CS 537 Spring 2020, Project 4a: MapReduce

NEW

- Deadline extended to April 3
- Minor edit in spec to rename CombinerGetter to CombineGetter.
- Minor change in Reducer definition to include ReduceStateGetter
- Functions added in spec to make Eager mode easier. Please read the section on Eager mode carefully!

Teaming up!

For this project, you have the option to work with a partner. Read more details in Submitting Your Implementation and Collaboration.

Administrivia

- **Due Date** by ~~Apr 2, 2020~~ **April 3, 2020** at 10:00 PM
- Questions: We will be using Piazza for all questions.
- This project is best to be done on the lab machines (https://csl.cs.wisc.edu/services/instructional-facilities). Alternatively, you can create Virtual Machine and work locally (installation video ()). You learn more about programming in C on a typical UNIX-based platform (Linux).
- Please take the quiz on Canvas that will be posted on Piazza shortly. It will check if you have read the spec closely. NOTE: The quiz does not work in groups and will have to be taken individually on Canvas.
- Your group (i.e., you and your partner) will have *2* slip days for the rest of the semester. If you are changing partners please talk to the instructor about it.

In 2004, engineers at Google introduced a new paradigm for large-scale parallel data processing known as MapReduce (see the original paper here (https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf), and make sure to look in the citations at the end!). One key aspect of MapReduce is that it makes programming 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 it is 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.

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 Threads (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf)
- Threads API (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf)
- Locks (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf)
- Using Locks (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf)
- Condition Variables (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf)

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, 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, value;
```

Apart from the `Map()` and `Reduce()` functions, there is an option to provide a third user-defined `Combine()` function, if the `Reduce()` function is commutative and associative.

The `Combine()` function does partial merging of the data emitted by a single `Mapper()`, before it is sent to the `Reduce()` function. More specifically, a `Combine()` function is executed as many times as the number of unique keys that its respective `Map()` function will produce.

Typically the functionality of `Combine()` and `Reduce()` functions can be very similar. The main difference between a `Combine()` and a `Reduce()` function, is that the former merges data from a single `Map()` function before it is forwarded to a reducer, while the latter from multiple mappers.

We can extend the previous example, by adding a `Combine()` function, as follows:

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

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

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, value;
```

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, 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()`, `Combine()` 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 *(*CombineGetter)(char *key);
typedef char *(*ReduceGetter)(char *key, int partition_number);
typedef char *(*ReduceStateGetter)(char* key, int partition_number);

typedef void (*Mapper)(char *file_name);
typedef void (*Combiner)(char *key, CombineGetter get_next);
// Simple mode: `get_state` is NULL and `get_next` can be called until you get NULL
// Eager mode: `get_state` and `get_next` will only be called once inside the reducer
// More details on the modes are provided later
typedef void (*Reducer)(char *key, ReduceStateGetter get_state,
                        ReduceGetter get_next, int partition_number);
typedef unsigned long (*Partitioner)(char *key, int num_partitions);

// External functions: these are what *you must implement*
void MR_EmitToCombiner(char *key, char *value);
void MR_EmitToReducer(char *key, char *value);
// NOTE: Needs to be implemented ONLY for the Eager mode
void MR_EmitReducerState(char* key, char* state, int partition_number);

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,
        Combiner combine,
        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`), a pointer to a Combine function (type `Combiner`, called `combine`), 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, a Reduce function, and possibly a Combine or Partition function (or both), and then call `MR_Run()`. The infrastructure will then create threads as appropriate and run the computation. If we do not wish to use a `Combine()` function, then `NULL` is passed instead as the value of `combine` argument. Note that your code should work in both cases (i.e. use the `Combine` function if a valid function pointer is passed, or entirely skip the combine step if `NULL` is passed).

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 combiners and from combiners 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_EmitToCombiner(token, "1");
        }
    }
    free(line);
    fclose(fp);
}

void Combine(char *key, CombineGetter get_next) {
    int count = 0;
    char *value;
    while ((value = get_next(key)) != NULL) {
        count ++; // Emmited Map values are "1"s
    }
    // Convert integer (count) to string (value)
    value = (char*)malloc(10 * sizeof(char));
    sprintf(value, "%d", count);

    MR_EmitToReducer(key, value);
    free(value);
}

void Reduce(char *key, ReduceStateGetter get_state,
            ReduceGetter get_next, int partition_number) {
    // `get_state` is only being used for "eager mode" (explained later)
    assert(get_state == NULL);

    int count = 0;

    char *value;
    while ((value = get_next(key, partition_number)) != NULL) {
        count += atoi(value);
    }

    // Convert integer (count) to string (value)
```

```
    value = (char*)malloc(10 * sizeof(char));
    sprintf(value, "%d", count);

    printf("%s %s\n", key, value);
    free(value);
}

int main(int argc, char *argv[]) {
    MR_Run(argc, argv, Map, 10,
        Reduce, 10, Combine, 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_EmitToCombiner()` 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_EmitToCombiner()` function is used to propagate data from a mapper to its respective combiner; it takes key/values pairs from a **single** mapper and temporarily stores them so that the the subsequent `Combine()` calls (i.e. one for each unique emitted key from its mapper), can retrieve and merge its values for each key.

The `Combine()` function is invoked for each unique key produced by the respective invocation of `Map()` function. It first goes through all the values for a specific key using the provided `get_next` function, which returns a pointer to the value passed by `MR_EmitToCombiner()` or `NULL` if all values have been processed. It then merges these values to produce *exactly one* key/value pair. This pair is emmited to the reducers using the `MR_EmitToReducer()` function.

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

The `Reduce()` function is invoked once per intermediate 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_EmitToReducer()` 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, which is just printed in the 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.

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 (more details below).
- **Memory Management** Another concern is memory management. The `MR_Emit*()` functions are 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 processing engine. Then, when the entire mapping and reduction is complete, it is the responsibility of the MR library to free everything.
- **API** You are allowed to 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.

Partitioning

Assuming that a valid combine function is passed, there are two places where it is necessary to store intermediate results in order to allow different functions to access them: (a) store mappers output to be accessed by combiners, and (b) store combiners output to be accessed by reducers. You should think which data structure is appropriate for each scenario, and what properties you need to guarantee in your implementation. For example, can you re-use the same data structure for both cases? Should both data structures support concurrent access? It is essential to have clearly answered the above questions before you start writing any code.

Depending on whether you are working alone or in a group, there are different requirements regarding the functionality of reducers:

- **One person group**: You should launch the reducer threads once all the combiners have finished (i.e. **simple mode**). Waiting for all combiners (and thus mappers) to finish means that the mappers output data would be available in your intermediate data structure. Then, each reducer thread should start invoking `Reduce()`

functions, where each one will process the data for a single intermediate key. In this case the `get_state` (of type `ReduceStateGetter`) should always be `NULL`. For the simple mode, you *do not* need to implement the `MR_EmitReducerState()` function.

- **Two person group**: You should implement additional functionality (i.e. **eager mode**) that will allow reducers to process output from combiners "on-the-fly". This means that mapper and reduce threads should be running simultaneously, and thus be launched from the beginning. Recall that each reducer thread is responsible for a set of keys that are assigned to it (i.e. partition number). Therefore, each reducer thread should wait until data becomes available (for any key that belongs into that set) in the intermediate data structure. This data is "pushed" by the combiners using the `MR_EmitToReducer()` function. When that occurs, the reducer thread must (wake up if it sleeps and) remove the newly arrived data from the data structure. Then, it should *immediately* process it by running the appropriate `Reduce()` function. Finally, it will get the merged partial result from the `ReduceStateGetter` and store the updated partial result using the `MR_EmitReduceState()`. If no data is available at a certain point, it should go to sleep again, in order to yield resources to the mappers. Note that it is entirely possible to run the `Reduce()` multiple times for the same key, as data from different mappers will keep being inserted into the intermediate data structure. When the mappers (or combiners) finish, and there are no more values to merge, the `get_next()` should return `NULL`. This can be used to "notify" the reducer that the final result is ready, and therefore can be printed.

Below is an example of how the word count `Reduce()` function might look like specifically for the eager mode:

```
void Reduce(char *key, ReduceStateGetter get_state,
            ReduceGetter get_next, int partition_number) {
    char *state = get_state(key, partition_number);
    int count = (state != NULL) ? atoi(state) : 0;

    char *value = get_next(key, partition_number);
    if (value != NULL) {
        count += atoi(value);

        // Convert integer (count) to string (value)
        value = (char*)malloc(10 * sizeof(char));
        sprintf(value, "%d", count);

        MR_EmitReducerState(key, value, partition_number);
        free(value);
    }
    else {
        printf("%s %d\n", key, count);
    }
}
```

At first, `get_state` is called to retrieve the state (result) of potential previous `Reduce` invocations. If no state for the specified key exists, `get_state` should return `NULL`. Next, the `get_next` function retrieves the newly arrived data from a mapper. Then, the two values are merged, and finally the new state (result) is emitted (call to `MR_EmitReducerState`). Once the last value is received (i.e. get_next returns NULL), the Reducer prints the final count using printf.

Testing

We will soon provide few test applications to test the correctness of your implementation. We will post on Piazza when they become available.

Grading

Your code should turn in `mapreduce.c` which implements the following functions correctly and efficiently: `MR_EmitToCombiner()`, `MR_EmitToReducer()`, `MR_EmitReducerState()` (if working in groups) and `MR_Run()`. You should also have implemented two versions of `get_next()` function (one for the `Combiner()` and one for the `Reducer()`), as well as a single version of `get_state()` function.

Your program will be compiled using the following command, where test.c is a test program that contains a main function and the Mapper/Reducer (and potentially the Combiner) as shown in the wordcount example above:

```
gcc -o mapreduce test.c mapreduce.c mapreduce.h -Wall -Werror -pthread -O
```

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 to see if it runs within suggested time limits.

Submitting Your Implementation

Please read the following information carefully. **We use automated grading scripts, so please adhere to these instructions and formats if you want your project to be graded correctly.**

If you choose to work in pairs, only **one** of you needs to submit the source code. But, **both** of you need to submit an additional **`partner.login`** file, which contains **one** line that has the **CS login of your partner** (and nothing else). If there is no such file, we are assuming you are working alone. If both of you turn in your codes, we will just randomly choose one to grade.

To submit your solution, copy your files into `~cs537-1/handin/<cs-login>/p4a/`. One way to do this is to navigate to your solution's working directory and execute the following command:

```
cp -r . ~cs537-1/handin/$USER/p4a/
```

Consider the following when you submit your project:

- If you use any slip days you have to create a `SLIP_DAYS` file in the `/<cs-login>/p4a/` directory otherwise we use the submission on the due date.
- Your files should be directly copied to `~cs537-1/handin/<cs-login>/p4a/` directory. Having subdirectories in `handin/<cs-login>/p4a/` 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.

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.