MapReduce Design Document

Authors: Dylan Finkbeiner, Arvind Govindaraja, Kathryn Ricci

(NOTE: See the README for instruction on how to run our tests)

1. Overview of the System

We have implemented a MapReduce system in Java that runs on a single server. Immediately, we should acknowledge that in choosing Java over, for example, C++, we made a tradeoff. Java's memory management and garbage collection introduce an overhead that could have been avoided via explicit memory management in a language like C++. But we felt that the tradeoff was worth it for a conceptually simpler codebase and smoother development with a much lower risk of subtle bugs like memory leaks.

Users extend abstract Mapper and Reducer classes from our library, and implement the abstract methods map() and reduce(), respectively. The user writes a configuration file with three lines containing the number N of workers they want for their map reduce job, as well as the name of the input file to be processed and the directory where program outputs should be deposited. In their application, they create a MapReduceSpecification object which parses this configuration file, and then they instantiate a new MapReduce object (essentially, a map reduce job) using this specification and their implemented Mapper and Reducer classes. Then they execute a job by calling execute() on the MapReduce object.

The process which is executing the execute() function of the MapReduce object can be thought of as the master process. It begins by spawning N new processes for the mapper workers. It communicates with the mapper workers via sockets. When a mapper worker has finished its work, the worker sends the master the paths to the N files where it has written intermediate key-value pairs. When the master receives the file paths, it marks the mapper's work as completed and ceases communication with the worker process.

When the master has acknowledged that all N mapper workers are finished, it then spawns N new processes for the reducer workers. Over sockets it sends each worker their associated list of N intermediate file paths, one from each mapper. A reducer worker merges its assigned files and sorts the merged file. It invokes the user-defined reduce function once for each key observed in the sorted file, then uses a socket to tell the master it has finished. The master then ceases communication with the worker. Once the master has acknowledged that all N reducer workers are finished, the execute() function terminates.

2. Master details

The master process utilizes the ProcessBuilder and Process classes from Java's standard libraries to spawn and manage worker processes. When a user compiles their implementations of the Mapper or Reducer class, those classes will contain a main function from our library. A worker process is essentially spawned as follows: using the Reflection library, we invoke the main function on the user's implementation of the Mapper or Reducer class, and ProcessBuilder executes it in a new Java virtual machine. When spawning a worker, the master passes command-line arguments which specify the details of the work that the worker is responsible for. For each worker spawned, the master listens for a socket connection with the new worker process. It then creates a runnable ClientHandler object to handle communication

over sockets with the worker process. This ClientHandler is run on its own thread on the master. The handlers are added to and executed within a thread pool created using Java's Executors library. There is a tradeoff here, due to the extra overhead required to manage the different threads for the handlers. However, there is a high degree of natural concurrency to exploit in communication with the workers, as the different ClientHandlers are not writing to any shared memory. Thus our program can be faster and conceptually simpler with multithreading here.

Each handler is aware of the "worker number" of the worker it is communicating with. It periodically writes "ping" out to the worker via the socket, and then attempts to read a response from the socket's input stream. The worker will be sending "ping"s back until it finishes its work, at which point it sends the handler a specific string which the handler knows marks completion.

3. Mapper Worker details

Once a mapper worker is spawned by the master, it immediately attempts to open a socket with which to communicate with the master process. After establishing a connection, the worker creates a runnable ServerConnection object and begins executing it on a new thread. The ServerConnection responds to pings from the master while the main thread of the mapper worker carries out the mapping work. There is overhead created by multithreading here, but the tradeoff is well worth it for the sake of conceptual simplicity in communication with the master.

The worker parses several command-line arguments it received from the master. Among these are the worker number and the information contained in the original configuration file for the job. It uses the Reflection library to create an instance of the user's implementation of our Mapper class. Using the worker number, the total number N of workers, and the input file size, the worker computes integers 'start' and 'end' indicating a preliminary range of bytes of the input file that it will work on. Then it adjusts these integers. It finds the first end-of-line character on or after the preliminary 'start' byte and sets 'start' to be the first byte after this end-of-line. It moves 'end' to the first byte containing an end-of-line character on or after the preliminary 'end' byte. The worker uses the skip() functionality of Java's BufferedReader class to avoid reading any more of the input file than is strictly necessary.

A worker instantiates one BufferedWriter object for each of the N reducers. The BufferedReader, now starting at the adjusted 'start' point, is wrapped in a MapInput object which knows the number of bytes that the worker should read based on the adjusted 'start' and 'end' points. The worker invokes the user's map() function, passing the new MapInput object, and the user's code treats the object like a standard reader with a readLine() function. Their function will make calls to emit(), passing key-value pairs they want written out. The worker hashes the key to determine which of the N reducers it should be processed by, and then writes the key-value pair to the associated buffer. The buffers are periodically flushed.

Java must, of course, allocate memory for the buffers, which would not be necessary if we wrote each key-value pair directly to disk. On the other hand, by buffering key-value pairs we reduce the total number of I/O operations and effectively amortize the cost of I/O's over many key-value pairs.

Once a mapper worker has finished its work, it interrupts the ServerConnection thread, waits for it to join with the main thread, and then finally sends the intermediate file paths over the socket to the master. It waits to receive acknowledgement from the master that the paths were received, and then finally terminates.

4. Reducer Worker details

Like a mapper worker, a reducer worker begins by creating a socket. It waits to receive the intermediate file paths from the master over the socket, and then sends an acknowledgment back to the master. Then it creates a ServerConnection object to handle replying to pings from the master, which runs it in its own thread on the worker. The worker merges the intermediate files it has been assigned and passes the merged file to an external sorting library from Google to sort it. The library expects one input file for sorting, so the merging process is necessary, but we merge using File Channels which are a highly efficient file content transfer mechanism in Java NIO. So while there is some overhead introduced by the merging, it is kept to a minimum and we accept the tradeoff for the sake of using the external sorting library. The worker always uses an external sort. While in some cases it might be possible to sort the file in memory, and theoretically faster to do so due to fewer I/O operations, using the external sort every time was conceptually simpler.

The reducer worker then begins reading the sorted file sequentially. A ReduceInput object is first created, passing in the file reader. ReduceInput is a wrapper class. Upon instantiation, it attempts to read a line from the reader. If the end of the sorted file has been reached, the ReduceInput object's 'eof' flag will be set and the reducer worker will know its work is complete. Otherwise, the ReduceInput constructor will extract the key and value from the line and set private 'key' and 'currValue' variables. The reducer worker will invoke the user's reduce() function and pass in this ReduceInput object, and the user's code will call the object's nextValue() method, simulating having a list of values for the current key. Once the ReduceInput reads a line with a new key it will back up the input file reader to before this line and set its 'done' flag. The user's code uses this flag to know when a particular ReduceInput is done. Then the reducer worker will create a new ReduceInput object starting where the previous ReduceInput left off, repeating the process for each new key in the sorted file.

By sorting, we are trading off extra computation up front for the ability to process the intermediate key-value pairs sequentially. This way, we exploit spatial locality and can amortize the cost of I/O operations. It also makes processing the intermediate files conceptually simpler.

5. Fault Tolerance

As we have seen, ClientHandlers executing in separate threads on the master process periodically ping worker processes via sockets throughout the periods where workers are busy. Each handler has private AtomicBoolean variables called 'finished' and 'successful', initially set to **False**. As long as a handler is receiving "ping"s back from the workers (via their ServerConnections), these variables remain unchanged. If the handler receives a specific string that marks the worker's successful completion, then it will set 'successful' to **True** and 'finished' to **True** and stop running. But if a certain amount of time elapses without the handler receiving such a string or a "ping" from the worker, it considers the worker to have failed and sets 'finished' to **True** but leaves 'successful' set to **False**, and stops running.

In the main thread of the master, for each of the mapping and reducing phases there is a loop in which ClientHandlers are dequeued from a queue named 'unfinished', and have their 'finished' and 'successful' variables examined. If a handler is finished but not successful, the corresponding worker is restarted in a new process. If a handler is not finished, it is simply

enqueued again. If a handler is finished and successful, it is not enqueued again. There may be some ‘busy waiting’ that this queue design choice introduces which could be avoided by more complex multithreading code, but we chose this tradeoff for the sake of simplicity in understanding the semantics of the program.

A worker process could die before a socket connection is established with the master, and we handle this case by restarting the worker process if the ServerSocket on the master times out listening for a connection from the worker.

The fault tolerance measures introduce barriers where, for example, we wait to ensure an old process is dead, or we wait to spawn worker (i+1) until worker i is successfully spawned, but we are quite willing to make these tradeoffs for straightforward fault tolerance semantics.