15-440 Project #3: Building a MapReduce Facility
Fall 2013

1. Introduction

This project asks you to implement a *MapReduce facility,* similar to *Hadoop*, but with certain design constraints aimed at enabling it to work more efficiently in our computing environment with smaller data sets. You will construct a facility capable of dispatching parallel Mappers and Reducers across multiple hosts, as well as recovering from worker failures. You will document the API for your MapReduce facility, and example programs, also thoroughly documented, to demonstrate its capability

2. Requirements Overview

As detailed in the several sections of this handout, your solution should:

- Operate on any Andrew machines.
- Use configuration files, which are readily human-readable and human-editable, to configure your instance, including identifying the participant node(s), master node(s), port numbers etc.
- The system should be designed to minimize dispatch latency and overhead by reusing resources, for example by maintaining processes at all participants, rather than by launching them anew for each job. The system should also honor work conservation; if resources should be kept busy as long as there are runnable, pending jobs.
- Initiate the execution of the program from any participating node.
- Execute portions of the program other than maps and reduces locally on the initiating node, or on other nodes, as your design should dictate.
- Allow each participant to run multiple jobs concurrently and correctly, without any concurrency related problems, except as the result of programmer-visible and mitigatable sharing.
- Schedule and dispatch maps and reduces, to maximize the performance gain through parallelism within each phase, subject to the constraints of the initiating program.
- Recover from failure of map and reduce workers.
- Provide a general-purpose I/O facility to support the necessary operations.
- Provide management tools enabling the start-up and shut-down of the facility, as well as the management of jobs, e.g. start, monitor, and stop.
- Provide documentation for system administrators describing system requirements, configuration steps, how to confirm successful configuration, how to start, stop, and monitor the system, and how to run and manage jobs.
- Provide documentation for programmers, describing each of your MapReduce library and your I/O library, and how they are used together. This should include programmer references, as well as a tutorial with at least two described examples.
- Include at least two working examples, including code and data or links to data.

3. Main Components

- **Master(s)**: keeps track of compute nodes as they join, leave, or fail.
- **Compute Nodes**: talks to master(s) and execute assigned tasks.
- **Programming API:** defines how MapReduce programs should be written to run in the system.
- **Scheduler**: manages assignments of MapReduce tasks to compute nodes, including reassignments upon failures.
- **Distributed File System:** allows parallel access to data used by MapReduce tasks.
- **Management Tools:** provides the administrator with means to check and manage MapReduce tasks and nodes. Also, there should be those for checking the status of the distributed file system.
- **Example Programs:** demonstrate the functionalities of your MapReduce framework, exemplifying how programs can be written with the programmer API.

3.1. Master(s) and Compute Nodes

Your master(s) and compute nodes should maintain coordinating processes on themselves that are brought on-line as part of the systems initiation, retired as part of its termination, and restarted, when able, if failed. The system should not start these processes, worker or master, with each job – though the system is welcome to use threads or create additional processes as appropriate. It might be helpful to recall that the system gets its power through the use of multiple cores and multiple hosts in parallel, especially mappers, not by queuing work. The goal here is to reduce the dispatch latency and to minimize the overhead across multiple runs and multiple jobs.

For the convenience of the user, and to enable the distribution of the parts of the program that are not maps and reduces, processes should be able to use the MapReduce facility from any participant (you can decide whether or not this includes the master(s)).

3.2. Programming API

You should provide a library that provides a convenient interface for application programmers to solve problems by performing Map and Reduce phases. A *map* operation should take an pointer to a partition of input records on disk (for example, filename and record number range), a pointer to a file to which to append the output of the map operation, and an operation to perform on each input record to map it to an output record (function pointer, function object, functor, etc). A *reduce* operation should take the same types of inputs, but should produce only one output per input partition. To keep things simple, you can assume that the partitions are homogenous and that reduce is intended to be left-associative.

Note that the result type of a map or reduce operation can be different than the input type. To mitigate this, you might want your reduce operation to take an initial value of the result type, in effect the identity for whatever the operation.

3.3. Scheduler

The power of the paradigm largely derives from the fact that Mappers can be executed in parallel across the provided list or sequence of data. Your system should schedule in a way that enables this – without overloading a host beyond what is productive. Although there is often less parallelism in Reduce phases the same is true. Also, don't forget about the possibility for multiple programs to run simultaneously – your scheduler should support and enable this in cases where resource availability permits it. The goal here is for the scheduler to dispatch maps and reduces to maximize the performance gain through parallelism within each phase, and across multiple programs, as much as is possible, subject to the constraints of the logic of the program(s), themselves.

Also, the scheduler should consider data locations, as described in Part 3.4. Specifically, the scheduler can assume a constant cost (namely, one) for data transfer between each pair of nodes, and should try to minimize the number of transfers needed when running MapReduce tasks.

3.4. Distributed File System

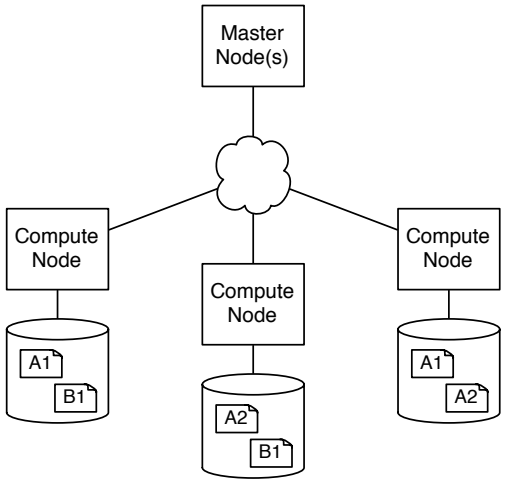

Figure 1. Distributed file system with replication factor of two

You need to implement a distributed file system, similar to Hadoop Distributed File System (HDFS). The goal of your distributed file system is to work efficiently with your MapReduce framework, providing parallel access to input data. Figure 1 illustrates the idea. Each computing node also acts as a participant in the distributed file system, and holds part of the files stored in it. The file contents are replicated across machines, allowing them to be accessed by MapReduce jobs concurrently. For example, in the figure file A is split into two parts, A1 and A2, and each is stored on two machines. Thus, if Mappers are scheduled and given input appropriately, two of them can work concurrently to

process file A. This degree of replication, which is two in this case, is called the *replication factor*. The efficiency of your MapReduce framework is achieved by scheduling your tasks in a manner that considers data distribution.

Your distributed file system should:

- Take a configuration file specifying the set of participating nodes and the replication factor. In addition, it should have a way to bootstrap itself by taking this configuration and data files, distributing them among the nodes and allowing your test programs to access them.
- Have a policy for distributing data among the participants; a simple one suffices for this project that avoids obvious bad decisions like duplicated contents on the same node.
- Provide a simple flat name space, without a hierarchical structure. If data being accessed does not reside in the current node, it should be retrieved transparently from another node holding it (which could create a new temporary replica on the current node).
- Should have some means of allowing interaction between the scheduler and itself.
- Be resilient to node failures, providing access to particular data even if some, not all, of its replicas are inaccessible.
- Provide an API for accessing its files from your MapReduce programs, exposing the flat name space.

Intermediate results produced by Mappers should be stored in the local file system of their machines (e.g., under /tmp on Andrew machines, which you would want to clean afterwards). Your MapReduce tasks access data either through your distributed file system or the compute node's local file system, NOT AFS.

You can assume that files in the distributed file system are immutable once written. Namely, after bootstrapping, input files are only read by Mappers. Also, Reducers should store the final results in the distributed file system.

Additional replicas, created as a result of remote file accesses, should be stored as local cache for subsequent accesses (i.e., they are *not* like original replicas, which may be retrieved from other nodes). Note that the bootstrapping phase must respect and enforce the replication factor for the original replicas stored in the distributed file system.

As stated before, the scheduler should be able to interact with the distributed file system to reduce the number of data transfers. For input files for Mappers, this cost would be equivalent to the number of extra replicas created since the initial set-up of the file system.

3.5. Management Tools

You want to make this facility easy to use. This is for your benefit as much as for anyone else – you'll be the heaviest user of your own system! This is also crucial for your users, including the graders.

You want to make it really easy to start your system, to stop your system (cleaning up all of the cruft), to start jobs, to find out what jobs are queued, what jobs are running, where the Mappers and Reducers associated with jobs are running, and to kill jobs, including their Mappers and Reducers. The same thing applies to the distributed file system. You want to be able to see the current distribution of data. Take the time to do this right – you'll really help yourself. We promise.

3.6. Example Programs

You should turn in two examples of MapReduce programs, including code, data, and documentation. They should demonstrate that your system can parallelize Mappers and Reducers, and combine results into a single file (or far fewer files than mappers). You should turn in the data, or pointers to it, e.g. URLs (make sure to credit the source). You may generate your own data, or find data on the Web. You may want to write a program or script to preprocess the data to make the record format simpler, as discussed in the File I/O section, to fit your file I/O model. In general, it is likely to be easier to preprocess your data than to simplify its form than to make a more general, more complex I/O model.

The most important objective of these programs is to exercise the features of your system in a manner that allows them to be confirmed. E.g., in order to show that your system parallelizes tasks well, there needs to be a program that does benefit of parallel execution. Also, note that there should be a form of evidence. E.g., parallelism could be made visible by the capability to check the status of running tasks and compute nodes. Please try to provide test cases that cover as many of the system's important features as possible, such as parallelism, efficient scheduling, and resiliency.

4. Requirements

4.1 Concurrent Use

Keep in mind that your system may be used concurrently by many users. It should not have any internal concurrency related problems. Having said that, if programmers break things by, for example, sharing files unsafely – that is certainly nothing the system can prevent. Systems such as this do not generally include file locking tools etc., because their use could destroy throughput. For this reason, is better to manage the jobs, or which files they use, than the files, themselves – which you can leave to the user and application programmer, respectively.

4.2 Failure and Recovery

Your system should provide for the graceful recovery from short-lived transient failures, as well as longer-lived failures on, or affecting, participants. This includes recovering from network glitches directly affecting the host, or affecting, for example, the host's connectivity to AFS. It should also include the failure of the host, or the death of the coordinating process on the host. The goal here is to continue working when possible, restart the work on the same host when appropriate, and to restart on a different host, if appropriate.

You are only required to manage failure on participants, not master(s). But, you are welcome to use redundant masters, a committee of masters, or any other scheme you'd like to improve the resilience of the master, or the system as a whole, should you choose – but it is not a requirement and will only earn respect, admiration, and bragging rights – not points.

4.3 Configuration

You should use one, or more, configuration files, which are readily human-readable and human-editable, to configure your instance, including identifying the participant node(s), master node(s), port numbers, maximum maps/host, maximum reduces/host, etc. You shouldn't have any "magic numbers", "magic paths" etc. in your code. These files may take the form of run-time configuration files, such as files containing "PARAMTER=value" pairs, that are read at initialization.

The prime directive here is to make it easy and intuitive for a user to go from a blob of your source files to a real-world deployed system with as little effort as possible. To this end, self-documenting files are better than those that require external documentation, etc.

Note that you'll be sharing hosts with many other students, in this class and others. Try to pick a random distribution of machines and port numbers to spread the load around, and select others if you come upon heavily loaded or latent host(s).

4.4 File I/O Library/Interface

In order to perform map operations, the mapper will need a partition of data and the map operation to perform upon each element of it. The same is true of the reducer, which will need to pairwise combine the elements using the provided operation.

As a result, the File I/O model is one that views a file as a collection of records. The I/O library/interface needs to make it easy to access a partition of records within the file and to access each record within that partition. For simplicity, you are welcome to simplify the problem by considering only files with fixed-length records with fixed structures, such that it is possible to seek to any record within the file by record number, based upon the record size. Of course, the record structure and corresponding fixed size should be determined by the application programmer, not your library/interface, though.

You should implement a library/interface to make this type of operation convenient, to enable you to design a clean interface for mapper and reducer functions, and the application programmer implementing mappers and reducers to do so cleanly.

Your library may only use standard libraries in the language you are using, such as *RandomAccessFile* in Java. But, you might want to look at *Hadoop's* implementation, for example *RecordReader*, for inspiration. Your solution will be much simpler if you limit it to fixed-length records with a fixed structure.

You can, of course, choose to support anything more general or agile than fixed-structure/fixed-length records. But, you won't get extra points – just more bragging rights, respect, and admiration.

Your mapper and reducer will also need to be able to write data. This is likely to be a much easier problem and may well be solvable using native I/O function.

What this library/interface looks like will vary dramatically depending upon your language of choice. What you want to be able to do here is to make it easy to point your mapper and/or reducer at file and access their partition of the records, and easy for each of them to write out their results. Your solution should lead to a clean interface for the mapper and reducer functions and also a clean implementation of the record input, output, and parsing code.

5. Report

A dramatic portion of your score will relate to the quality of your documentation. Your project needs to include:

- The overall design of your MapReduce framework and distributed file system. What their main components are, what their roles are, and how they interact with each other.
- Documentation for system administrators describing system requirements, configuration steps, how to confirm successful configuration, how to start, stop, and monitor the system, and how to run and manage jobs
- Documentation for the application programmers, describing each of your MapReduce library and your I/O library, and how they are used together. This should include a reference manual for the APIs, as well as a tutorial with at least two described examples (This documentation, and the documentation mentioned in the *Examples* section are one and the same).
- How the features of the MapReduce framework and distributed file system can be demonstrated, using test programs, management/monitoring tools, log files etc. Specifically, the key features include, but are not limited to, parallelism, resiliency, and file distribution. Steps for confirming such features need to be clearly explained, so that the readers can perform them on their own.
- The requirements you met, the requirements you didn't meet, the capabilities and limitations of your project, what you would improve with more time, and anything you did "above and beyond" or in a really cool way.

We strongly recommend you to try thinking from the readers' perspective, assuming they have no knowledge about your work, or this project. How would you need to write to help them understand your work? Please do not dive into details like explanations of classes and methods. They are best placed in your code.

6. Submission Guidelines

- Create a directory named "FirstAndrewID-SecondAndrewID" at /afs/andrew/course/15/440-f13/handin/proj3. If you need to submit updated versions, create new directories, naming them "FirstAndrewID-SecondAndrewID.2" "FirstAndrewID-SecondAndrewID.2" etc. We will look at only the newest version.
- Provide a Makefile or something equivalent (e.g., for ant). We will need to be able to compile your submitted code on Andrew machines. In particular, we will not be able to use Eclipse for importing and compiling your project.
- All your source code should be placed in a directory named "src," under the root of your submission directory.
- Avoid any hard-coded parameters, such as host names, path names, and port numbers.
- Submit your report in PDF, named "report.pdf," under the root of your submission directory.
- Test your implementation on Andrew machines before submission.
- Please do not submit a revision history with your code. E.g., no .git in your source directory.
- Make your code readable by others, with appropriate indentation and comments.

Please note that submissions not following these guidelines can significantly affect the grading process.