Programming Assignment 1 - MapReduce

- The intermediate submission is due **Mon, Feb 16 2026 at 11:59 PM**. This is outlined more in the Submission section of this document.
- The final submission of this project is due **Wed, Feb 25, 2026 at 11:59 PM**
- You can submit up to one day late, with a 10% deduction. This due date is **Thurs, Feb 26, 2026 at 11:59 PM**
- The project can be completed in groups up to size 3. While you *can* work as an individual, it is recommended to at least work with a partner

Introduction

In the world of distributed computing, some tasks are too large to run on one single machine. For example, large web services (AWS, Google, etc.) all collect data about every request that is sent to their servers, which they occasionally need to extract information from (user behaviors, faulty routes, etc.). When you serve millions of daily users, the task of examining their behavior is no small feat. Google generates somewhere in the ballpark of 20 petabytes (20 million gigabytes) of log files every day. To extract information from these on one machine would take far too long. Issues with jobs that require huge compute have existed for ages, and Google scientists in 2004 published a paper titled "MapReduce: Simplified Data Processing on Large Clusters", which introduced a new method for massive data processing on a distributed cluster of machines, titled MapReduce.

In this project, you will be implementing a full MapReduce pipeline. First, the initial process, which we will call the "main" counts the number of files to read and distributes the workload equally among a set of mappers. These mappers then read the files and apply their "map" translation, which essentially reads their assigned data and transforms it into some intermediate space, with aggregation by similar values. The main process then waits for all mappers, and then dispatches a specified number of "reducers" to take the result of the map stage, split the key space, and reduce the data down further. If your key domain is IP addresses, as we will work with in this project, an example of "splitting the key space" would be to assign reducers to read data based on the first byte of the IP. So, one could be dispatched to read 0 through 85, one 85 through 170, and the final one 170 through 255. Essentially, a MapReduce is used to ingest huge amounts of data and reduce it down into a manageable size.

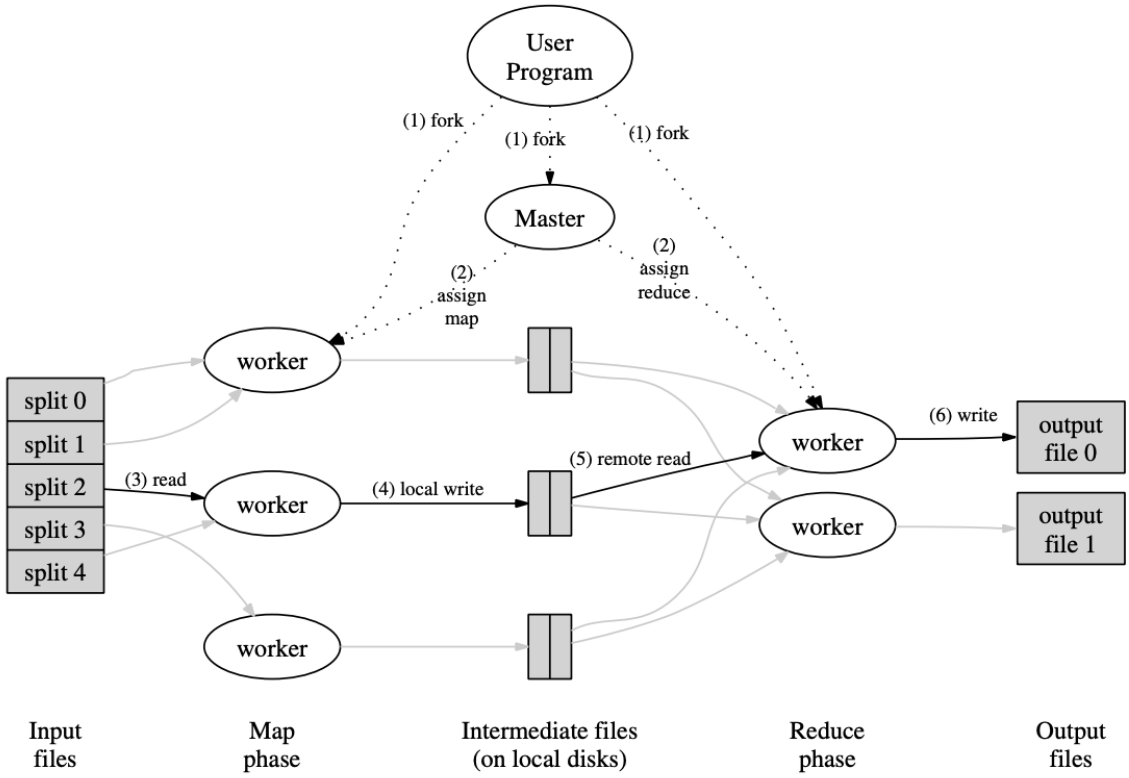

Figure 1: A diagram of MapReduce architecture. From "MapReduce: Simplified Data Processing on Large Clusters" by Jeffrey Dean and Sanjay Ghemawat

Implementation

Your solution should edit the following files:

- `main.c`
- `map.c`
- `table.c`
- `reduce.c`

There is no need to edit or create any external files to create a satisfactory submission of this assignment. You are allowed to add extra include statements if you find external functions you want to use.

Points to Remember

- In a main function, `argc` tells you how many arguments were passed in, and `argv` holds them as an array of strings
- Print out an error message with `perror` after any failed system call
- When any function fails in a main, you can either `return` or call the `exit` function with some non-zero status code to signify failure
- In any case of failure, make sure you free any resources you have allocated, and any files opened
- Remember that data allocated by `malloc` is *not* guaranteed to be all 0s
- If you use the `atoi` function to convert a string to an integer, remember that on failure it returns a 0, which can be a valid byte of an IP. If you think any conversions from int to string may be failing, check

if you have an overwhelming number of 0s being returned.

Assumptions

Throughout this project, you are safe to assume that:

- Your MapReduce program will not be called with more reducers than mappers
- Neither the map nor reduce stages will run with more processes than files
- Every entry in the folders that mapreduce has to read will be files, with no nested directories

Main

The main function for this project serves as the controller for every child processes that is spawned in the MapReduce process. Your main should take these steps:

1. Accept input arguments for: <log directory> <number of mappers> <number of reducers>
    - If the user passes a number of arguments less than necessary, print the following error and return from the function:

```
Usage: mapreduce <directory> <n mappers> <n reducers>
```

    - If the number of mappers or reducers is less than one, print the following error and return from the function:

```
mapreduce: cannot have less than one mapper or reducer
```

    - If the directory is invalid, use the perror function to print "opendir", and perror will automatically add the reason for the failed syscall.
    - These error messages above are necessary to pass the testcases
2. Use the `opendir`, `readdir` functions to read the files in the requested directory, and keep track of the filenames to distribute evenly among child processes
    - If you plan to iterate through the directory twice, use the `rewinddir` function before your second loop. Otherwise, your directory stream pointer will be at the end, resulting in no files read on the second iteration
    - Reading directories also includes the `.` and `..` files. Make sure you skip over these, so you don't assign processes to read them
    - View documentation of `struct dirent` for how to get the name of a file from a directory entry
3. Dispatch the requested number of child processes to run the `./map` executable, and wait for them to complete only after they have all been forked.
    - The files that these map processes write to must be named: `0.tbl, 1.tbl, ... n-1.tbl` for n mappers. They must be written to the `./intermediate` directory. Do not worry about creating the directory automatically, just make sure that you have an `./intermediate` directory present when running mapreduce. See the file structure below for an example with n mappers

```
pa-1/
├── test_cases/
├── intermediate/
│   ├── 0.tbl
│   ├── 1.tbl
│   ...
│   └── n-1.tbl
└── ...
```

- If any child exits with a nonzero status, print an error message and return from the main, you can use the `WEXITSTATUS` macro to extract this information. Make sure you wait on all map children before spawning reduce children
- Remember that when passing arguments into exec, you should include NULL as the last argument

4. Dispatch the requested number of reducer processes to run the `./reduce` executable, and wait for them to complete after forking them all.
   - Each child should be given an even portion of the key space to examine, unless it is not evenly divisible, in which case one process will need to take the remainder
   - They should all be instructed to read from the `./intermediate` directory
   - The files should be written in the same style as above, but to the `./out` directory
   - The same rule with the child exit status applies as well
5. Read all of the files in the `./out` directory and print the resulting tables. You should not need to combine them, since they each examine a unique subset of the key space.

Hash Tables

Much of the work of this project is to be done with hash tables, as they are an extremely efficient data structure for aggregating data from multiple sources with a finite set of keys. The `./include/table.h` file defines several functions and their behaviors necessary to implement for a working hash table.

All of the information necessary for this portion of the assignment is outlined in the `./include/table.h` file above each function.

Map

In the map portion of this assignment, processes must:

1. Receive arguments for an output file, as well as a variable number of input files
   - If these requirements are not met, print the following error message:

     ```
     Usage: map <outfile> <infiles...>
     ```

2. For each requested file, read every log line and increment the request count of the given ip in the hash table
   - For example, you read the line:

```
2026-01-26 18:32:59,171.12.54.177,POST,/unsub,200
```

You should lookup the ip `171.12.54.177` in the hash table. If it doesn't exist, add a new bucket for it. Otherwise, increment the request count.

- You should read the lines of the file one by one with the `fgets` function. This approach is better than reading the entire file at once, since log files in practice *could* be too large to fit into memory
- For parsing lines into `log_line_t` structs, consider using the `sscanf` function, with the format string `"%19[^,],%15[^,],%7[^,],%36[^,],%3s"`, which allows you to read fields of the CSV file directory into fixed-length variables
- Another approach would be to use the `strtok` function, which produces similar output when used properly as `split` methods in higher level languages

3. Write the resulting table to the `outfile` file. This should be something like `./intermediate/{mapper_num}.tbl`
4. Return a status of 0 on success, 1 on failure

Reduce

The reduce portion of this assignment is different from map as it splits the resulting key space, the steps are as follows:

1. Receive arguments for a directory to read, a file to write to, as well of the range of their start and end IPs to read.

    - If these requirements are not met, print the following error message:

    ```
    Usage: reduce <read dir> <out file> <start ip> <end ip>
    ```

    - For the IP range, the starting ip is *inclusive*, and the end is *exclusive*. So for example, a range like [0, 256) would cover all possible IPs
    - To check if an argument is not a digit, you can use the `isdigit` function, which operates on individual characters of a string. You can also use `strtol` to convert strings to numbers with more robust error handling than `atoi`, although it has a more complex API
    - If either of the IP arguments are not valid, return the following error message:

    ```
    reduce: invalid IP range
    ```

2. Read every file in the read dir into a hash table. Maintain a master table with all IPs the process is to read, and

3. Iterate through it and aggregate together in another table all IPs that are within the specified key range

4. Write the resulting table to the `outfile` file. This should be something like `./out/{reducer_num}.tbl`

5. Return a status of 0 on success, 1 on failure

Testing

To run all of the tests in this project, run the `make test` command in the root directory. You should see output in your terminal describing your performance. If you see lots of "TIMED OUT," and you're not running an Ubuntu operating system, refer to the Docker development section later on in this writeup.

If you prefer to run a specific test, you can run the `make test testnum=x` command, where x is the test index. You can also run `make test testnum=x-y` to run tests x through y, inclusive.

The tests are meant to be passed in order, so some tests may depend that your MapReduce implementation can successfully pass previous tests. For example, some reduce tests may utilize map to generate testing data.

If you want to remove all of your test results, run the `make clean-tests` command to remove all the test_results directory (which can also be done manually, if you prefer).

Docker Runtime

The testing framework for this project is build to run in an Ubuntu 22.04 environment. If you are developing on a lab machine, you should have no issue with this. However, those completing this project on their personal machines may find that your tests timeout when running the `make test` command. Therefore, you must work within our docker container.

In order to run your local development environment within a Docker container, you must first install the Docker application. You should download the devcontainer.zip file from the course canvas page, which should contain the following files: `devcontainer.json Dockerfile .clang-format`. These files define how your environment will be constructed when the Docker container is built. To do so, follow these steps:

1. download the `.devcontainer` folder from Canvas, and place it within your csci4061 directory. The downloaded folder should also contain a `.clang-format` file, which should go at the root of the directory. An example file structure can be seen below:

```
csci4061/
├── .devcontainer/
│   ├── devcontainer.json
│   └── Dockerfile
├── labs/
├── projects/
├── .../
└── .clang-format
```

2. download the dev containers extension within VSCode

3. Open your csci4061 folder in VSCode.
4. When prompted to "Reopen in container" in the bottom right of your screen, click to do so. See an example of the prompt below

- If the "Reopen in container" prompt does not appear, enter the VSCode command palette (CTRL + P, >) and execute the "Dev Containers: Open folder in container" command

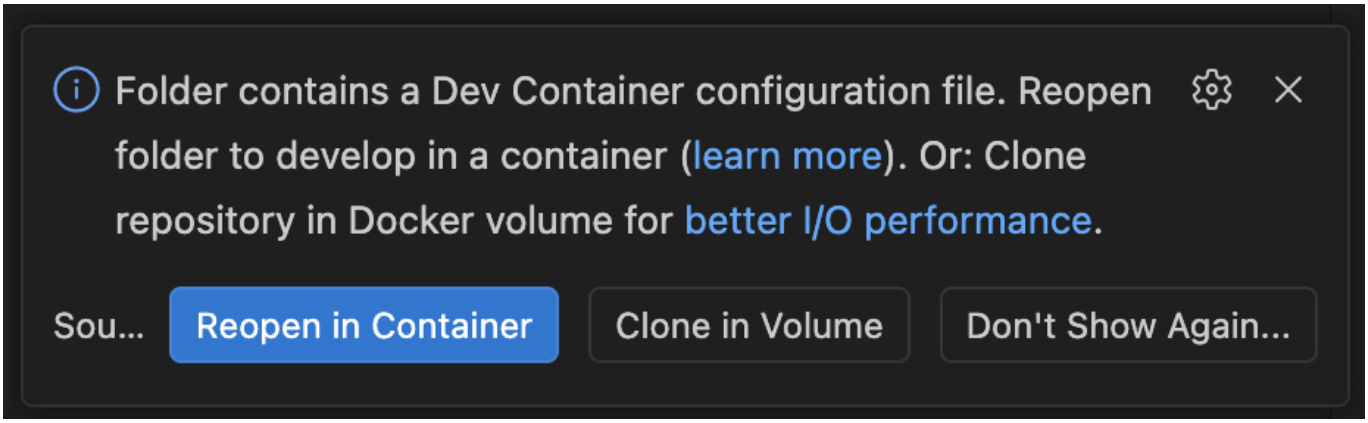

Figure 2: An example window displayed when the user opens a project with a .devcontainer folder in the root

5. Once this setup is complete, you should see a marker in the bottom left of your screen that reads "Dev Container: csci4061-sp26 @ ..."

Submission

To submit this assignment, run the `make zip` command in the root directory of the project. Submit the resulting `pa1-code` file to gradescope. Make sure that the autograder runs completely upon submission. If you have any questions about the status of your submission, contact course staff via Piazza or email.

Include a README.md file in your submission that documents the following:

- A high level design overview of the features in your MapReduce implementation. (Can be the same as your intermediate submission)
- Any assumptions that need to be known about your project, code, etc.
- The portion of work completed by each team member
- Documentation of any AI use (refer to the AI Use section below)

Intermediate Submission

For the intermediate submission, you must submit only your README.md file to gradescope with the high level overview of your system outlined. Your design should *describe your overall MapReduce system*. This includes all actions taken in the main process, the mappers, and the reducers. Your intermediate submission should demonstrate that you have a strong understanding of the MapReduce pipeline, especially how the different stages connect.

We are not requiring an extremely in depth explanation, like diagramming or line by line explanations, but rather a general overview of this MapReduce process, and how it simplifies massive log data.

AI Use

If you are in doubt as to whether you are using AI language models appropriately in this course, I encourage you to discuss your situation with me. Examples of citing AI language models are available at: libguides.umn.edu/chatgpt. You are responsible for fact checking statements and correctness of code composed by AI language models.

If you are ever doubtful about what may or may not be considered academic dishonesty, please do not hesitate to ask the instructor. The consequences of academic dishonesty can be extremely serious, and you must avoid getting into such situations.

The purpose of this assignment is to learn. While AI has its place in generating menial, repetitive, or boilerplate code, it should not be used to complete the entire assignment for you.

Grading Criteria

There are 31 testcases worth 34 points included for you in this project. You can run them by running the `make test` command in the same directory as your `Makefile`. Once you have passed all testcases, you can be very sure that you have produced an accurate solution. There will be more hidden testcases included in the autograder, but none that are built upon material you cannot test on your own machine.

Manual Grading Criteria (60 points)

We will review the `main.c`, `map.c`, `reduce.c`, and `table.c` files by hand. The purpose of this is to ensure that you followed good systems programming practices, and that your implementation contains the solution elements we expect.

We reserve the right to deduct 1/2 point for every C systems call that could cause an error and is not error-checked.

Additionally, we may take off points if your code is unreadable. Generally, *most* code should be able to speak for itself. If you feel as though your code is not self-explanatory through a cursory examination, leave a comment explaining what you are doing.

Intermediate Submission (10 points)

- Intermediate submission meets requirements listed above **10 points**
    - Partial credit will be given if the submission is completed but the explanation is not satisfactory

Submission (10 points)

- The README file is present filled out **7 points**
- All requested files are present with their original names **3 points**
    - Make sure your directory is still named "pa1-code," otherwise you will fail this check and the autograder

Main (25 points)

- Returns an error message on invalid arguments **1 point**
- Assigns even numbers of files for map processes **2 points**
- Assigns even numbers of IP addresses for reduce processes **2 points**
- Uses fork + exec to run map + reduce executables **9 points**
- Main waits for all child processes to finish and checks exit codes **9 points**
- Prints the output of the MapReduce pipeline **1 points**
- Returns proper exit code after execution (0 for success, 1 for failure) **1 point**

Map (5 points)

- Returns an error message on invalid number of arguments **1 point**
- Reads each log file and parses it into a hash table **2 points**
- Writes the hash table to the requested outfile **1 point**
- Returns proper exit code after execution (0 for success, 1 for failure) **1 point**

Reduce (5 points)

- Reads all intermediate files **1 point**
- Merges request counts of only IPs within specified range **2 points**
- Writes the hash table to the requested outfile **1 point**
- Returns proper exit code after execution (0 for success, 1 for failure) **1 point**

Table Functions (5 points)

- All table functions are completed according to the instructions in `table.h`

Written by Peter Olsen, olse0321. Jan 28, 2026