15-440 Project #3: Building a Map-Reduce Facility
Summer 2014/Kesden

Introduction

This project asks you to implement a *Map-Reduce 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 maps and reduces across multiple hosts, as well as recover from worker failure. You will document the API for your MAP reduce facility, and example programs, also thoroughly document, to demonstrate its capability

Requirements Overview

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

- Operate on the unix.andrew and/or ghcXX.andrew hosts, as well as other similar systems
- 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, such as by maintaining processes at all participants, rather than by launching them anew for each job.
- 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
- Execute several 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 map-reduce 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

Environment

In order to facilitate the grading of your project, it should operate on the unix.andrew and/or ghcXX.andrew hosts, without the installation of additional software. Your solution should not contain any hard-coded tying it to these systems, and should be configurable to work on any set of similar, modern Linux hosts.

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. Or, they can take the form of compile-time configuration files, such as header files, files containing the definitions of configuration constants compiled into the program, etc. Or, your project can use some combinations of both.

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).

Processes on Participant(s) and Master(s)

Your participant(s) and master(s) should maintain coordinating processes on each master or participant 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.

Running a Map-Reduce Process

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 map-reduce facility from any participant (you can decide whether or not this includes the master(s)).

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.

Scheduling

The power of the paradigm largely derives from the fact that maps 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.

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 hosts 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 resiliance

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.

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.

Map-Reduce Library

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.

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!

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.

Take the time to do this right – you'll really help yourself. We promise.

Examples

You should turn in two examples of your system, including code, data, and documentation. They should demonstrate that your system can parallelize mappers, parallelize 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. Your should be sure to credit the source in your documention.

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.

Documentation

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

- 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 map-reduce 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).
- A report on your project that describes 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.