (Average)** | 0.878015 |
| **3 (Worst)** | 1.047338 |

# BENEFITS

- Fault Tolerance
- Resilience
- Fast
- Parallel Processing
- Availability
- Scalability
- Cost Effective