Locality Aware Fair Scheduling for Hammr

Li Jin

January 12, 2012

Abstract

Hammr is a distributed execution engine for data parallel applications modeled after Dryad. In this report, we present a locality aware fair scheduler for Hammr. We have developed functionality to support hierarchical scheduling, preemption and weighed users and a minimum flow based algorithm to maximize task preference. For evaluation, we've run Hammr on Hadoop Distributed File System on Amazon EC2 Cloud with five instances.

1 Introduction

Hammr is a distributed execution engine modeled after Dryad [4]. It is similar to MapReduce[3] that provides a way to express and execute data parallel applications and aims to run these applications on cluster with thousands of machines. The main difference between Hammr and MapReduce is their computational model. Instead of expressing the application as multiple map and reduce phases, Hammr expresses the application as directed acyclic data flow graphs. Also, in MapReduce, data is always passed through file channels, whereas in Hammr, data could be passed through file channels, tcp channels and shared memory channels. Hammr's computational model is more complicated, flexible and brings more opportunity to optimize the application than MapReduce, on the other hand, it also brings difficulty and challenge to the design of the scheduler, as we will see in the following sections.

The original scheduler for Hammr is trivial: When a task is ready for scheduling, it is assigned to a slave machine randomly. This strategy obviously has several limitation (1) it doesn't take into account the capacity(how many tasks can executed at the same time) of each machine (2) it doesn't take into account the preference of each task (3) it doesn't have any mechanism to support fair sharing among multiple users.

In this report, we present a locality aware fair scheduler for Hammr that solves the limitation mentioned above. We uses a hierarchical scheduler to perform fair scheduling among multiple users and a minimum flow based algorithm to find the optimal scheduling plan maximizing task preference.

The reminder of the report is organized as follows: in Section 2, we provide an overview of the Hammr system.

2 Related Work

Fair Scheduler[2] for Hadoop[1] is a project developed in Facebook to support fair sharing of the cluster between multiple pools. The hierarchical described in the following section is similar to Hadoop Fair Scheduler but adjusted to the Hammr system. Also, Quincy[5] is a fair scheduler developed by Microsoft for Dryad. It uses a minimum cost flow based algorithm to perform global scheduling. However, Quincy is not publicly available. We adopt the idea of minimum cost flow and adjust the algorithm to perform locality aware scheduling as described later in details.

3 Hammr System

In section, we briefly describe the Hammr system. The system is written in Java, using Java RMI technology with about 5000 lines of code. Hammr is a typical master-slave system just like Hadoop and Dryad. The master, called Manager maintains a list of slaves, called Launchers. The Manager is very similar to Job Tracker in Hadoop and is responsible for setting up applications, maintaining state of the system and assigning tasks to Launchers. The Launcher, similar to TaskTracker in Hadoop, receives task from the Manager and does the actual computation. There is one Launcher for each physical machine.

The structure of a Hammr job is determined by its communication flow. As mentioned above, a Hammr job is a directed acyclic graph where each vertex is a program and edges represent data channels. These vertices, called Nodes, are the basic computation units that take input, do processing and send output. The input of a Node comes from either initial input files or other Nodes. Similarly, the output of a Node goes to either final output files or other Nodes. Nodes are further grouped together according to their data channels. A NodeGroup is a set of Nodes that communicates (sends and receives data) with each other using shared memory channels, which implies that they need to be scheduled on the same machine. A NodeGroupBundle is a set of NodeGroups that communicates with each other using tcp channels, which implies that they need to be scheduled together. Finally, a Hammr job is broken into several NodeGroupBundles and each of them can be scheduled separately (Note that one NodeGroupBundle might need the output of another NodeGroupBundle, in which case an ordering is forced, otherwise they can be scheduled independently).

The Hammr system assumes an underlying distributed storage system. The original Hammr system doesn't have any additional requirements about the underlying storage system. However, since our scheduler is locality aware, we require the underlying distributed storage system to provide locality information about where the data is stored. Here we use Hadoop Distributed File System because is the widely used, well developed and meets with our requirement. Though this report, we assume Hadoop Distributed File System is used for storing all files, both input and output files of the Hammr job and temporary files work as file channels. Note that in Hadoop, temporary files are stored as local files and each TaskTracker has a file server to transfer these local files. We don't use the same approach because they are fundamentally the same and the approach used in Hadoop is an implementation optimization.

4 Resource Model

For the purpose of limiting number of Nodes running on a Launcher at the same time, we use a simple resource model to present physical machines. A physical machine is broken into several resource slices called slots. Each slot can take one Node. The idea of slots are used in Hadoop and is a simple and clean way to express the machine resource. Usually, a slot consists of one CPU core and several gigabytes of memory. This model, though not perfect, is simple and widely used therefore considered a good fit to build our scheduler on. Though this report, we will assume such resource model is used.

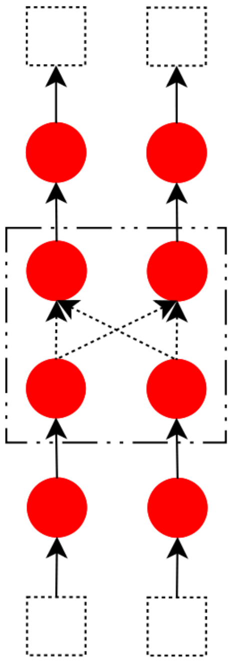

Figure 1: An example of Hammr job. Dashed squares on the top and bottom represent input and output files. Red circles represents Node. Solid arrow lines represent file channels. Dashed arrow lines represent shared memory channels. Dashed selection in the middle represents a NodeGroup.

5 Fair Scheduling

In Hammr, jobs are grouped into pools and each pool can present one user or multiple users. We define the fair share of a pool to be the number of slots that should be given to pool based on its weight. For instance, if pool A with weight 1 and pool B with weight 2 is sharing a cluster of 60 slots. The fair share for pool A and B is 20 and 40, respectively. If the demand of a pool is less than its fair share, we reduce its fair share to its demand and give the spare slot to other pools according to their weight. For instance, in the above example, if pool B only need 30 slots, we give the spare 10 slots to pool A therefore the fair share of both pool is 30 slots. The goal of fair scheduling is to keep the slots given to each pool same or close to its fair share. At any moment, the share of a pool is the number of slots given to it and the weighted share equals share divided by pool weight.

5.1 Hierarchical Scheduler

At a high level, the fair scheduler uses hierarchical scheduling to assign Nodes to Launchers. First it selects a pool whose weighted share is the smallest to provide a NodeGroupBundle. Then the selected pool pick a NodeGroupBundle among its jobs using its internal scheduling policy, for instance, FIFO or fair sharing. Using this approach, each pool can use their own scheduling algorithm regardless of what algorithm is used on the first level, providing flexibility to the whole system.

To implement the hierarchical scheduler mentioned above, we introduce Schedulable interface that contains a scheduleNodeGroupBundle() method to return an assignment from NodeGroup to Launchers. Recall that NodeGroups in a NodeGroupBundle must be executed at the same time and Nodes in a NodeGroup

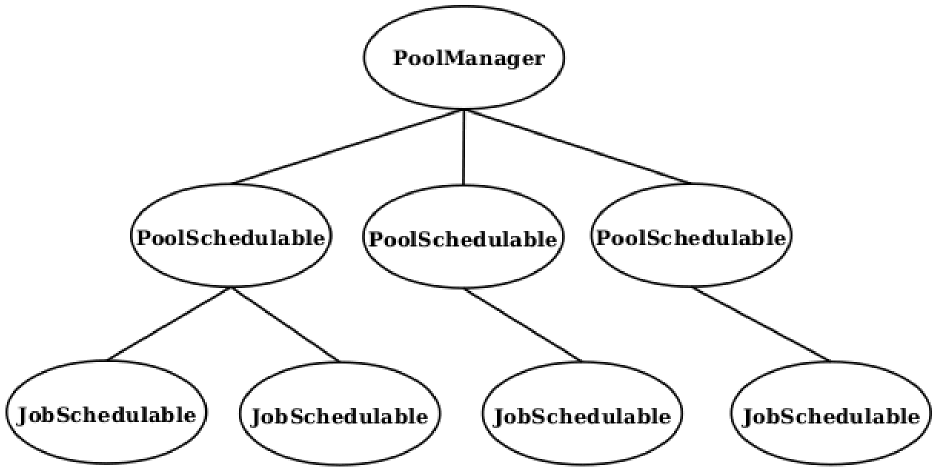

Figure 2: Hierarchical scheduler. All classes implement scheduleNodeGroupBundle method to be called by the higher level.

must be executed on the same Launcher because of the restriction of different data channels.

There are several classes that implements Schedulable interface, they are PoolManager, PoolSchedulable and JobSchedulable. We describe them in a bottom up order.

- JobSchedulable represents a Hammr job, its scheduleNodeGroupBundle() method picks the most preferred NodeGroupBundle with in a job, given the global available resource and calculates and assignment for the NodeGroups in the NodeGroupBundle. We will describe the algorithm to find such assignment in the Section 5.
- PoolSchedulable represents a pool, containing a list of JobSchedulable. Its scheduleNodeGroupBundle() method picks a JobSchedulable, either using FIFO or fair sharing policy and calls the scheduleNodeGroupBundle() of that JobSchedulable.
- PoolManager behaves similar to PoolSchedulable, but instead of keeping a list of JobSchedulable, it keeps a list of PoolSchedulable. The fair sharing among pools is implemented in PoolManager.

Finally, a scheduling thread in the Manager calls scheduleNodeGroupBundle() of PoolManager periodically.

5.2 Preemption

A pool can be staved for its fair share. A common situation is that a new pool jumps in when the cluster is fully utilized. Although it is the pool with the smallest weighted fair share, still, it could be a long time until enough Nodes finish and slots are assigned to the new pool. To deal with such situation, we've also implemented preemption functionality. To determine when to preempt tasks, the Manager maintains the last time when the pool was at two-thirds of its fair share for each PoolSchedulable. The value is periodically checked and updated by a update thread in the Manager. When a PoolSchedulable has been below two-thirds of its fair share for more than a threshold value, the update thread preempts Nodes of other over scheduled PoolSchedulable and make space for it. Here we use a soft kill approach, instead of killing the Nodes immediately, we give the Nodes some time to do some work and perform self termination.

6 Locality Aware Scheduling

In a typical distributed computing system like Hadoop and Hammr, storage node and computation node reside on the same physical machine. Therefore, we can reduce IO costs by schedule tasks close to their input. The locality aware scheduling is developed for this purpose.

6.1 Cost Model

First we define the cost model of our scheduling. Here we only consider file channels. Though it makes sense to schedule two Nodes communicate using tcp channels close to each other, we find it too difficult to do here. This might be a interesting future work. The shared memory channels do not involve any IO costs and therefore we also don't consider them here. The file input data of a Node is divided into two part, local input which means such input data is local to the Node and is considered low cost data, and remote input which means such input data is on some remote machine and is considered high cost data. Our goal is to find an assignment that minimize such data cost, given a NodeGroupBundle and the current available slots in the cluster.

6.2 Minimum Cost Flow Based Scheduling Algorithm

With the cost model described above, now we define our assignment problem: Given a NodeGroupBundle, current available slots and data locality information, find and assignment from NodeGroup to Launcher that minimize the total data cost. We solve the problem by mapping the assignment problem mentioned above into an integer minimum cost flow problem. The minimum cost flow graph is constructed as follows:

- The first level is a source vertex S. S sends an amount of flow d, where d is the number of NodeGroups in the NodeGroupBundle.
- The second level consists of d vertices Na,Nb..., each for a NodeGroup. An edge with capacity 1 and zero cost exists between s and each vertex in this level.
- The third level consists of a singe vertex X, called the cluster vertex. An edge with unlimited capacity exists between each vertex in the second level and the cluster vertex. The cost of each edge is set to the total data cost of that NodeGroup if all of its input data is remote. This means if a flow goes though such edge, that NodeGroup corresponding to that flow is scheduled on an arbitrary machine in the cluster and therefore we just assume all of its input data is remote, for the worst case.
- The fourth level consists of n vertices La,Lb...., each for an currently available launcher in the cluster. An edge with unlimited capacity and zero cost exists between the cluster vertex and each vertex in this level.
- The last level is a sink vertex t, t receives an amount of flow d. An edge with zero cost exists between t and each vertices in the fourth level. The capacity of each edge is set to the number of empty slots available in the corresponding Launcher. The number of NodeGroup that can be scheduled on each Launcher is limited by the capacity of these edges.

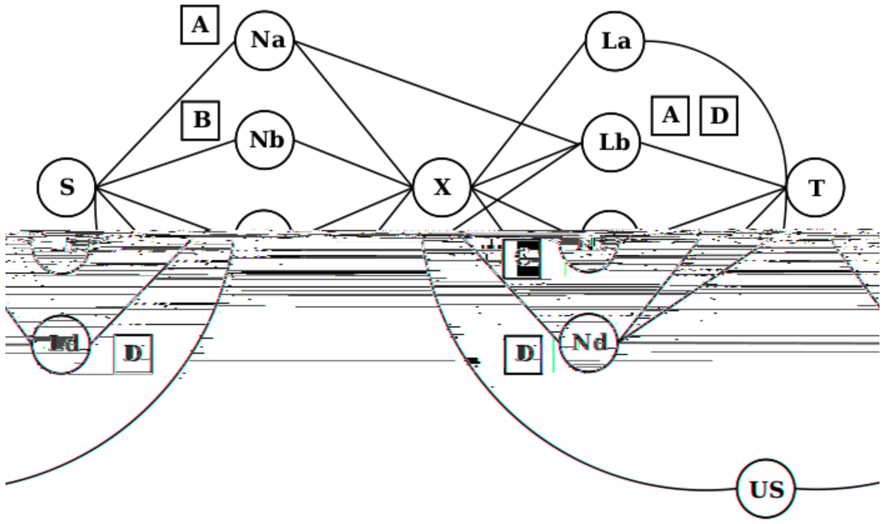

Figure 3: We are scheduling a NodeGroupBundle consists of four NodeGroups, represented as vertex Na, Nb, Nc, Nd in the graph. Vertex La, Lb, Lc, Ld represent currently available launchers, for simplicity, we assume each launcher has one available slot. Squares marked as A,B,C,D are input data for each NodeGroup. Lb has a local copy of A,D and Ld has a local copy of D. Each NodeGroup is connected with each launcher through vertex X, meaning that each NodeGroup can be scheduled on each launcher, but with a high cost. There are additional edges connecting Na, Nd and Lb, Ld since input of Na and Nd are local to Lb and Ld. These edges are low cost edges.

- If a NodeGroup Ni has local input data on a Launcher Lj, an edge with unlimited capacity is generated between Ni and Lj. The cost of the edge is set according to the amount of local data on that Launcher. If a flow goes though this edge, Ni is assigned to Lj and therefore the data cost is lower.
- A special vertex US, stands for unscheduled, connects S with T directly to make the minimum cost flow problem always feasible. The cost of going through this path is set to a high value.

Figure 3 and Figure 4 give an demonstration of how this algorithm works.

It is not hard to prove that the assignment problem is equivalent to the minimum cost flow problem, and the minimum cost flow problem can be solved use a standard solver.

7 Evaluation

We deploy Hammr and evaluate the performance on Amazon Elastic Compute Cloud (Amazon EC2). The evaluation of the system consists of two parts. First, we evaluate the behavior of the fair scheduling. Second, we evaluate the behavior of the locality aware scheduling. As mentioned above, we use Hadoop Distributed File System as the distributed storage layer for Hammr. We use a word count MapReduce program, described later in detail, for all our experiments.

7.1 Environment Description

We use an Amazon EC2 cluster with five instances, including one master and four slaves. The Amazon instances are of type m1.large, with 7.5GB of memory and 2 virtual cores. We deploy Hadoop Distributed

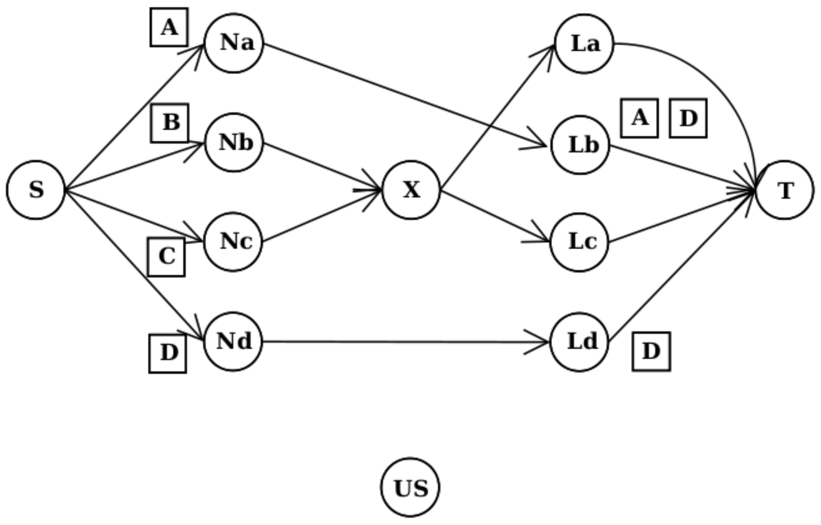

Figure 4: The resulting minimum flow. A flow of 1 goes through each edge in the graph. A flow goes from a NodeGroup directly to a Launcher means that this NodeGroup is to be assigned to that Launcher. A flow goes from a NodeGroup to vertex x means that this NodeGroup is assigned to an arbitrary Launcher in the cluster. Here we assign Na to Lb, Nd to Ld, and Nb,Nc to La,Lc.

File System on the cluster, with one NameNode on the master machine and one DataNode on each slave machine. The blocksize is set to 512m and the replication factor is set to 1. Hammr is deployed on the same machines as Hadoop Distributed File System. The Manager is running on the master machine and one Launcher on each slave machine. The number of slots on each launcher is set to 2, the number of virtual cores on each instance.

7.2 MapReduce Interface for Hammr

For the purpose of comparing Hammr against Hadoop, we develop a MapReduce interface for Hammr so that it is easy to write Hadoop style MapReduce job in Hammr. We use a simple word count MapReduce program for all our experiments. The number of mappers is set to the number of data blocks of the input and the number of reducer is set manually in individual experiments. We use file channel for all communicate between Nodes. Therefore, the structure of the Hammr MapReduce job is identical of a Hadoop MapReduce job.

7.3 Fair Scheduling

We design the following experiment to evaluate the fair scheduling. Three pools, A, B and C with weight 1, 2 and 3 receptively share a Hammr cluster of 1 launcher. The launcher has 6 slots. Each pool runs an identical word count MapReduce program with 6 mappers and 1 reducer. We submit jobs in pool A and B first and after a short time, submit the job in pool C. The result is presented in Fig 5.

7.4 Locality Aware Scheduling

To evaluate the locality aware scheduling, we use the same word count MapReduce program generated by the Hammr MapReduce Interface described above, with an input file of 14 gigabytes. We've also run the

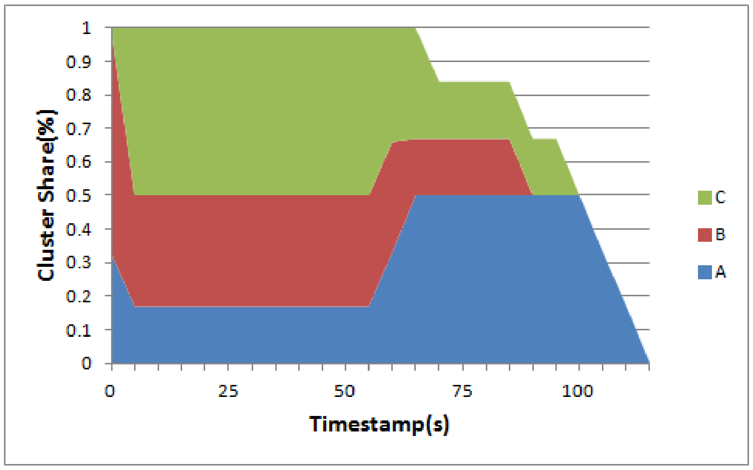

Figure 5: The stacked shares of the three pools. Pool C enters the cluster at time 5, causing preemption of tasks in both Pool A and Pool B. During time 5 and 55, each Pool is given its fair share. At time 55, Pool B and C both have got only 2 mappers to run, therefore, Pool A gets 1 spare slot. From this point, the demands of Pool B and C remain lower than their fair share and therefore, the spare slots are given to Pool A.

	Local Mappers	Total Running Time
Hammr with Locality Aware Scheduling	20	8m21s
Hammr without Locality Aware Scheduling	6.3	9m14s
Hadoop with Fair Scheduler	20.6	9m30s

Table 1: The number of local mappers and total running time of the word count MapReduce program.

same program on a Hammr system without locality aware scheduling (we still use the same minimum flow based algorithm, but all edge costs are set to 0) and a word count Hadoop program with Hadoop Fair Scheduler. All the word count MapReduce programs consist of 26 mappers, each of them takes an input of 512m (one data block in Hadoop Distributed File System) and 8 reducers. Recall that we set the replication factor of Hadoop Distributed File System to 1, each block resides on only one machine. If a mapper is assigned to the same machine holding that data block, we say that this mapper is a local mapper. Since the input of reducers consists of 26 data blocks, one for each mapper, distributed across the cluster, the locality of reducers doesn't matter. Therefore, we only measure the number of local mappers in our evaluation. The result is presented in Table 7.4 and is the average of three executions

The number of local mappers of Hammr with Locality Aware Scheduling is much higher than Hammr without Locality Aware Scheduling and is about 10% faster, proving the correctness of our algorithm. Also in general, Hammr is a little bit faster than Hadoop. This is reasonable because Hammr system doesn't do the monitoring and other work done in Hadoop.

8 Conclusion

Implementing the Fair Scheduler and Locality Aware Scheduler is an interesting experience. Though similar works has been done on Hadoop, the more complicated computation model of Hammr brings more challenge and therefore is more interesting. We have learned how to break things into small components to provide a clean and flexible structure in the hierarchical scheduler and seen how a classic algorithmic problem minimum cost flow can be used to solve a real problem. We have also seen how the complication of the system could bring difficulty to the design the subsystems that depends on it. If it were not for the constraints brought by different channel types, locality aware scheduling would have been much easier as is implemented in Hadoop Fair Scheduler. However, we still believe the more complex computation model of Hammr could bring benefits to certain applications and therefore the Hammr system remains to be an interesting research topic.

References

[1] Apache hadoop. `hadoop.apache.org`.

[2] Fair scheduler. `hadoop.apache.org/common/docs/current/fair_scheduler.html`.

[3] Jeffrey Dean. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[4] Michael Isard. Dryad: distributed data-parallel programs from sequential building blocks. *EuroSys*, 2007.

[5] Michael Isard. Quincy: fair scheduling for distributed computing clusters. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.