**CS1760: Multiprocessor Synchronization** Due 10 December 2023

# Project 4: MapReduce

*Professor: Maurice Herlihy* *TA: cs1760tas@lists.brown.edu*

## 1 Introduction

`MapReduce` is a programming framework designed for handling large-scale data processing tasks in a parallel and distributed computing environment.

A `MapReduce` program first divides data into independent fragments and assigns each fragment to a `Mapper` thread, which produces a list of key-value pairs. The framework then collects the outputs of all `Mapper` and merges all values paired with each key into a list. Each key-value list pair is assigned to a `Reducer` thread, which produces an application-specific output for that key, and lastly, all outputs are merged into a map as the final output of the program.

In this assignment, you will implement the `MapReduce` framework and two sample programs — word count and inverted index — that make use of the framework.

## 2 Stencil

To spare you from delving into the complex and less engaging Java APIs, we offer you stencil code, which can be found HERE. The following provides a description of the structures of the stencil:

- The `mapreduce` package (under `mapreduce/`) includes the main implementation of the `MapReduce` framework.
- The `FileParser.java` class includes static helper methods that parses a file into a list of words and splits them into list of list of words if needed.
- The `WordCount.java` class implements a sample `MapReduce` program that counts the number of occurrences of each word in a file.
- The `InvertedIndex.java` class implements another sample `MapReduce` program that takes in a list of file and outputs the name of the files each word appears in.
- The `Test.java` class allows you to test your implementations by running `WordCount` and `InvertedIndex` with your own sample input files and comparing to your expected output. You can run it using the `make` command. Note that testing is not graded in this assignment.

## 3 Requirements

Within the stencil code, you'll encounter comments marked with `TODO`, indicating areas where modifications are required. Please do not modify any existing function signatures or use any synchronization primitives.

1. `MapReduce.java`: the main implementation of the `MapReduce` framework, which should be consistent with the logic described in the Introduction, the lectures, and the textbook.
    - `call()`: starts the execution of a `MapReduce` instance and outputs the final aggregated results

2. `WordCount.java`: a `MapReduce` program that reads in all words in a file and produces a map from each word to its number of occurrences.
    - `run()` takes in a filename, creates a `MapReduce` instance, and utilizes it to compute the final output
    - `Mapper.compute()` overrides the map function
    - `Reducer.compute()` overrides the reduce function
3. `InvertedIndex.java`: a `MapReduce` program that reads in a list of files and produces a map from each word to a list of file it appears in.
    - `run()` takes in a list of files, creates a `MapReduce` instance, and utilizes it to compute the final output
    - `Mapper.compute()` overrides the map function
    - `Reducer.compute()` overrides the reduce function

# 4 Tips

Here are several useful methods that you might need in your implementation:

- `Supplier<T>.get()` returns an instance of `T`. To create a `Supplier<T>`, simply use Java lambda expression `() -> new T();`
- `ForkJoinPool.execute()` takes in a `ForkJoinTask`, which is extended by both `Mapper` and `Reducer`, and arranges to start the execution of the input task.
- `ForkJoinTask<T>.join()` waits for the task to finish execution and returns its output as type `T`. Again, the `Mapper<IN,K,V>` class extends `ForkJoinTask<Map<K,V>>` and the `Reducer<K,V,OUT>` class extends `ForkJoinTask<OUT>`
- When overriding the `compute()` methods of `Mapper` and `Reducer`, you can directly access their public/protected global variables, such as `input` in `Mapper` and `key` and `valueList` in `Reducer`

# 5 README

Please include a README.md file in your submission with very brief answer (1-3 sentences) to each of the following questions:

1. What scheduling and work distribution technique does the Java's `ForkJoinPool` implement? How does it make parallel execution more efficient?
2. How does data locality apply to `MapReduce`? Does `MapReduce` experience the contention/-caching issues observed in some spin lock implementations on a cache-coherent system? What about in large distributed systems?
3. On that note, why don't we use any synchronization primitives in the implementation of `MapReduce` or any program that make use of `MapReduce`?
4. What is one limitation of `MapReduce`, and is it inherent to all data parallelism techniques?