**Report (Project 3)**

sdai, kpayson

**Overall Design:**

The primary components to our Map-reduce framework are the following:

**MapReduceManager/SystemNode**: This management section of the framework is dedicated to setting up the system (bootstrapping) by taking configuration settings and initial distribution of files. This is the highest level of abstraction in our system. Later, when tasks are being run on the machines in our system, the MapReduceManager has the ability to track progress of tasks on individual work nodes.

**Master/Scheduler/ComputeNode**: The Master/Scheduler's primary job is to distribute work across our work (Compute) nodes. The master has knowledge of all of the map/reduce tasks our system needs to perform, and even distributes them across all compute nodes, by using the scheduler. Also, the Master keeps track of the progress each ComputeNode is making on their given task, so it can determine when to send each node more work, or restart the task if the node fails. The way in which it does this is explained in more detail in the next section. The compute node simply executes the map/reduce task given to it by the master node.

**MasterDFS/DFSNode**: These nodes are for the distributed file system(DFS). There is a single MasterDFS instance, which corresponds to the Master class, which primarily handles location queries (asking where a given file is located, or asking where we should send a newly created distributed file). The DFSNode represents the node of the distributed file system at a given machine. So, this node locally stores some files that are part of our DFS. This node also handles retrieving temporary copies of files that are not local to this machine, creating new distributed files, and receiving requests for a distributed file that is stored locally at this node.

**MapReducer**: This class is simply an interface that the application user should implement, if they want to run a map-reduce job (this is the application programmer’s part).

Note: We also have several wrapper/transport classes to aid us in achieving the tasks performed by the classes above, but they are not that important from an overall design perspective.

**Specific Design Decisions:**

**Bootstrap/Initialization:**

To start our map reduce system, we process a config file that defines which machines will be a part of the system, and other related information (for more detail, see setup instructions). Every potential machine in our system should be running a SystemNode (So, every machine that could be running a Master/Compute node). Then, after we parse the config file, we start the Master/MasterDFS and ComputeNode/DFSNodes by telling each SystemNode specified to start a Master or a ComputeNode, depending on the configuration of the system. This way, the user does not have to manually set up each machine to be a Master of ComputeNode, but can simply specify which a machine will be through the config file. Additionally, files are initially split and distributed through the system here. Furthermore, the MapReducer class defined by the user is shipped to all nodes. After distributed files are specified at bootstrapping, our system breaks each file into a manageable number of pieces (there IS an option for the user to specify this, but typically, our system will do it automatically, to maximize work distribution), then replicates them according to a supplied replication factor. Then, we use our DFS to send copies of each piece of the file to different nodes. We ensure that no node will get 2 copies of the same file, and that the files will be evenly distributed across all machines. After this bootstrapping/setup phase, a user can, from any compute node in the system, provide a map-reducer and run their map-reduce program through our system. Then, the work will be distributed through the scheduler and performed on the individual compute nodes.

**Scheduler:**

The scheduler is designed to minimize the number of active nodes while minimizing the number of temporary file copies. In order to create an initial schedule, a maximal covering bipartite graph is established with files/nodes as vertex nodes and an edge between the files contained on nodes. This is done using an augmenting path algorithm. Remaining unmatched nodes are tasked arbitrarily. Following this step, when a node becomes free, it will query the scheduler for a task, and at this point if there is a task that uses data located on that node, it will be assigned, otherwise, it is arbitrary. Because reduce tasks will have their data spread relatively evenly across the system, they are given no preference to node-task pairings. In order to support scheduling for multi-processor systems, we simply will begin by scheduling by allowing up to N tasks to be on a node, and then proceed in the normal fashion, every time a processor becomes available, we give it the best task for that node at that time.

**Map-Reduce:**

Our Master will use our scheduler to determine which nodes should be performing which tasks. Then, the Master will tell each ComputeNode what to do. Upon receiving a task, the ComputeNode will start runnning the task, periodically sending a heartbeat to let the Master know it is still working. When it is done, it will also notify the Master, which will then assign it more work (if there is work to be done).

The compute nodes start by receiving map tasks given by the scheduler. The user defined Map function then maps over these values.

The partition function is then called on the results of each map, and then multiple files (local to the map) are created containing map values from each of the partitions.

Once all the necessary files are available, the scheduler will assign reduce tasks to each of the each of the nodes that will make local copies of this partitioned map data, and then run the reduce operation on it. The results of the reduce operation will be placed in the distributed file system in files named result-\*.txt, where \* is the partition that that is the result of.

**Failure/Recovery:**

The Master expects to receive heartbeats from the nodes at fixed intervals, and if it does not receive a response (with some delayed window to account for network latency), it will assume that the node has failed, and will send the task to some other node. However, if a node goes down, we do not re-replicate the files that were originally on that machine, so if we lose all of the replicated machines for a given file, we would lose the file (this could be improved).

**DFS/Files:**

On creation of a new file, we first notify the MasterDFS server, which will tell us which nodes will host the file and its replicated copies. Then, we send a copy of the file to each replicating node. To access a distributed file, that is not local to the current machine, we would send a location query to the MasterDFS server, and then make/cache a temporary file locally (under the /tmp directory, which should be manually cleared after each execution), which would be used. If a file node has died, then we try each other file node that contains this file. If all are down, our system would be unable to recover the file, since every copy is inaccessible. After the maps are done, the results are stored in temporary files (also under the /tmp directory), that would get locally combined/merged into a single local intermediate file, then fed into the local reducers.

Files are processed line by line. We do not explicitly require a record implementation by the user. Rather, we use implicit records, by passing in the entire String line of each file as a record (making the user defined map and reduce take in and return String arguments), and having the user defined map/reduce functions implicitly convert lines to and from their records. Advantages of this method versus an explicit record type conversion, is that we do less preprocessing of the data, and the application user should have a better idea of how to parse/generate their records than our system will (intuitively, the user will have a better semantic interpretation of a record than our system will, see our examples for a more concrete idea). However, this comes at the cost of some robustness and flexibility, but still behaves effectively the same.

**API/Interaction between Map-Reduce and DFS:**

Importantly, the Master node has a MasterDFS server associated with it, and each ComputeNode has a DFSNode associated with it (equivalently, the MasterDFS has an associated Master node, and each DFSNode has an associated ComputeNode). This is to separate the file I/O and network traffic of file requests/sends from the actual map/reduce work being performed. For example, a ComputeNode could be running some map task, but another ComputeNode needs a file that is on the first node, so it uses its own DFSNode to send a request for a copy of the file to the first ComputeNodes's DFSNode. Then, the file transfer happens between the DFSNodes, rather than the ComputeNodes which could be running a task. The API for the DFS is contained entirely within the DFSNodes, the MasterDFS, and a static DFSUtil class. MasterDFS is strictly a server that responds to requests that ask where a specific file is located, where to send replicated copies of a new file, or where to send .class files at bootstrapping (functions as a central hub of information, so basically the Master but for files). DFSNode has methods to create new distributed files on that node, but also has a server component, where it receives requests for a file on that node (responds by sending a copy of the file), or it can receive a new file that its associated ComputeNode has requested. Lastly, the DFSUtil class has static methods that given a filename, and a MasterDFS server location, asks where the copies of the file are, and retrieves them, generating a local copy (there is also a sendFile method, which simply sends a node a copy of a file). Once a node has a temporary local copy of a file, they can read it as they would any other local file (Note: they could even write, but it would be worthless, since the file is only a temporary copy). These classes combined can perform all of the DFS-related tasks our system would need to perform.

**Plugging into the framework:**

Our framework really only needs some number of machines that act as nodes in our system, a config file that describes how we set up and bootstrap our system, and a MapReducer program that we can run on our system. Then, starting our framework with the above should run the map-reduce program in the system, producing the desired result in the form of a series of distributed files (1 per reducer).

The config file contain lines of the form: "PROPERTY = VALUE" (see example config files for a better idea)

Beyond the obvious port location and host location in the config files, we have

REPLICATION - The DFS replication factor

CORES - The number of cores (simultaneous tasks) on each machine. Note for simplicity we assume a heterogeneous environment.

PARTITIONS - integers 0-PARTITIONS-1 are valid outputs of the partitions function, this will be the number of partial results produced by the complete map-reduce function.

DATAFILE - the data that you want to map-reduce over

MAXFILESPLIT - the maximum splits of any file in the DFS

Once the system is running, the user has the option to kill the operation and to query the state of the map reduce. For simplicity, when the user query's the state of the map reduce, they get a mapping from nodes to tasks (which contains information such as if its mapping/reducing, over what data, its PID ext). It also contains a list of active nodes, as well as a mapping of nodes to files. For more information, see Task.java and MapReducerState.java.

**Running our Examples:**

We have two examples, Histogram and MaxInt. Histogram maps a list of words to a word int pairing that counts the number of occurrences of that word in the text. MaxInt maintains a list of occurrences of the largest integer in a list of integers. When a larger integer is encountered, the list resets. This result is portioned based on the interval the integer resides in so we get 2 results, one for each interval out.

**Requirements/Improvements:**

We should have successfully implemented all of the basic requirements.

Cool parts of our DFS:

One interesting feature of our system is the systematic deployment of the users .class file associated with their MapReduce task. This is done using dynamic loading of classes and reflections in java. This makes it easy for the programmer to deploy their map-reduce task.

Things that could be improved:

We could, upon failure of a node, re-replicate the files that were on that node to maintain the replication factor. We also have no means of recovering the Master node, so if the Master goes down our system dies. We could implement some system that would either recover the master, or have multiple masters, so losing 1 would not halt progress. However, this kind of "truly" distributed system also has more overhead cost.

We could have improved our file splitting, especially for multiple file cases. Rather than split each file into the same number of pieces, we could split the files into pieces such that the map tasks would take approximately the same amount of time for any partition of any file.

We could have automated the removal of tmp directories after a map reduce task, but currently the user must do this.