Project 6: Map-Reduce

- **Due** Nov 22 by 5pm

- **Points** 100

In 2004, engineers at Google introduced a new paradigm for large-scale parallel data processing known as MapReduce (the original paper is here; (Links to an external site.) as a bonus, see that authors of the the citations at the end). MapReduce makes developers efficiently and easily program large-scale clusters, especially for tasks with a lot of data processing. With Map Reduce, the developer can focus on writing their task as a set of Map and Recue functions, and let the underlying infrastructure handle parallelism/concurrency, machine crashes, and other complexities common within clusters of machines.

In this project, you'll be building a simplified version of MapReduce for just a single machine. While it may seem simple to build MapReduce for a single machine, you will still face numerous challenges, mostly in building the correct concurrency support. You'll have to think about how to design your MapReduce implementation, and then build it to work efficiently and correctly.

There are three specific objectives to this assignment:

- To gain an understanding of the fundamentals of the MapReduce paradigm.
- To implement a correct and efficient MapReduce framework using threads and related functions.
- To gain experience designing and developing concurrent code.

Background

To understand how to make progress on any project that involves concurrency, you should understand the basics of thread creation, mutual exclusion (with locks), and signaling/waiting (with condition variables). These are described in the following book chapters:

- Intro to ThreadsLinks to an external site.

- Threads APILinks to an external site.
- LocksLinks to an external site.
- Using LocksLinks to an external site.
- Condition VariablesLinks to an external site.

Read these chapters carefully in order to prepare yourself for this project.

General Idea

Let’s now get into the exact code you’ll have to build. The MapReduce infrastructure you will build supports the execution of user-defined `Map()` and `Reduce()` functions.

As from the original paper: “`Map()`, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key K and passes them to the `Reduce()` function.”

“The `Reduce()` function, also written by the user, accepts an intermediate key K and a set of values for that key. It merges together these values to form a possibly smaller set of values; typically just zero or one output value is produced per `Reduce()` invocation. The intermediate values are supplied to the user’s reduce function via an iterator.”

A classic example, written here in **pseudocode** (note: your implementation will have different interfaces), shows how to count the number of occurrences of each word in a set of documents:

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediateValues(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseIntermediate(v);
    print key, result;
```

The following image from https://dzone.com/articles/word-count-hello-word-program-in-mapreduce (Links to an external site.)might help you understand the behavior of the MapReduce framework for this sample wordcount program.

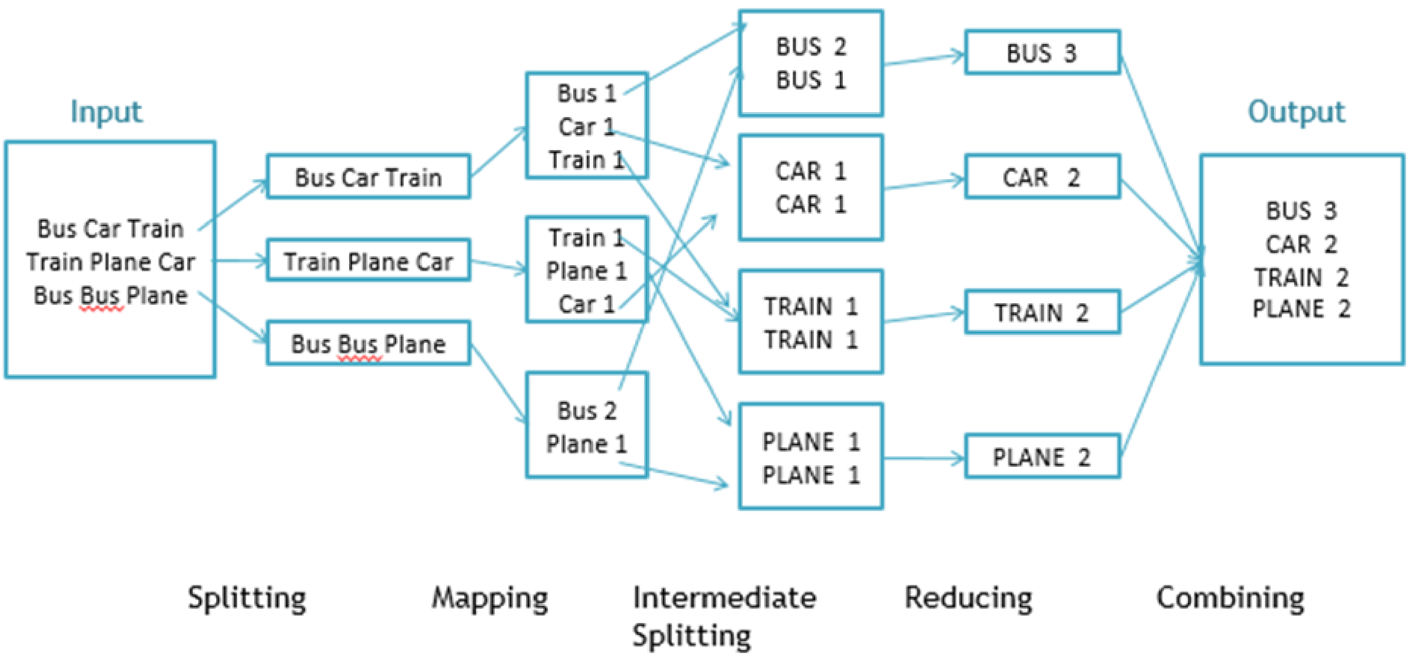

Fig. WorkFlow of MapReducing

What's fascinating about MapReduce is that so many different kinds of relevant computations can be mapped onto this framework. The original paper lists many examples, including word counting (as above), a distributed grep, a URL frequency access counters, a reverse web-link graph application, a term-vector per host analysis, and others.

What's also quite interesting is how easy it is to parallelize: many mappers can be running at the same time, and later, many reducers can be running at the same time. Users don't have to worry about how to parallelize their application; rather, they just write `Map()` and `Reduce()` functions and the infrastructure does the rest.

Code Overview

We give you here the `mapreduce.h` header file that specifies exactly what you must build in your MapReduce library:

```
#ifndef __mapreduce_h__ #define __mapreduce_h__  // Different function
 pointer types used by MR typedef char *(*Getter)(char *key, int parti
tion_number); typedef void (*Mapper)(char *file_name); typedef void (*
Reducer)(char *key, Getter get_func, int partition_number); typedef un
signed long (*Partitioner)(char *key, int num_partitions);  // Externa
l functions: these are what *you must implement* void MR_Emit(char *ke
y, char *value);  unsigned long MR_DefaultHashPartition(char *key, int
 num_partitions);

unsigned long MR_SortedPartition(char *key, int num_partitions);  void
 MR_Run(int argc, char *argv[],           Mapper map, int num_mappers,
           Reducer reduce, int num_reducers,      Partitioner partitio
n, int num_partitions);  #endif // __mapreduce_h__
```

The most important function is `MR_Run`, which takes the command line parameters of a given program, a pointer to a Map function (type `Mapper`, called `map`), the number of mapper threads your library should create (`num_mappers`), a pointer to a Reduce function (type `Reducer, called reduce`), the number of reducers (`num_reducers`), a pointer to a Partition function (`partition`, described below), and the number of partitions.

Thus, when a user is writing a MapReduce computation with your library, they will implement a Map function, implement a Reduce function, possibly implement a Partition function, and then call `MR_Run()`. The infrastructure will then create threads as appropriate and run the computation.

One basic assumption is that the library will create `num_mappers` threads (in a thread pool) that perform the map tasks. Another is that your library will create `num_reducers` threads (in a thread pool) to perform the reduction tasks. Finally, your library will create some kind of internal data structure to pass keys and values from mappers to reducers; more on this below.

Simple Example: Wordcount

Here is a simple (but functional) wordcount program, written to use this infrastructure:

```
#include <assert.h> #include <stdio.h> #include <stdlib.h> #include <s
tring.h> #include "mapreduce.h"  void Map(char *file_name) {    FILE *
fp = fopen(file_name, "r");    assert(fp != NULL);     char *line = NU
LL;    size_t size = 0;    while (getline(&line, &size, fp) != -1) {
      char *token, *dummy = line;        while ((token = strsep(&dumm
y, " \t\n\r")) != NULL) {            MR_Emit(token, "1");        }    }
     free(line);    fclose(fp); }  void Reduce(char *key, Getter get_n
ext, int partition_number) {    int count = 0;    char *value;    whi
le ((value = get_next(key, partition_number)) != NULL)        count++;
     printf("%s %d\n", key, count); }  int main(int argc, char *argv[])
 {    MR_Run(argc, argv, Map, 10, Reduce, 10, MR_DefaultHashPartition,
 10); }
```

Let's walk through this code, in order to see what it is doing and describe the requirements for this project.

First, notice that `Map()` is called with a file name. In general, we assume that this type of computation is being run over many files; each invocation of `Map()` is thus handed one file name and is expected to process that file in its entirety.

In this example, the code above just reads through the file, one line at a time, and uses `strsep()` to chop the line into tokens. Each token is then emitted using the `MR_Emit()` function, which takes two strings as input: a key and a value. The key here is the word itself, and the token is just a count, in this case, 1 (as a string). It then closes the file.

The `MR_Emit()` function is another key part of your library; it needs to take key/value pairs from the many different mappers and store them in a partition such that later reducers can access them, given constraints described below. Designing and implementing this data structure is thus a central challenge of the project.

After the mappers are finished, your library should have stored the key/value pairs in partitions such that the reducers can access them.   Your implementation must ensure that partition **i** is sent to a reducer before partition **i+1.**

`Reduce()` is invoked once per key, and is passed the key along with a function that enables iteration over all of the values that produced that same key. To iterate, the code just calls `get_next()` repeatedly until a NULL value is returned; `get_next` returns a pointer to the value passed in by the `MR_Emit()` function above, or NULL when the key's values have been processed. The output, in the example, is just a count of how many times a given word has appeared, and is just printed to standard output.

All of this computation is started off by a call to `MR_Run()` in the `main()` routine of the user program. This function is passed the `argv` array, and assumes that `argv[1]` … `argv[n-1]` (with `argc` equal to `n`) all contain file names that will be passed to the mappers.

One interesting function that you also need to pass to `MR_Run()` is the partitioning function. In most cases, programs will use the default function (`MR_DefaultHashPartition`), which should be implemented by your code. Here is its implementation:

```
unsigned long MR_DefaultHashPartition(char *key, int num_partitions)
{     unsigned long hash = 5381;     int c;     while ((c = *key++) != '\
0')         hash = hash * 33 + c;     return hash % num_partitions; }
```

The function's role is to take a given `key` and map it to a number, from `0` to `num_partitions - 1`. Its use is internal to the MapReduce library, but critical. Specifically, your MR library should use this function to decide which partition (and hence, which reducer thread) gets a particular key/list of values to process.   Note that the number of partitions can be greater (or smaller) than the number of reducers; thus, some reducers may need to handle multiple (or no) partitions.

For some applications, which reducer thread processes a particular key is not important (and thus the default function above should be passed in to `MR_Run()`); for others, it is, and this is why the user can pass in their own partitioning function as need be.

You will also need to write a new partitioning function, `MR_SortedPartition,` that ensures that keys are in a sorted order **across** the partitions (i.e., keys are not hashed into random partitions as in the default partition function).   For simplicity, you should assume that these keys can be represented with (converted to) a 32-bit unsigned integer. You should split keys across partitions such that if the keys are uniformly distributed across the 32-bit space, they will be divided uniformly across the partitions. For even more implementation simplicity, you can assume that the number of partitions is a power of 2 when the MR_SortedPartition function is used (big hint: the high bits of the key then indicate the partition number).

One last requirement: For each partition, keys (and the value list associated with said keys) should be sorted in ascending key order; thus, when a particular reducer thread (and its associated partition) are working, the `Reduce()` function should be called on each key in order for that partition. For this sort within a partition, the sorted order should be the lexical order provided by **strcmp()** over the original keys.

Considerations

Here are a few things to consider in your implementation:

- **Thread Management** This part is fairly straightforward. You should create `num_mappers` mapping threads, and assign a file to each `Map()` invocation in some manner you think is best (e.g., Round Robin, Shortest-File-First, etc.). Which way might lead to best performance? You should also create `num_reducers` reducer threads at some point, to work on the map'd output.
- **Partitioning and Sorting** Your central data structure should be concurrent, allowing mappers to each put values into different partitions correctly and efficiently. Once the mappers have completed, a sorting phase should order the key/value-lists. Then, finally, each reducer thread should start calling the user-defined `Reduce()` function on the keys in sorted order per partition. You should think about what type of locking is needed throughout this process for correctness.
- **Memory Management** One last concern is memory management. The `MR_Emit()` function is passed a key/value pair; it is the responsibility of the MR library to make copies of each of these. However, avoid making too many copies of them since the goal is to design an efficient concurrent data structure. Then, when the entire mapping and reduction is complete, it is the responsibility of the MR library to free everything.
- You could use data structures and APIs only from `stdio.h`, `stdlib.h`, `pthread.h`, `string.h`, `sys/stat.h`, and `semaphore.h`. If you want to implement some other data structures you can add that as a part of your submissions.

Testing

Information provided early next week.

Grading

We will verify that your code is not similar to other's in the class (both this semester and previous semesters) with our automated code checking tools. Code that is judged to be similar to others will cause you and your project partner to receive a harsh penalty for your project grades.

It will be compiled with test applications with the `-Wall -Werror -pthread -O` flags; it will also be **valgrind**ed to check for memory errors.

Your code will first be measured for correctness, ensuring that it performs the maps and reductions correctly. If you pass the correctness tests, your code will be tested for performance to see if it runs within suggested time limits.

If you do not think you will be able to fully implement the entire specification, we recommend thinking about implementing functionality in the following order.   Note that implementing each of these steps separately is likely to NOT be the most efficient way for reaching a complete solution, but implementing these as steps will enable you to pass some number of tests.

1. Implement all interfaces correctly, but without any concurrency; that is, your code does not create any threads and only processes a single argument (a single file, in the example above).   Thus, you will only be able to handle **MR_Run(2, "program arg1", Map, 1, Reduce, 1, MR_DefaultHashPartition, 1).**
2. Still have only one thread, but now add the ability to handle multiple input files; that is, you can handle arbitrary values for argv, as in **MR_Run(argc, argv, Map, 1, Reduce, 1, MR_DefaultHashPartition, 1).**
3. Create multiple mapper threads, where the number of mapper threads is equal to the number of input files. **MR_Run(argc, argv, Map, argc-1, Reduce, 1, MR_DefaultHashPartition, 1).**
4. Create multiple reducer threads.   Make the number of partitions equal to the number of reducer threads. **MR_Run(argc, argv, Map, argc-1, Reduce, R, MR_DefaultHashPartition, R).**
5. Remove the requirement that the number of mapper threads matches the number of input parameters; that is, one mapper thread end up being responsible for multiple input files (or no input files:) **MR_Run(argc, argv, Map, M, Reduce, R, MR_DefaultHashPartition, R).**
6. Remove the requirement that the number of reducer threads matches the number of partitions; that is, one reducer thread end up being responsible for multiple partitions (or no partitions:) **MR_Run(argc, argv, Map, M, Reduce, R, MR_DefaultHashPartition, P).**
7. Finally, work on optimizing the performance of your framework for different workloads.   Think about the allocation of work to different threads: how should you allocate files (of potentially different sizes) to different mappers?   how you should allocate partitions (of potentially different sizes) to different reducers?   Does performance change with different pthread scheduling policies?   Are you keeping your critical sections as small as possible?   Is your sort efficient?

Submitting Your Implementation

To handin your code, you should only need to submit `mapreduce.c` and any new .h files you've created in your P6 directory.   You should not divide your code into any other .c files.   You should create a **src** directory or any other subdirectores.   Your file **mapreduce.c** should implement: `get_next()`, `MR_Emit()`, `MR_Run(), MR_DefaultHashPartition(), and MR_SortedPartition().`

We should be able to compile mapreduce.c with our test applications with the `-Wall -Werror -pthread -O` flags.

We use automated grading scripts, so you must adhere to these instructions and formats if you want your project to be graded correctly.

If you choose to work in pairs, ...

Your files should be directly copied to `~cs537-1/handin/<cs-login>/p6/` directory.   Having subdirectories in `<cs-login>/p6/` is **not acceptable**.

Collaboration

This project is to be done in groups of size one or two (not three or more). Now, within a group, you can share as much as you like. However, copying code across groups is considered cheating.

Again, we will verify that your code is not similar to other's in the class (both this semester and previous semesters) with our automated code checking tools.   Code that is judged to be similar to others will cause you and your project partner to receive a harsh penalty for your project grades.

If you are planning to use `git` or other version control systems (*which are highly recommended for this project*), just be careful **not** to put your codes in a public repository.