In 2004, Google released a general framework for processing large data sets on clusters of computers. We recommend you read [this link](http://en.wikipedia.org/wiki/MapReduce) on Wikipedia for a general understanding of MapReduce. Also, [this paper](http://static.googleusercontent.com/media/research.google.com/zh-CN/us/archive/mapreduce-osdi04.pdf) written by Jeffrey Dean and Sanjay Ghemawat gives more detailed information about MapReduce. However, we will explain everything you need to know below.

To demonstrate what MapReduce can do, we’ll start with a small dataset–three lines of text:

Hello

there

class!

The goal of this MapReduce program will be to count the number of occurrences of each letter in the input.

MapReduce is designed to make it easy to process large data sets, spreading the work across many machines. We’ll start by splitting our (not so large) data set into one chunk per line.

|  | **Chunk #1** | **Chunk #2** | **Chunk #3** |
| --- | --- | --- | --- |
| **Input** | “Hello” | “there” | “class!” |

**Map**. Once the data is split into chunks, map() is used to convert the input into (key, value) pairs. In this example, our map() function will create a (key, value) pair for each letter in the input, where the key is the letter and the value is 1.

|  | **Chunk #1** | **Chunk #2** | **Chunk #3** |
| --- | --- | --- | --- |
| **Input** | “Hello” | “there” | “class!” |
| **Output** | (h, 1) | (t, 1) | (c, 1) |
|  | (e, 1) | (h, 1) | (l, 1) |
|  | (l, 1) | (e, 1) | (a, 1) |
|  | (l, 1) | (r, 1) | (s, 1) |
|  | (o, 1) | (e, 1) | (s, 1) |

**Reduce.** Now that the data is organized into (key, value) pairs, the reduce() function is used to combine all the values for each key. In this example, it will “reduce” multiple values by adding up the counts for each letter. Note that only values for the same key are reduced. Each key is reduced independently, which makes it easy to process keys in parallel.

|  | **Chunk #1** | **Chunk #2** | **Chunk #3** |
| --- | --- | --- | --- |
| **Input** | (h, 1) | (t, 1) | (c, 1) |
|  | (e, 1) | (h, 1) | (l, 1) |
|  | (l, 1) | (e, 1) | (a, 1) |
|  | (l, 1) | (r, 1) | (s, 1) |
|  | (o, 1) | (e, 1) | (s, 1) |
| **Output** | (a, 1) |  |  |
|  | (c, 1) |  |  |
|  | (e, 3) |  |  |
|  | (h, 2) |  |  |
|  | (l, 3) |  |  |
|  | (o, 1) |  |  |
|  | (r, 1) |  |  |
|  | (s, 2) |  |  |
|  | (t, 1) |  |  |

MapReduce is useful because many different algorithms can be implemented by plugging in different functions formap() and reduce(). If you want to implement a new algorithm you just need to implement those two functions. The MapReduce framework will take care of all the other aspects of running a large job: splitting the data and CPU time across any number of machines, recovering from machine failures, tracking job progress, etc.

For this MP, you have been tasked with building a simplified version of the MapReduce framework. It will run multiple processes on one machine as independent processing units and use IPC mechanisms to communicate between them. map() and reduce() will be programs that read from standard input and write to standard output. The input data for each mapper program will be lines of text. Key/value pairs will be represented as a line of text with “: “ between the key and the value:

key1: value1

key two: values and keys may contain spaces

key\_3: but they cannot have colons or newlines

For your initial implementation, start with one mapper process and one reducer.

input\_file

|

MAP

|

REDUCER

|

output\_file

Command line:

mr0 <input\_file> <output\_file> <mapper\_executable> <reducer\_executable>

Sample Usage:

% ./mr0 test.in test.out my\_mapper my\_reducer

my\_mapper exited with status 1

my\_reducer exited with status 2

output pairs in test.out: 9

Your program will:

* Pipe the input file into a mapper process.
* Write the output of the reducer process to the output file.
* If either the mapper or the reducer exits with a nonzero exit, print the exit code to stdout.
* Count the number of lines in the output file and print it to stdout.

You won’t implement your own map() or reduce() function–your program will take the names of a map program and a reduce program on the command line and run those.

Close all unused file descriptors!

The mapper and reducer processes won’t exit until they reach the end of their input file. An EOF won’t be triggered on their input file until all processes that have a copy of their input file descriptor close that file descriptor.

For example, if the main process doesn’t close its copy of the write end of the pipe that the reducer is reading from, the reducer will never see an EOF and will never exit. In each child process created with fork() you’ll also need to close all unused file descriptors inherited from the parent process. To aid you in this, we’ve provided a set of functions, declared in common.h, that make it easy to keep track of all the additional file descriptors you create, and to close them all.

void descriptors\_add(int fd);

void descriptors\_closeall();

void descriptors\_destroy();

You don’t have to use these, but they may make the project easier.

You also should consider using the function pipe2() to create your pipes. If you use pipe2(), you can set a flagO\_CLOEXEC which will instruct the system to close both ends of the pipe if a call to exec is made. See the man pages for pipe2() and open() for more information.

Since most of the child processes in this program have their stdin and stdout redirected, you may wish to create a function for that.

“Reference Implementation”

You can implement the equivalent of this program in a Unix shell quite easily:

% my\_mapper < test.in | my\_reducer > test.out

For version 1, you’ll spread the work across multiple instances of the mapper executable.

[...input\_file...]

| | |

MAP1 MAP2 MAP3

\ | /

REDUCE

|

output\_file

The input file will need to be split into chunks, with one chunk for each mapper process. To split the input file, we’ve supplied the tool splitter. Run it without arguments for a brief explanation of how it works. You’ll start up one instance of splitter for each mapper, using a pipe to send stdout of splitter to stdin of the mapper program.

splitter inputfile 3 0

|

| splitter inputfile 3 1

| |

| | splitter inputfile 3 2

| | |

MAP1 MAP2 MAP3

\ | /

REDUCE

|

output\_file

Command line:

mr1 <input\_file> <output\_file> <mapper\_executable> <reducer\_executable> <mapper\_count>

Sample Usage

% ./mr1 test.in test.out my\_mapper my\_reducer 3

my\_mapper 0 exited with status 1

my\_mapper 2 exited with status 2

output pairs in test.out: 9

Your program will:

* Split the input file into parts and pipe the contents into different mapper processes (use splitter).
* Write the output of the reducer process to the output file.
* Print any nonzero exit statuses.
* Count the number of lines in the output file and print it to stdout.

**Remember to close all the unused file descriptors!**

This too can be done in the Unix shell:

% (./splitter inputfile 3 0 | my\_mapper ; \

./splitter inputfile 3 1 | my\_mapper ; \

./splitter inputfile 3 2 | my\_mapper ; ) | my\_reducer > test.out

Files used for grading part 1:

* mr1.c
* common.c
* common.h

Things we will be testing for in Part 1:

* Inputs of varying size
* Different types of mapper and reducer tasks
* Both mapper and and reducer generating accurate output to stdout file descriptor independently
* Splitter being used correctly to generate equally sized input data for each mapper
* All mappers being run in parallel resulting in at least 2x performance speedup for the pi executable

Things that will not be tested for in Part 1:

* Illegal inputs for either the mapper or reducer (Input data in a format other than as described above)
* Input data larger than 1 MB
* Empty inputs
* Greater than 5 mappers

For the final version of your program, you’ll add support for multiple reducers. Each reducer will handle a distinct subset of the keys. In a full implementation of MapReduce, each map process would know how to send data to each reducer process and send each key/value pair to the appropriate place.

In our simplified version of MapReduce, one process will handle the routing of key/value pairs to the correct reducer process. We’ll call it the shuffler.

splitter inputfile 3 0

|

| splitter inputfile 3 1

| |

| | splitter inputfile 3 2

| | |

MAP1 MAP2 MAP3

\ | /

SHUFFLER

/ | | \

R1 R2 R3 R4

\ \ / /

\ \/ /

output\_file

As before, all mapper processes will read from stdin and write to stdout. You should redirect stdout of splitter tostdin of mapper as in Version 1 and send stdout of all mapper to write end of a pipe that is attachted to stdin of the shuffler process, which you will write.

The shuffler will be run with N output filenames as command line parameters, where N is the number of reducer processes. It will read key/value pairs from stdin, hash the key with hashKey() function declared in commmon.hand then use the hash, modulo N, to decide which output file will the key/value pair will be written to. Please note, you**must** put your shuffler implementation in ‘shuffler.c’.

const char \*key = ...

const char \*value = ...

FILE \*outf = output\_files[ hashKey(key) % N ];

fprintf(outf, "%s: %s\n", key, value)

Sample shuffler input:

a: 1

c: 1

e: 3

h: 2

l: 3

o: 1

r: 1

s: 2

t: 1

Sample shuffler output (with 3 output files):

| **File1** | **File2** | **File3** |
| --- | --- | --- |
| c: 1 | e: 3 | a: 1 |
| l: 3 | h: 2 | s: 2 |
| o: 1 | t: 1 |  |
| r: 1 |  |  |

Sample shuffler output (with 4 output files):

| **File1** | **File2** | **File3** | **File4** |
| --- | --- | --- | --- |
| c: 1 | r: 1 | a: 1 | h: 2 |
| o: 1 |  | e: 3 | l: 3 |
| s: 2 |  |  | t: 1 |

mkfifo.

Time for the fun part. You can create pipes that look like normal files (see man page for mkfifo()). Your main program will use mkfifo() to create a pipe file for each reducer. Use ./fifo\_N as the names of the fifo files (so they will be created in the current directory), where N is the index of the reducer that will read from it. Use S\_IRWXUas the mode argument for mkfifo(). Your program will give the names of those fifo files to shuffler on its command line and set up each reducer such that it reads from corresponding fifo file.

mkfifo() will fail if the file exists already. You should try to remove all fifo files before creating any in order to prevent your program from crashing during testing. And at the end of your program, remove all fifo files created. Make sure to check the return value of mkfifo(). Your program should print a helpful error message and exit with a nonzero exit status if it mkfifo() fails.

All the reducers will send their output to one output file. When opening the output file (with open()) be sure to setO\_APPEND (if you do not, the reducers will overwrite each other).

Command line:

mr2 <input\_file> <output\_file> <mapper\_executable> <reducer\_executable> <mapper\_count> <reducer\_count>

Sample Usage

% ./mr2 test.in test.out my\_mapper my\_reducer 3 4

my\_mapper 2 exited with status 1

my\_reducer 3 exited with status 1

output pairs in test.out: 9

Your program will:

* Split the input file into parts and pipe the contents into different mapper processes (use splitter).
* Split the results of the mappers among the reducers using the shuffler.
* Write the output of each of the reducer process to the output file.
* You must use fifo files to communicate between the shuffler and the reducers.
* Print any nonzero exit statuses.
* Count the number of lines in the output file and print it to stdout.

Files used for grading part 2:

* mr2.c
* shuffler.c
* common.c
* common.h

Things we will be testing for in Part 2:

* Input data of varying sizes
* Different types of mapper and reducer tasks being spawned based on the arguments specified
* Both mapper, reducer and shuffler tasks will be tested both independently of one another and run together
* Splitter being used correctly to generate equally sized input data for each mapper
* All mappers and reducers being run in parallel resulting in at least 2x performance speedup for the pi executable
* Ensuring that mkfifo is used correctly and cleaned up
* No memory leaks and memory errors when running application

Things that will not be tested for in Part 2:

* Illegal inputs for either the mapper or reducer (Input data in a format other than as described above)
* Input data larger than 1 MB
* Empty inputs
* Greater than 5 mappers or reducers

Building

This MP has a very complicated Makefile, but, it defines all the normal targets.

make # builds provided code and student code in release mode

make debug # builds provided code and student code in debug mode

# there is no tsan target because threading is not needed for this MP.

If you are curious, you can run make test to build and run all of the tests in the unit\_tests directory. These tests test the provided code, and can serve as some examples of how it all works.

Input Data

To download the example input files (books from [Project Gutenberg](https://www.gutenberg.org/)), use the Makefile:

make data

You should now see data/dracula.txt and data/alice.txt in your mp folder

Running Your Code

We have provided the following mappers:

* mapper\_wordcount
* mapper\_lettercount
* mapper\_asciicount
* mapper\_wordlengths
* mapper\_pi

These each be used anywhere we specify my\_mapper in these docs.

And the following reducers:

* reducer\_sum
* reducer\_pi

These each be used anywhere we specify my\_reducer in these docs.

For example, if you wanted to count the occurrences of each word in Alice in Wonderland, you can run and of the following

./mr0 data/alice.txt test.out ./mapper\_wordcount ./reducer\_sum

With 4 mappers:

./mr1 data/alice.txt test.out ./mapper\_wordcount ./reducer\_sum 4

With 4 mappers and 4 reducers

./mr2 data/alice.txt test.out ./mapper\_wordcount ./reducer\_sum 4 4

Record Setting Pi Code

As well as the simple mapper/reducer pairs, we also have also included some really cool pi computation code (see[this](http://www.karrels.org/pi/) for more info). For instructions on how to use the pi code, see the file pi/README.txt. Note that we don’t currently compile this with an NVIDIA compiler, so you won’t be able to use the CUDA version of this code (which we haven’t tested) unless you fiddle with the Makefile.