MapReduce

MapReduce due **2019-03-27 23:59**
Graded files:

- mapreduce.c

Content

MapReduce
The Lab
Many mappers, one reducer
Building and Running

Learning Objectives

The learning objectives for MapReduce are:

- Interprocess Communication (IPC)
- Pipes
- Files and File Descriptors
- MapReduce
- Jeff Dean

MapReduce

In 2004, Google released a general framework for processing large data sets on clusters of computers. We recommend you read this paper (http://static.googleusercontent.com/media/research.google.com/en/us/archive/mapreduce-osdi04.pdf) by Jeffrey Dean and Sanjay Ghemawat this link (http://en.wikipedia.org/wiki/MapReduce) 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!"

	Chunk #1	Chunk #2	Chunk #3
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 for `map()` 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.

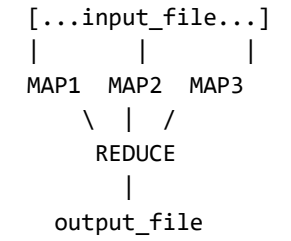
The Lab

For this lab, 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:

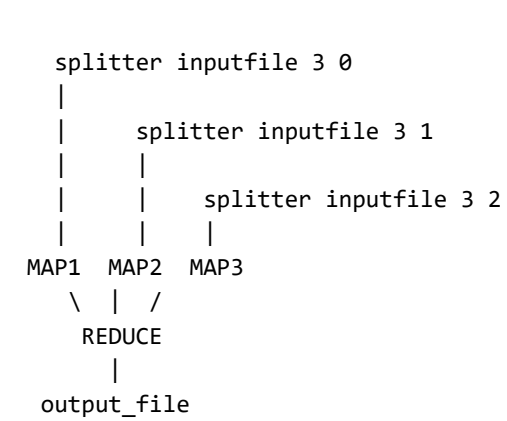
```
key1: value1
key two: values and keys may contain spaces
key_3: but they cannot have colons or newlines
```

Many mappers, one reducer

You'll spread the work across multiple instances of the mapper executable.



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`. **Please** 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 to stdin (https://en.wikipedia.org/wiki/Standard_streams).



Command line:

```
./mapreduce <input_file> <output_file> <mapper_executable> <reducer_executable> <mapper_count>
```

Sample Usage

```
$ ./mapreduce test.in test.out ./my_mapper ./my_reducer 3
```

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.

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
```

Things we will be testing for:

- 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

- **No memory leaks and memory errors** when running the application

In addition, see the comments we've placed in the `main` method for more specific instructions!

Things that will **not** be tested for:

- Illegal inputs for either the mapper or reducer (Input data in a format other than as described above)
- Invalid mapper or reducer code (mappers or reducers that do not work)
- Key Value pairs that are larger than a pipe buffer of 4096 bytes.

Building and Running

Building

This lab 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 lab.
```

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 lab 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

```
./mr1 data/alice.txt test.out ./mapper_wordcount ./reducer_sum 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 for more info). For instructions on how to use the pi code, see the file `pi/README.txt`. Note that we do not currently compile this with an NVIDIA compiler, so you will not be able to use the CUDA version of this code (which we have not tested) unless you fiddle with the `Makefile`.

Restricted Functions

Since a learning objective of this assignment is to use the fork-exec-wait pattern and to learn interprocess communication, if you use `popen` `system` (https://linux.die.net/man/3/popen) automatically fail this MP.