Lab 4: A Simple MapReduce Framework

CMPSC 473, SPRING 2019

Released on April 8, 2019, due on April 26, 2019, 11:59:59pm

Adrien Cosson, Yu-Tse Lin, Devin Pohly, Shruti Mohanty, Mitchel Myers, and Bhuvan Urgaonkar

Purpose and Background

This project is designed to give you experience in writing multi-threaded systems by implementing a simplified "MapReduce" framework. The end product will be a base on which a variety of different parallel computations can be run.

Many commonly occurring data processing tasks can be expressed as "feed-forward sequences" of stages wherein: (i) the first stage s_1 reads inputs from one or more files, (ii) stage s_k ($1 \leq k \leq n$) applies some transformation upon its input to create an intermediate output which serves as the input for s_{k+1}, and (iii) the last stage s_n writes the final output to one or more files. Furthermore, very often individual stages are highly parallelizable (offering the user an effective knob for using available machines/resources for performance improvements). Consider how the following two popular tasks can be programmed to fit this basic structure:

- *wordcount:* The first stage reads individual words and produces key-value pairs of the form `(word, 1)`. The second stage consumes these, groups them by key, and sums up the counts in each group to produce the final output. Notice how the first stage can be parallelized.
- *grep:* The first stage reads individual lines from a file or set of files and matches each line against a given regular expression, producing key-value pairs of the form `(match, 1)`. The second stage, as in wordcount, consumes these and sums the number of occurrences.

Given this staged structure, many researchers/engineers have found it beneficial to implement a common software infrastructure ("framework") for the task-independent portions of this processing (e.g., setting up threads). This allows programmers to focus on coding their task-specific stages and simply re-use the task-independent functionality offered by the framework. MapReduce, developed by Google, is among the most popular examples of such a framework.

This project brings together many different ideas covered in class and will introduce you to several challenges and best practices in systems programming. Most notable among these are as follows.

- You will learn to write multi-threaded code that correctly deals with race conditions.
- You will appreciate the role of layering and interfaces in the design of systems software.
- You will continue to carry out simple performance evaluation experiments to examine the performance impact of (i) the degree of parallelism in one of the stages and (ii) the size of the shared buffers which the two stages of your application will use to communicate.

Description

You will implement a simplified version of the MapReduce framework. Note that you are not writing the *application* (e.g., wordcount); you are writing the *system*, or framework, on which these applications will run. In other words, you will still be using the the C library and UNIX system call APIs to implement the given functions, but unlike previous labs, you will also be *providing* a precisely specified API to allow other software to utilize your system. You are required to ensure that your framework correctly implements this precise interface expected by the application.

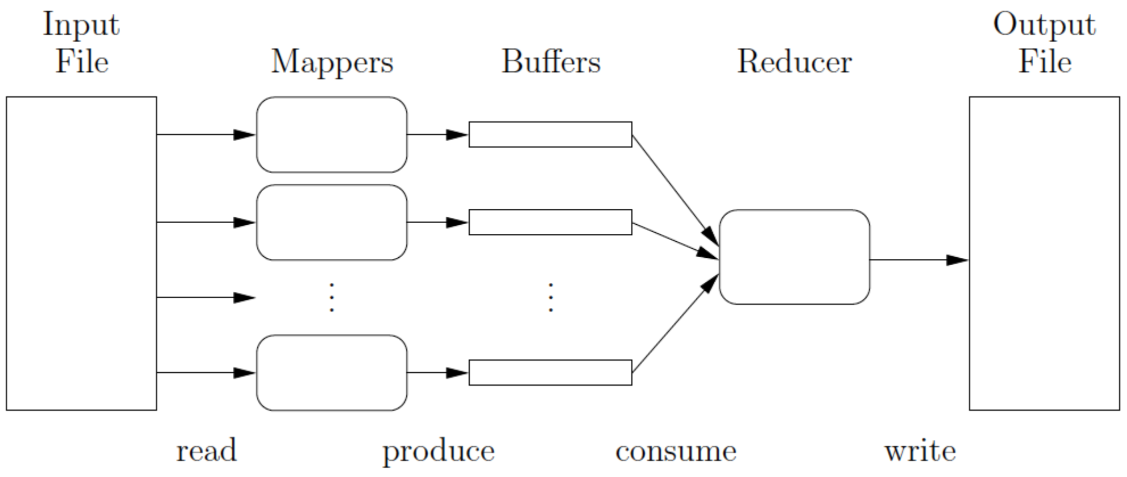

Figure 1: Overview of the simplified MapReduce framework

In the simplified MapReduce framework (illustrated in Figure 1), there are only two stages: Map and Reduce. We will call individual threads making up Map and Reduce mappers and reducers, respectively. The mappers will read from an input file and produce key-value pairs to use as input for the reducer. The single reducer will consume these key-value pairs and write its results to an output file. The application will provide you with functions "Map" and "Reduce" to be executed in each stage, and your code will create a thread for each mapper and one for the reducer and run them concurrently. Each mapper thread will share a separate buffer with the reducer for transferring data between stages, and you will need to be sure to synchronize their access to this shared data.

You will be given a header file defining the API, a source file which contains the functions you must implement, and a set of pre-compiled MapReduce tasks. These tasks will include an exectutable called `mr-wordc`. More executables and tests may be provided as the project progresses.

MapReduce framework API (your code)

Your code will provide an API of six functions which applications can use to perform parallel MapReduce computations. The application will provide its own Map and Reduce code (using callback function pointers), and your framework will use these to run the MapReduce operation. It is up to you to determine how best to implement these functions within the constraints laid out in the assignment. (See the file `mapreduce.h` for more detail on these functions and the related types and structures.)

The first four functions have to do with the overall setup of the framework:

- `mr_create(mapfn, reducefn, threads, buffersize)`: Allocates, initializes, and returns a new instance of your MapReduce framework, which will eventually execute the given Map and Reduce callbacks using the specified number of mapper threads with buffers of the given size. Returns `NULL` on error.
- `mr_destroy(mr)`: Destroys and cleans up an existing instance of your MapReduce framework. Any resources which were acquired or created in `mr_create` should be released or destroyed here.
- `mr_start(mr, inpath, outpath)`: Begins a multi-threaded MapReduce operation using the initialized framework, using the specified files for input and output. If the output file does not exist, it should be created; if it does, it should be truncated and overwritten. Returns 0 if successful and nonzero on error.
- `mr_finish(mr)`: Blocks until the entire MapReduce operation is complete and all threads are finished. Returns 0 if every Map and Reduce callback returned 0 (success), or nonzero if any of them returned nonzero.

You must also implement a shared memory buffer object which stores key-value pairs as they are produced by a mapper until they are consumed by the producer. You will create one of these buffers for each mapper thread, and you will need to synchronize the mapper and reducer to avoid race conditions on the shared data. The size to use for the buffers is passed to `mr_create`, and part of your performance evaluation will be to demonstrate the effects of changing its value.

The mapper and reducer will communicate using a general key-value pair structure called `kvpair`. This structure has pointers called `key` and `value` which point to the data for the key and value, respectively, and integers `keysz` and `valuesz` which give their sizes in bytes. These pointers are opaque to your framework, so you will not need to interpret the data, only move it from one place to another. The functions which you will implement to provide this communication are:

- `mr_produce(mr, id, kv)`: Adds the key-value pair `kv` to the buffer for the given mapper ID. If there is not enough room in the shared buffer, this function should block until there is. If the pair is larger than the entire buffer, then this function should fail. Returns 1 if successful and -1 on error (this convention mirrors that of the standard "write" function).

- `mr_consume(mr, id, kv)`: Retrieves the next key-value pair from the buffer for the given mapper ID and writes it in the locations indicated by `kv`. If no pair is available, this function should block until one is produced or the specified mapper thread returns. Returns 1 if successful, 0 if the mapper returns without producing another pair, and -1 on error (this convention mirrors that of the standard "read" function).

  IMPORTANT: Note that your framework is expected to serialize the data passed to `mr_produce(...)` into the buffer, and deserialize it in `mr_consume(...)` back into a key-value pair. You may not just add the key-value struct to the buffer, since it contains pointers to data; and that data may change after the call to `mr_produce(...)` finishes. It is up to you to decide how to serialize and deserialize the data. One potential solution would be to serialize the size of the key, then the key data, then the size of the value, and then the value data.

  Note: The caller is responsible for allocating memory for the key and value, and will specify the size of the available space in the corresponding size fields. The `mr_consume` function should update the size fields to indicate the actual number of bytes placed in each.

Map and Reduce functions (not your code)

Every application in our MapReduce framework defines its computation in terms of a different Map and Reduce function. The application will provide these two "callback" functions when it initializes the framework, and your code will "call them back" to do the processing work for each stage. They are defined as follows:

- `map(mr, infd, id, nmaps)`: Implements the application's Map function, which will be run in parallel using the number of mapper threads specified in `mr_create`. You will pass this function a pointer to your MapReduce structure, a file descriptor that can be used to read input, a unique mapper ID from 0 to one less than the number of mapper threads, and the total number of mapper threads being used.

- `reduce(mr, outfd, nmaps)`: Implements the application's Reduce function. You will pass this function a pointer to your MapReduce structure, a file descriptor that can be used to write output, and the number of mapper threads being used.

Each of these functions returns 0 to indicate success and nonzero to indicate failure.

Note that your framework and implementation remain exactly the same from application to application, but by changing these two functions it can be applied to a wide variety of parallel tasks.

Performance Evaluation

Your performance evaluation will study the impact of two parameters: (i) the number of mapper threads and (ii) the size of the shared memory buffer via which a mapper communicates with the reducer. For the one of the input files provided by us (details coming in future update), you will report the minimum value of the completion time of `mr-wordc` for the following parameter settings: (i) number of mapper threads $\in \{1, 2, 4, 8, 16, 32, 64\}$ and (ii) size of buffer $\in \{100\,\mathrm{B}, 1000\,\mathrm{B}, 10000\,\mathrm{B}\}$. Write a few sentences in the performance evaluation component of your `README` file on whether or not your measurements match up with your intuition/expectation. You must run each experiment at least 5 times and then obtain the minimum value. **The format of the `README` file will be released shortly.**

You may add code in to your project to produce these statistics. However, you should remove any extraneous print statements, etc., from your final submission.

Procedure

1. Make sure you work on a CSE Lab204 Linux machine.
2. After accepting the invitational link to Lab 4 and creating your Github team for this lab, you will get access to the code we provide. You will find a directory `lab4` containing these files:
   - `mapreduce.h`: Header file defining the API you must provide as well as a `map_reduce` struct type for storing your framework's state. Fill out the struct as needed, but *do not* change anything else in this file!
   - `mapreduce.c`: Framework implementation. This is where you should write all of your code (except the struct definition mentioned above).
   - `bin/`: Folder containing the pre-compiled static object files for the sample programs. The Makefile will link your framework to these files. Contents: `mr-wordc.o`.
   - `input/`: Folder containing input files for word count program.
   - `output_compare/`: Folder containing expected outputs for the word count program.
   - `Makefile`: Makefile for the entire project; you will not need to modify this.
   - `test.sh`: Runs the word count program on the available inputs, and checks if the produced output is correct.
   - `README`: Text file containing group member names, performance evaluation results, and list of sources consulted. **This will be released in a future update with information on it's specific format.**
3. Run `make` to build the project. The starter code should build with no changes, and you should make sure that your code continues to compile successfully at all times.

4. Remember to comment your code *as you write it* to explain how everything works.

5. Implement the six functions in the `mapreduce.c` file according to the API specification. You may create helper functions in `mapreduce.c` as needed; these should be declared `static` and not included in the header file. Use the tools described in the next section to help with development.

6. Carry out the performance evaluation of your completed implementation and document your results in the `README` file.

Testing

The provided Makefile will compile your framework into an object file and link it with the word count program. As in real-world systems programming, you do not have the source code for the applications that will be using your system.

There is currently one testing application included with the project:

- `mr-wordc`: MapReduce version of word count, where Map calculates a mini-wordcount and Reduce merges the lists and generates the output.

Additionally, the script `test.sh` is provided to test the word count program with your implementation of the MapReduce framework. Running `./test.sh` will compile the `mr-wordc` against your `map-reduce.c` and run it on all available inputs. Additionally, it will compare the output of the word count program to the provided expected output. The results will be available in `output/mr-wordc/` directory by input file name. Any output produced by your framework is available in `output/mr-wordc/out/` by input file name. If your output does not match, you can find a diff with the expected output at `output/mr-wordc/diffs/` by input file name.

Additional tests and applications will be provided as the lab progresses, distributed as updates to the `base` repository. An email will be sent over canvas to inform you of each such update.

Submission

1. Ensure that you have the correct version of all of your files and that your `README` file is complete.

2. Add, commit, and push all your changes to the `master` branch of your repository before the deadline. **Do not upload an archive of your project to your repository.**

Grading

The performance of your code on the provided sample inputs and other withheld inputs will account for approximately 80% of your grade. The remaining 20% will be based on your report. Additional details regarding grading and the report will be released shortly.

Important reminder

This project is significantly more complex than the previous projects. To finish it successfully, you must start early and be proactive about seeking clarifications from the instructors. While most of the concepts relevant to this project have been covered in class, there are some that may be new to you and not defined here, e.g., callback functions and key-value pairs.

As with all projects in this class, you are not permitted to:

- Use code from outside sources without citing (Internet, previous class projects, etc.)
- Share code with other teams or post parts of your code publicly where others could find and use them (e.g., Piazza). You are responsible for protecting your work from falling into the hands of others.
- Allow anyone outside your team to write any part of the code for this assignment.

If there is any question at all about what is acceptable, ask the instructor. **Above all, be honest. This always goes a long way.**

We will run a code comparison tool on all submitted code to determine if there has been any sharing between teams. Any teams found out to be sharing code will receive a 0 for the lab.