Map Reduce

In 2004, engineers at Google introduced a new paradigm for large-scale parallel data processing known as MapReduce. One key aspect of MapReduce is that it makes programming such tasks on large-scale clusters easy for developers; instead of worrying about how to manage parallelism, handle machine crashes, and many other complexities common within clusters of machines, the developer can instead just focus on writing little bits of code (described below) and the infrastructure handles the rest.

In this project, you'll be building a simplified version of MapReduce for just a single machine. While somewhat easier to build MapReduce for a single machine, there are still numerous challenges, mostly in building the correct concurrency support. Thus, you'll have to think a bit about how to build the MapReduce implementation, and then build it to work efficiently and correctly.

There are three specific objectives to this assignment:

- To learn about the general nature of the MapReduce paradigm.
- To implement a correct and efficient MapReduce framework using threads and related functions.
- To gain more experience writing concurrent code.

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, 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:
        EmitIntermediate(w, "1");

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

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 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__
```

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`), and finally, a pointer to a Partition function (`partition`, described below).

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 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 <string.h>
#include "mapreduce.h"

void Map(char *file_name) {
    FILE *fp = fopen(file_name, "r");
    assert(fp != NULL);

    char *line = NULL;
    size_t size = 0;
    while (getline(&line, &size, fp) != -1) {
        char *token, *dummy = line;
        while ((token = strsep(&dummy, " \t\n\r")) != NULL) {
            MR_Emit(token, "1");
        }
    }
    free(line);
    fclose(fp);
}

void Reduce(char *key, Getter get_next, int partition_number) {
    int count = 0;
    char *value;
    while ((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);
}
```

Let's walk through this code, in order to see what it is doing. 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 thus another key part of your library; it needs to take key/value pairs from the many different mappers and store them in a way 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 such a way that the `Reduce()` function can be called. `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. 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.

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.

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. Then, when the entire mapping and reduction is complete, it is the responsibility of the MR library to free everything.

Grading

Your code should turn in `mapreduce.c` which implements the above functions correctly and efficiently. It will be compiled with test applications with the `-Wall -Werror -pthread -O` flags; it will also be valgrinded 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; higher performance will lead to better scores.