Lab 4: A Simple MapReduce-Style Wordcount Application

CMPSC 473, SPRING 2021

Released on April 15, 2021, due on April 29, 2021, 11:59:59pm

Raj Kumar Pandey and Bhuvan Urgaonkar

1 Purpose and Background

This project is designed to give you experience in writing multi-threaded programs by implementing a simplified MapReduce-style *wordcount* application. By working on this project:

- You will learn to write multi-threaded code that correctly deals with race conditions.
- You will carry out a simple performance evaluation to examine the performance impact of (i) the degree of parallelism in the mapper stage and (ii) the size of the shared buffer which the two stages of your application will use to communicate.

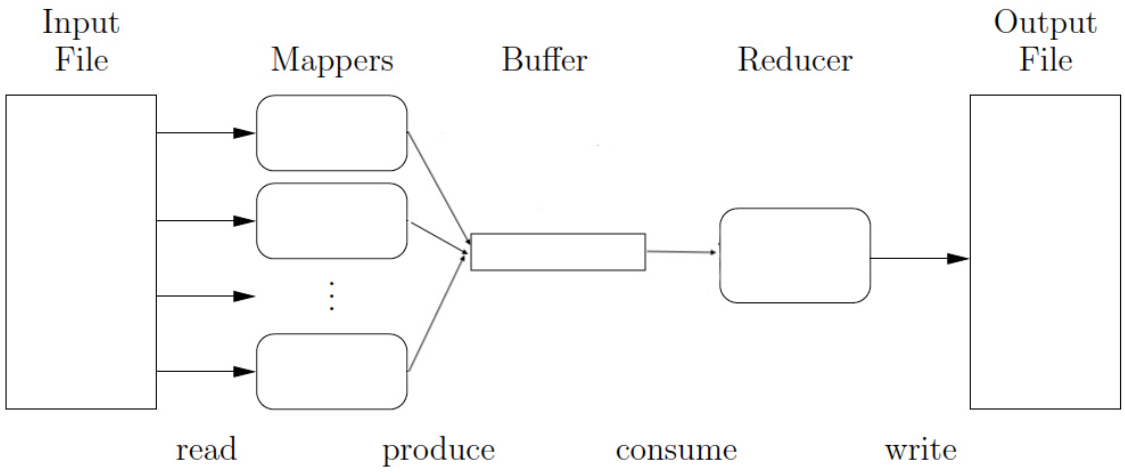

Figure 1: Overview of our Mapreduce-style multi-threaded wordcount application.

As covered briefly in Lecture 22, the wordcount application takes as input a text file and produces as output the counts for all uniquely occurring words in the input file arranged in an alphabetically increasing order. We will assume that the words within our input files will only contain letters of the English alphabet and the digits 0-9 (i.e., no punctuation marks or other special characters). Our wordcount will consist of two stages. The

first stage, called "mapper," reads individual words and produces key-value pairs of the form `(word, 1)`. The second stage, called the "reducer" 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. That is, the mapper stage can consist of multiple mapper threads, each working on its own separate portion of the input file. The reducer stage only contains a single thread which runs concurrently with the mapper threads. The communication between the mapper threads and the reducer thread occurs via a shared in-memory FIFO buffer. Figure 1 summarizes these ideas.

2 Starter code: unsynchronized word-count

We are providing you starter code that implements a preliminary version of wordcount that works correctly in the absence of multi-threading. To appreciate this, you should confirm that this code is able to pass the "serialize" test in which accesses to the buffer occur in a strictly serial manner (i.e., an operation is issued only upon the completion of the previous operation). However, our implementation is *not* thread-safe. That is, it has race conditions. You will need to enhance this code to create a thread-safe version of wordcount.

3 What you need to implement

Take the time to skim all the starter source code files. All your implementation will be confined to the files `helper.c and helper.h`. Specifically, you need to consider enhancing the following functions (recall that the provided versions of these functions work correctly in the absence of multi-threding). Though we have implemented the `buffer_create` function for you, you can enhance it to make any kind of initialization.

- `state_t* buffer_create(int capacity)`: creates an in-memory buffer with the specified capacity in bytes). Returns a pointer to `state_t`, see definition in `helper.h`. This function will be called once at the beginning, you can do any kind of initialization in it based on your implementation.

- `enum buffer_status buffer_send(state_t *buffer, void* data)`: writes `data` to the buffer. In case the buffer is full, the function waits till the buffer has space to write the new data. Returns `BUFFER_SUCCESS` for successfully writing data to the buffer, `CLOSED_ERROR` if the buffer is closed, and `BUFFER_ERROR` on encountering any other error. The size of data can be found out using the function `get_msg_size(data)`.

- `enum buffer_status buffer_receive(state_t* buffer, void** data)`: Reads data from the given buffer and stores it in the function's input parameter, data (Note that it is a double pointer). This is a blocking call i.e., the function only returns on a successful completion of receive In case the buffer is empty, the function waits till the buffer has some data to read.

- `enum buffer_status buffer_close(state_t* buffer)`: closes the buffer and informs (you may think of giving signal) all the blocking send/receive calls to return with `CLOSED_ERROR`. Once the buffer is closed, send/receive operations will return `CLOSED_ERROR`. Returns `BUFFER_SUCCESS` if close is successful, `CLOSED_ERROR` if the buffer is already closed, and `BUFFER_ERROR` for other errors.

- `enum buffer_status buffer_destroy(state_t* buffer)`
  Frees all the memory allocated to the buffer , using own version of sem flags The caller is responsible for calling `buffer_close` and waiting for all threads to finish their tasks before calling `buffer_destroy` Returns `BUFFER_SUCCESS` if destroy is successful, `DESTROY_ERROR` if `buffer_destroy` is called on an open buffer, and `BUFFER_ERROR` in any other error case

4 Programming rules

You are not allowed to take any of the following approaches to complete the assignment:

- Spinning in a polling loop to implement blocking calls.
- Sleeping instead of using condition-wait.
- Trying to change the timing of your code to overcome race conditions.
- Using global variables.

You are only allowed to use the pthreads library, the standard C library functions (e.g., malloc/free), and the provided starter code. If you think you have a valid reason to use some code outside of these, please contact the course staff to determine its eligibility.

5 Testing your code

Your code will be evaluated for correctness, properly handling synchronization, and ensuring it does not violate any of the programming rules from the previous section. We are providing several test cases which are located in `test.py`—you can find the list of tests at the bottom of the file. Although we are providing you all the tests that are part of our default plan for grading your work, we reserve the right to employ additional tests during the final grading if the need arises. Therefore, you are responsible for ensuring your code is correct, where a large part of correctness is ensuring you don't have race conditions, deadlocks, or other synchronization bugs.

Running the `make` command in your project will create two executable files: `wordcount` and `wordcount_sanitize`.

- The default executable is `wordcount`. If you want to run a single test, run the following: `./wordcount [test_case_name] [iters]`, where `[test_case_name]` is the test name and `[iters]` is the number of times to run the test. If you do not provide a test name, all tests will be run. The default number of iterations is 1.

- The other executable, `wordcount_sanitize`, will be used to help detect data races in your code. It can be used with any of the test cases by replacing `./wordcount` with `./wordcount_sanitize`. Any detected data races will be output to the terminal. You should implement code that does not generate any errors or warnings from the data race detector.
- Valgrind is being used to check for memory leaks, report uses of uninitialized values, and detect memory errors such as freeing a memory space more than once. To run a valgrind check by yourself, use the command: `valgrind -v --leak-check=full ./wordcount [test_case_name] [iters]`. Note that `wordcount_sanitize` should not be run with valgrind as the tools do not behave well together. Only the `wordcount` executable should be used with valgrind. Valgrind will issue messages about memory errors and leaks that it detects for you to fix them. You should implement code that does not generate any valgrind errors or warnings.

To run all the supplied test cases, simply run the following command in the project folder: `./test.py`. This runs all of the provided tests to autograde your assignment.

6 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 the mapper threads communicate with the reducer thread. For one of the input files provided by us, you will report how execution time varies with (i) and (ii) in the ranges described below.

For your performance evaluation, you will be using `wordcount` in a "custom" mode (wherein the `custom_eval()` function is used). You may additionally use this mode for any custom input files you may want to test your code with. Here is the usage of `wordcount` in this custom mode: `./wordcount custom_eval #mapperthreads buffersize infile outfile`. For example if you want to run `wordcount` with 2 mapper threads, a buffer size of 100B, input file `perf_eval.txt` and output file `perf_out.txt`, you will execute: `./wordcount custom_eval 2 100 perf_eval.txt perf_out.txt`.

To measure the execution time, simply use the `time` command. For example: `time ./wordcount custom_eval 2 100 perf_eval.txt perf_out.txt`. You will conduct your performance evaluation on a W204 machine on the input file named `perf_eval.txt` that we are providing you. You are to vary the parameters (i) and (ii) as follows:

- number of mapper threads $\in \{1, 2, 4, 8, 16, 32\}$ with size of buffer $1000\,\mathrm{B}$,
- size of buffer $\in \{100\,\mathrm{B}, 1000\,\mathrm{B}, 10000\,\mathrm{B}\}$ with 8 mapper threads.

You will write your results in a `README` file. The format of the `README` file as follows. The `README` file appears in the root of the repository. You should begin by filling in the name and email of each member of your group on the specified lines at the beginning of the `README`. The `README` has several sections for you to fill in. Each section has some text describing what you should write in that section. You should replace that text with your

own response for that section. For reporting your experimental results, you will find 9 lines starting with `'@Test:'`. Each of these lines has the format:

```
@Test:  <n maps>, <buffer size>, <minimum execution time in us>
```

For example:

```
@Test: 1, 100, <minimum execution time in us>
```

On each of these lines replace the `<minimum execution time in microseconds>` field with the execution time (in microseconds) you found for this choice of parameters. You are to take the minimum of 5 runs for each such number that you report. **Do NOT include units (us). Do NOT edit any other part of this line.** For example, if the minimum time value was 20 for the experiment with 1 mapper and buffer size of 100 B:

```
@Test: 1, 100, 20
```

7 Submission instructions

Similar to the last assignment, we will be using GitHub for managing submissions, and you must show your partial work by periodically adding, committing, and pushing your code to GitHub. This helps us see your code if you ask any questions on Piazza (please include your GitHub username) and also helps deter academic integrity violations.

Additionally, please input the desired commit number that you would like us to grade in the Canvas assignment that will be created closer to the submission deadline. You can get the commit number from GitHub. In your repository, click on the commits link to the left above your files. Find the commit from the appropriate day/time that you want committed. Click on the clipboard icon to copy the commit number. Note that this is a much longer number than the displayed number. Paste your very long commit number and only your commit number in the assignment submission textbox.

8 Grading

Your score for the project will consist of two parts:

- *Part A:* 80% of your grade will depend on the various test cases we are providing.
- *Part B:* The remaining 20% will be based on the results of your performance evaluation from Section 6.

There are 33 test cases (numbered 0-32) worth a total of 100 points. You only need to score 80 points to meet the entire 80% corresponding to Part A. Anything above 80 will be BONUS. So you can score up to 20 bonus points (i.e., a total score of 120/100) if you are able to pass all the test cases.

You will receive zero points if:

- You violate the academic integrity policy (sanctions can be greater than just a 0 for the assignment)

- You break any of the programming rules
- Your code does not compile/build
- Your code crashes the grading script
- You don't show your partial work by periodically adding, committing, and pushing your code to GitHub

9 Hints and resources

- **Make it a habit to ALWAYS** do a `git pull` before starting work on your code. This will help you in both avoiding possible conflicts as well as you will be updated with any possible updates that we may push. We do not expect to make any updates to the source code after our initial release but there is always a chance we may need to. Similalry, there is a chance we may want to add an extra test that we realize is interesting for you to pass. We will always announce any such update on Canvas but your habit of pulling the latest version of the repo will make things easier.
- An excellent tutorial on the pthreads library appears at which you may find useful: https://computing.llnl.gov/tutorials/pthreads/.
- Carefully read the output from the sanitizer and valgrind tools and think about what they're trying to say. Usually, they're printing call stacks to tell you which locations have race conditions, or which locations allocate some memory that was being accessed in the race condition, or which locations allocate some memory that is being leaked, etc. These tools are tremendously useful, which is why we have set them up for you for this assignment.
- While these tools are very useful, they are not perfect. Some race conditions are rare and don't show up all the time. A reasonable approach to debugging these race condition bugs is to try to identify the symptoms of the bug and then read your code to see if you can figure out the sequence of events that caused the bug based on the symptoms.
- Debugging with gdb is a useful way of getting information about what is going on in your code. To compile your code in debug mode (to make it easier to debug with gdb), you can simply run: `make debug`. It is important to realize that when trying to find race conditions, the reproducibility of the race condition often depends on the timing of events. As a result, sometimes, your race condition may only show up in non-debug (i.e., `release`) mode and may disappear when you run it in debug mode. Bugs may sometimes also disappear when running with gdb or if you add print statements. Bugs that only show up some of the time are still bugs, and you should fix these. Do not try to change the timing to hide bugs.

- If your bug only shows up outside of gdb, one useful approach is to look at the core dump (if it crashes). Here's a link to a tutorial on how to get and use core dump files: http://yusufonlinux.blogspot.com/2010/11/debugging-core-using-gdb.html
- If your bug only shows up outside of gdb and causes a deadlock (i.e., seems to hang forever), one useful approach is to attach gdb to the program after the fact. To do this, first run your program. Then in another command prompt terminal run: `ps aux`. This will give you a listing of your running programs. Find the `PID` for your program. Then within gdb (may require sudo) run: `attach PID`. This will give you a gdb debugging session just like if you had started the program with gdb.

10 Important reminder

To finish the project successfully, you must start early and be proactive about seeking clarifications from the instructors. 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.