FlowScheduler: Towards Efficient Task Assignment for Hadoop

Girish Sastry

May 2, 2011

Abstract

In recent years, the MapReduce programming model has emerged as the leading framework for developing large-scale, data intensive, applications such as collective intelligence, web indexing, and scientific experimentation. Hadoop is the open source implementation of MapReduce, and is a key piece of infrastructure at many institutions. The performance of Hadoop is linked with its scheduler, which reactively assigns tasks to servers for computation with a greedy Round Robin algorithm. We implement and benchmark the proposed FlowScheduler algorithm (by Fischer et al.) for task assignment in Hadoop. While some algorithms are faster in theory but not in practice, we show that FlowScheduler consistently provides better data-locality than the Round Robin Algorithm. We also demonstrate average case analyses of FlowScheduler's performance and provide optimizations and analyses for the algorithm for real use cases.

1 Introduction

Today's consumer web applications such as Google, Facebook, Yahoo, and Twitter, are a mainstay on the web. As more and more users actively participate and contribute to the information on these sites, the volume of data that has to be warehoused, indexed, and searched as grown considerably. As such, the need for a parallel processing framework gave birth to the MapReduce programming model at Google and it's open source implementation in Hadoop. MapReduce allows companies such as Google to process over 20 petabytes of data daily[6]. Hadoop and it's counterparts are used at Yahoo to process data on over 10000 cores[5]. The abstraction and simplicity of MapReduce allows programmers to focus on writing programs and leave fault tolerance and parallelism to the framework.

1.1 MapReduce Framework

Hadoop-MapReduce runs on top of the Hadoop Distributed File System (HDFS). HDFS runs on networked commodity level computers. The data stored in HDFS are replicated across nodes to provide fault tolerance. The block size used in HDFS is typically 64 MB. When a MapReduce computation is run on HDFS, it falls prey to network latency. A data block being accessed from far across the cluster will take longer to process. Thus, one of the goals of MapReduce is to carry out the computation as close to the data as possible[6].

In MapReduce, a user's application is specified by a *job*. The input data is split into independent blocks. These blocks correspond to *map* tasks, which process the data in parallel. Each map task applies a *mapper* user defined function to an input record, which is a *(key,value)* pair. Each map task then spits out a list of intermediate (key,value) pairs which are sorted by the MapReduce framework. Next, *reduce* tasks operate on these intermediate lists of (key,value) pairs. These reduce tasks apply the user defined *reducer* function to aggregate the list of (key,value) pairs into an output value, which is written to HDFS. The mapper and reducer are analogs of *map* and *reduce* (or *fold*) found in functional programming languages.

The cluster in this framework consists of one *master* and a number of *workers*. The master runs a process called the *JobTracker*, which is responsible for scheduling and task assignment. Each worker runs a process called *TaskTracker*, which is responsible for scheduling and executing the tasks assigned to that worker.

Map task assignment is an important piece of the framework that directly impacts job execution time. Reduce tasks cannot be executed until the map tasks have finished, and task assignment determines the location of the intermediate data for the reduce tasks[6].

1.2 Previous Work

Fischer et al. proposed a theoretical model of MapReduce and a FlowScheduler algorithm which computes the optimal assignment within an additive constant[8]. Dhok et al. proposed a machine learning scheduler for task assignment, however, the feasibility of this is unclear for efficient data-locality[7].

1.3 Our Contributions

We extend the theoretical work by Fischer et al[8]. Their work assumes a simplified, theoretical framework of MapReduce and makes several assumptions for theoretical rigor. In reality, systems cannot be expected to fully conform to their theoretical counterparts. Thus, we implement the FlowScheduler algorithm proposed by Fischer et al. and carry out analyses on average use cases for the scheduler. Some algorithms are fast in theory but slower than expected in practice – a prime example being Radix Sort [5]. We show that the FlowScheduler provides better data-locality than the current Hadoop Round Robin Scheduler. Finally, we make several notes on the aspects of a real world implementation of this algorithm and how it can be tuned to optimize MapReduce performance.

2 Task Assignment in Hadoop

Each Hadoop job consists of a set of map and reduce *tasks*. The Hadoop *cluster* is a set of *nodes*, each of which has some number of *slots* to run tasks. The default number of slots per node is two[2].

2.1 Round Robin Algorithm

The Round Robin Algorithm is a form of Round Robin Scheduling that is commonly used for process scheduling in many systems.The Hadoop scheduler works

by incrementally assigning tasks in a greedy fashion. Each time a heartbeat from a TaskTracker is received, the JobTracker checks three conditions, in order:

1. If there is an unassigned task that is data-local to the TaskTracker, then assign it to the TaskTracker
2. Else if there is an unassigned task that is rack-local to the TaskTracker, then assign it to the TaskTracker
3. Else, choose the closest remote task to assign to the TaskTracker

In analyzing this algorithm, Fischer et al. showed two interesting lemmas:

- Assuming that remote tasks are higher latency than rack-local or data-local tasks, then increasing the number of data-block replicas may increase the maximum load from an assignment from the Round Robin Algorithm
- The load of a Round Robin assignment is at most $\frac{cost_{remote}}{cost_{local}}$ as good as the optimal load

These lemmas reveal that remote task assignments are very detrimental to the performance of the Round Robin Algorithm. This implies that the assignment computed by the Round Robin Algorithm is dependent on the network topology of the cluster, and the replication factor (and thus fault tolerance) of the Hadoop system. The Round Robin algorithm is unlikely to compute the optimal assignment for larger networks[8].

2.2 FlowScheduling Algorithm

Formally, the input to a task scheduler in a MapReduce system can be thought of as having:

- A set T of tasks,
- A set S of servers (TaskTrackers),
- A placement graph, ρ, dictating the data placements of tasks to servers,
- Costs for data-local, rack-local, and remote-local tasks (w_{data}, w_{rack},w_{remote})

If there were no such thing as remote tasks, this would simply be a case of bipartite matching. If we use bipartite matching to match as many local tasks with servers as we can, how do we deal with the remaining unassigned tasks? We can greedily assign the remaining tasks based on each server's load. This is the core idea behind the FlowScheduling algorithm proposed by Fischer et. al. They proposed to use a network flow algorithm for matching followed by a greedy algorithm to assign the remaining tasks.

The pseudocode for FlowScheduler is as follows:

Algorithm 2 A flow-based algorithm for HTA.

input: an HMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$
define A, B as assignments
define α as a partial assignment
$\alpha(t) = \perp$ (task t is unassigned) for all t
for $\tau = 1$ to m **do**
    $\alpha \leftarrow$ max-cover(G_ρ, τ, α)
    $B \leftarrow$ bal-assign$(T, S, \alpha, w_{\text{loc}}, w_{\text{rem}}(\cdot))$
end for
set A equal to a B with least maximum load
output: assignment A

MaxCover works by matching at most τ data-local tasks with servers at each iteration. At the core of this function is the augmenting path algorithm by Ford-Fulkerson. The input placement graph G_p is converted to a flow network, G'_p with an extra source u and sink node t. In G'_p, there is an edge (u, t) for all $t \in T$ and an edge (s, v) for all $s \in S$. Every edge in G_p remains in G'_p. All edges (s, v) have capacity τ for all $s \in S$, and every other edge has capacity 1. Thus, for any (u, v) flow, the maximum flow is 1. The previously computed partial assignment, α, is converted to a flow network. If a particular task t is assigned to server s, then flow one unit through the path u, t, s, v. Next, the Ford-Fulkerson algorithm is run on G'_p to find a matching. The output flow network is converted back into a partial assignment.

After the maxCover phase, balAssign greedily assigns the remaining unassigned tasks for that iteration. It simply chooses the server with minimum virtual load and assigns the next unassigned task to that server. Here, virtual load is defined as the sum of all costs of tasks assigned to that server.

There are τ full assignments computed during the course of the algorithm, and the one with minimum virtual load is selected as the task assignment[8].

3 Implementation & Experimental Methods

When a job is submitted to Hadoop, it is decomposed into a set of map and reduce tasks. The scheduler then assigns these tasks to slots, which run the tasks. The design of the Hadoop scheduler is centered around a three classes: `JobTracker`, `JobInProgress`, and `TaskScheduler`.

3.1 Scheduling Design

Every scheduler in Hadoop is required to inherit the `TaskScheduler` abstract class. The `TaskScheduler` provides access to the `TaskManager`, which is an interface to the `JobTracker`, and an instance of a job `Configuration`. There are three methods that the scheduler is required to implement from the `TaskScheduler` class: the lifecycle methods `start` and `terminate`, and `assignTasks`, which launches tasks on a given TaskTracker.

TaskAssignment in Hadoop is reactive. The periodic heartbeats from a TaskTracker contain metadata on the TaskTracker – list of running tasks, task statistics, number of slots, and other information. The heartbeat response from the JobTracker calls `assignTasks`, which selects a new task to add to the TaskTracker.

The implementation of `assignTasks` in Hadoop calls two methods from within `JobInProgress`: `obtainNewMapTask` and `obtainNewReduceTask`. Both methods return either a `Task` to run, or `null` if no such task is available. `JobInProgress.obtainNewMapTask` implements Round Robin Scheduling, and each time it is called, attempts to select the "closest" map task to a node. Here, closest is defined by the minimum physical distance between the data and the node. In Hadoop, there are three levels of distance and locality: data-local, rack-local, and remote. Some remote tasks may be closer than others, depending on the network topology. More concretely, `JobInProgress.obtainNewMapTask` queries some caches that were set up in the initialization of the job. These caches contain local maps, nonlocal maps, and remote maps, and `obtainNewMapTask` continues onto the next cache if there is a cache miss. This implementation allows the Round Robin Scheduling algorithm to actually assign tasks very fast in time linear to the number of tasks and servers[2].

3.2 FlowScheduling Implementation

Because task assignment in Hadoop is reactive, the design of the scheduler and classes surrounding it reflect this "on-line" nature. The method `assignTasks` is only called during the heartbeat mechanism between a TaskTracker and the JobTracker. This philosophy in design clashed with the assumptions in the theoretical model of the FlowScheduler algorithm, and initially led to two distinct avenues of thought: one implementation that is on-line and reactive, and one implementation that is "off-line" and preassigns tasks. With some consideration, we observed that these two versions are one and the same. An off-line scheduler that preassigns tasks in the initialization of the job allows the heartbeat mechanism to simply query the table of assigned tasks to select the next task. Thus, we design the FlowScheduler implementation by preassigning tasks. During job intialization, the `JobInProgress` computes the locality graph of tasks to nodes as the input splits are computed. The `JobTracker` maintains the lookup table of the assignment (and many intermediate lookup tables). The table of task assignments must be able to be atomically accessed and changed by only one TaskTracker at a time, or else we risk falling prey to concurrency issues.

There are two distinct parts to the FlowScheduling algorithm: `maxCover` and `balAssign`. For the FlowScheduler implementation, we introduce a new data type in the Hadoop class ecosystem called "TrackerTasks" which represents a tracker, it's assigned tasks, and its load. In our case, the load is not the server load, but the virtual load, which is defined previously.

`balAssign` was implemented using priority queues that operate on TrackerTasks. We wrote a `comparator` for the `TrackerTasks` class that simply compares on the value of the load. Each iteration of balAssign calls the `poll` method of the priority queue to select the `TrackerTasks` with least virtual load and assigns the next unassigned task to that TaskTracker. The input remaining map tasks to `balAssign` are randomized on each call. Because a greedy algorithm such as `balAssign` is succeptible to the ordering of its inputs, this randomization both reduces the chance of a worst case scenario increases the chance of a much better assignment.

The core of `maxCover` was implemented using the Edmonds-Karp maximum flow algorithm in a separate class called `FlowNetwork`. The `maxFlow` method in `FlowNetwork` returns a flow matrix, which contains the flow values for every edge

in the input graph. We added several functions to `FlowNetwork` to implement the "shortcut" in iteratively assigning tasks using maxCover – each iteration should use the partial assignment from the previous iteration as a jump start in computing a flow. `FlowNetwork` also contains functions to convert from a partial assignment to a flow matrix and vice versa. Note that the Edmonds-Karp maximum flow algorithm is $O(|V|*|E|^2)$, where V is the number of vertices in the graph and E is the number of edges. This is asymptotically slower than other solutions to the maximum flow problem, but we choose Edmonds-Karp for two reasons. First, there are abundant resources for coding the algorithm and it is relatively simple to read and understand. Second, Edmonds-Karp often performs much faster for sparse graphs, and our locality graph is sparse (the number of edges for each task is bounded a constant, by default 3).

Because the original design of Hadoop task assignment assumes reactive task assignment, there were a number of challenges in implementation. Much of the communication between `JobInProgress` and `JobTracker` had to be written to process the data types present in `JobInProgress` to "nice" data types for the `JobTracker` and its lookup tables. This caused some overhead and complicated the implementation, but metrics show that this processing did not significantly contribute to the runtime of `maxCover`.

3.3 Experimental Methods

In order to evaluate the performance of the FlowScheduler, we choose several jobs that display different workloads and have previously been used to evaluate the performance of MapReduce systems. We run the jobs `pi`, `sort`, `wordcount`, and `grep` with randomly generated datasets of size 20GB and 50GB from `randomwriter` and `randomtextwriter`.

`randomwriter` is a job that writes randomly generated bytes to a HDFS. Similarly, `randomtextwriter` writes randomly generated words from a corpus to HDFS. `pi` runs a Quasi Monte-Carlo simulation to estimate the value of π. It writes its own input data to HDFS during the initialization of the job. `sort` simply sorts bytes of data. `wordcount` and `grep` count words and find patterns, respectively, in the input data. These jobs represent different classes of the typical workload for MapReduce systems. `pi` is a computationally expensive, map intensive job. `wordcount` is a I/O intensive and reduce intesive job. `grep` is a map intensive and I/O intensive job. Finally, `sort` is the canonical benchmark for MapReduce systems, and has been used by Yahoo, Google, and Facebook to measure performance[11].

FlowScheduler was evaluated on Amazon's Elastic Compute Cloud using `m1.small` instances. These are 32-bit virtual machines with 1.7 GB of RAM and 160GB storage[1]. Both the tests for FlowScheduler and Round Robin were run on the same Amazon instance group. In initial trials, we observed large variance in the data-local and rack-local map times. To combat this, we first run three "warm-up" jobs before running and recording six trials for a particular test. This helped reduce the effects of heterogeneity in the Amazon EC2 environment. The Round Robin algorithm in Hadoop was run with speculative execution off, as this was not implemented for FlowScheduler. Furthermore, speculative execution can both increase and decrease performance[11], so we did not use it to reduce variance in the data.

In all tests, we used the default configuration for Hadoop and HDFS. Each

machine by default can run up to two mappers and two reducers simultaneously. We chose input data size and cluster size to simulate a wide variety of jobs and job times. Most of our job times run in under 5 minutes, which simulates the shorter batch jobs commonly run with MapReduce. We also run some longer jobs that take over one hour to complete.

For each trial, we track the job completion time, number of maps, number of data-local maps, number of rack-local maps, 80% map completion time, 100% map completion time, scheduler run time, average data-local map time, average rack-local map time, and normalized job time. Normalized job time is a metric we introduce to compare performance in a theoretically homogeneous cluster. We normalize the job time by computing the mean of the data-local map times and the mean of the rack-local map times, computing the projected job time based on these values, and adding the time for scheduling.

4 Results

4.1 FlowScheduler Runtime

For the first set of experiments, we attempt to verify the the runtime bound on the FlowScheduling algorithm. We run the example job `pi` from the prepackaged Hadoop examples jar. The advantage of the `pi` job for this experiment is twofold: it is map-intensive and a relatively fast job to run. We run 6 trials each and plot the means of these trials on a 20-node Amazon EC2 cluster. We scale up from 100 maps to 800 maps. The results are as follows (fig. 1):

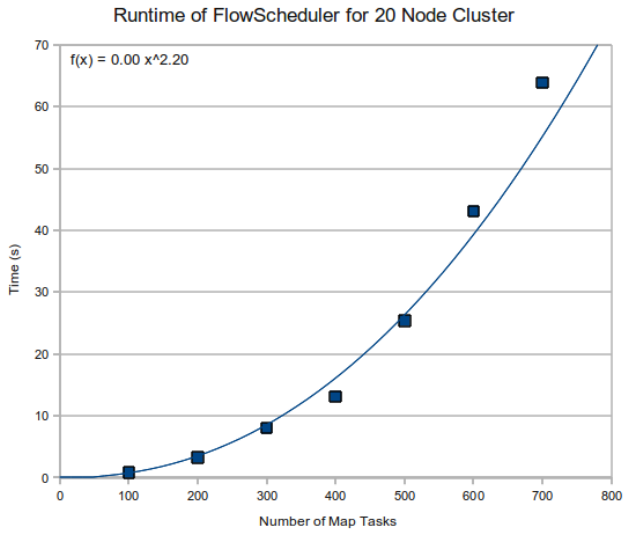

Figure 1: FlowScheduler run times

The augmenting path algorithm that is at the core of maxCover phase of the FlowScheduler algorithm runs in time $O(m^2n)$, where m is the number of tasks and n is the number of servers. However, it was conjectured that it would

actually run in time $O(m(m+n))$. This is because the placement graph is sparse. The replication factor that bounds the number of edges in the graph is by default three, and is a constant. Our experiment with the `pi` task shows that the runtime does indeed grow polynomially. The implication of this is that the maxCover phase is the primary determiner of the runtime of the FlowScheduler. This also shows that the runtime of the scheduler is heavily dependent on the number of map tasks for the job. A job can be very large and CPU intensive but have a small number of maps – in this case, the scheduler would perform very well. The flip side of this is that a job with an extremely high number of map tasks will take a time polynomial in the number of maps to schedule.

4.2 FlowScheduler Performance

Next, we run experiments to evaluate the performance of FlowScheduler vs the regular Round Robin Scheduler in Hadoop. We run the `randomwriter` and `randomtextwriter` jobs to produce 20GB and 50GB data sets, which were stored on Amazon S3. The `grep`, `wordcount`, and `sort` jobs were used to benchmark the schedulers. We expect the FlowScheduler to outperform the Round Robin scheduler in terms of efficient data-locality. That is, we expect the ratio of $\frac{number(data-local)}{number(rack-local)}$ to be higher with the FlowScheduler.

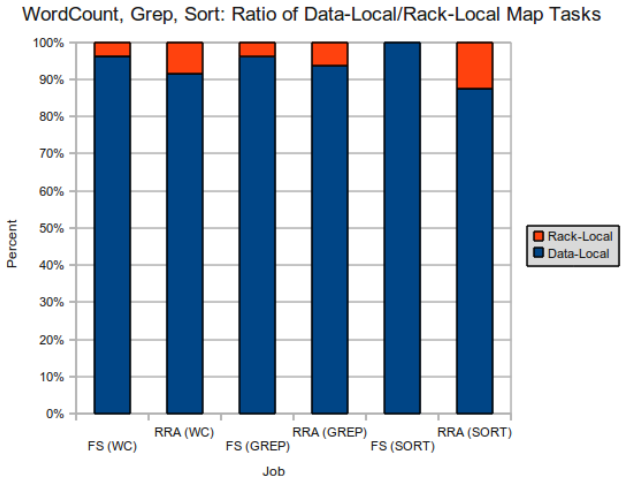

Figure 2: Ratio of Data-Local to Rack-Local tasks across jobs

Our tests show that this indeed is the case. Across the `grep`, `wordcount`, and `sort` jobs, we find a better data-locality in the assignments from the FlowScheduler (fig. 2). FlowScheduler also computes the optimal assignment of all data-local tasks in the case of the `sort` job consistently across trials, while the Hadoop scheduler does not. It is interesting to note that in the very initial trials for the non-optimized implementation of `balAssign`, we received slightly worse results. The improvement from randomizing the input tasks to `balAssign` helped with this. We expect that in extreme cases, where perhaps it is very difficult to find a data-local task to assign in the `maxCover` phase, this input randomization in `balAssign` will provide a much higher chance of landing on a good input order-

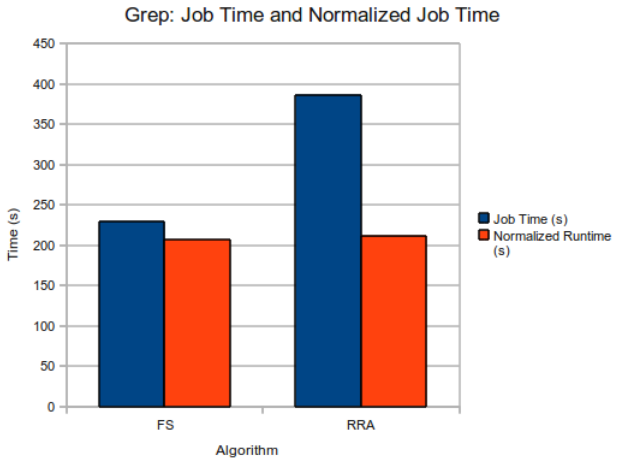

Figure 3: Grep Job times

ing. If there are m map tasks, and there is one optimal ordering for `balAssign`, then there will always be a nonzero probability of landing on this ordering ($\frac{1}{m}$ in this case).

Better data-locality is not the end-all be-all of a scheduler. We are using a relatively expensive algorithm to preassign tasks, and in order to completely evaluate the peformance of the scheduler, we turn to the actual job completion times. In evaluating these jobs on Amazon EC2, we noticed a high variance in the data-local and rack-local map task times. There were cases when rack-local maps would finish faster than data-local maps. This high variance is probably due to the inherent heterogeneity of the EC2 cloud. Even though EC2 uses virtualization to provide instances to a user, these virtual machines still share physical resources. Thus, contention for hardware from requests from other uses can still lead to heterogeneity. Further, we noted that the first few jobs on a cluster were significantly slower then jobs run after that. This is likely due to the disk I/O needing to "warm up" when reading data from HDFS. Thus, for our trials, we chose to discard the first three job times and then keep the next six trials to encourage homogeneity. In order to further demonstrate a comparison between the FlowScheduler and Round Robin Scheduler, we introduce the metric of normalized job time, as discussed earlier. This helps effectively compare the actual job performance due to scheduling. The results for `grep`, `wordcount`, and `sort` are reported in figures 3-5.

These jobs were run on clusters of 20 nodes. Because each node has 1.7GB of memory, we used datasets of 20GB and 50GB to ensure that the I/O was happening from disk and not from memory. We see that the FlowScheduler outperforms the regular Hadoop Round Robin Scheduler for these tests – in both the job run time and normalized job time.

While most MapReduce jobs are small, $\leq$ 5 minute jobs, we also ran larger jobs for more than an hour to investigate the effectiveness of the scheduler for larger jobs. In this case we ran one test, and we noted better locality and faster job time. We did find that variance due to EC2 heterogeneity threw off some intial results for these larger jobs.

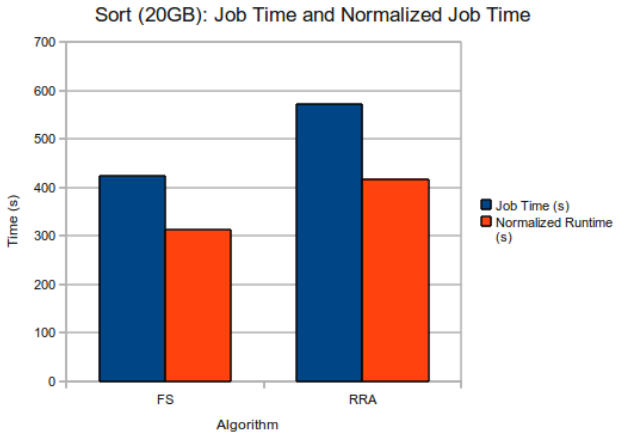

Figure 4: Sort Job times

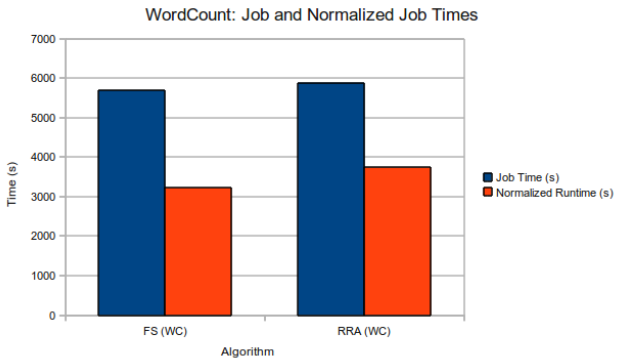

Figure 5: WordCount Job times

Finally, we turned to optimizing the cost functions for data-local (w_{data}) and remote-local tasks (w_{rack}). The default values in FlowScheduler are 1 and 2, respectively. While these are arbitrary values, we found that they are pretty good estimates of the actual differences in rack-local and data-local tasks on EC2. For the `sort` job, we calculated $w_{data} = 1$ and $w_{rack} = 1.7$. Here we simply used the average data-local and average rack-local map task times. However, tweaking these parameters for `sort` did not provide a significant increase in the locality or a decrease in job completion time. We expect that for clusters where $\frac{w_{data}}{w_{rack}}$ differs significantly from the default value of 2 to be much more susceptible to this tuning parameter.

It is interesting to note the 80% and 100% map completion times (fig. 6). For the longer `wordcount` job, we observed a pitfall of the current implementation of FlowScheduler. The long tail on the 100% map time was due to straggling tasks that took abnormally long. This could be chalked to up to either EC2 cluster heterogeity or the fact that speculative execution was not implemented.

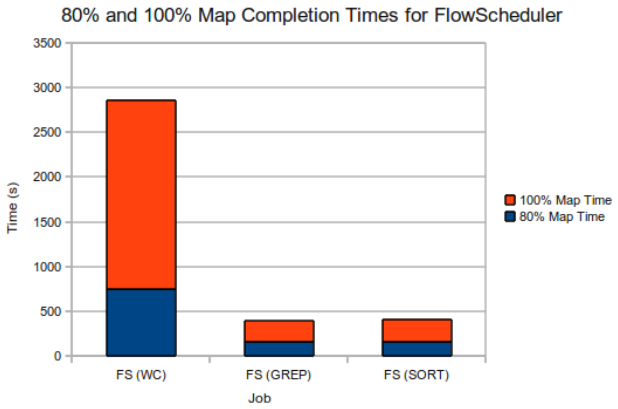

Figure 6: Long Tail: 80% and 100% map completion times

5 Discussion

For our tests, which were selected to benchmark the schedulers in manner analogous to normal use cases, the FlowScheduler outperformed the Round Robin Scheduler. While the effects of locality were better across the board, they were not necessarily as pronounced as they could be. For example, because of the constraints of the EC2 cluster size, we were unable to run any jobs that contained remote tasks. The cluster on EC2 were all on the same rack, so the biggest different in locality was between data-local and rack-local tasks. Remote tasks can be much, much more costly than rack-local tasks. We expect the FlowScheduler to provide even better results than what we observed here when a large cluster with remote tasks is involved.

Scaling the FlowScheduler to a larger scheduler will likely provide better locality, but may suffer on the scheduling runtime. The FlowScheduler runs in time polynomial to the number of map tasks. Thus, it may not scale well for a job that has many, many maps. In real world cases, too many maps equates to degradation of cluster resources, and too few maps means not taking advantage of MapReduce's inherent parallelism. For the simulations we ran, there were never any jobs that had too many maps such that the FlowScheduler's scheduling time was a real issue. However, just as an example, calculating π with the `pi` job and 800 maps takes over a minute. The Round Robin scheduler outperforms in this case.

There are certain random cases when the Round Robin Scheduler will provide a very bad assignment – this is more likely with remote tasks. In the implementation of the Hadoop scheduler, there are several heuristics to battle starvation and bad assignments in general: speculative execution, LATE scheduling, and load rebalancing. These have not been implemented for the FlowScheduler. Thus, the current implementation of the FlowScheduler is not fault tolerant and does not respond well to TaskTrackers leaving or entering the cluster. One quick fix would be to revert to the Round Robin Algorithm to assign tasks when TaskTrackers enter or leave. Another option would be to do a complete rebalancing.

There is a lot of tweaking and tuning involved to gain the most out of the

FlowScheduler. This tweaking was not done or implemented fully for these experiments, but here is a list of potential optimizations based on cluster statistics and network topology:

- Certain Network Flow algorithms are better for certain network topologies. Choosing the right one on a per-cluster basis may be important.
- In the general case, if we want to run jobs with many, many maps such that Ford-Fulkerson becomes too much of a time sink, then we should consider using a faster network flow algorithm, or an approximation algorithm for network flow which would provide a boost in speed.
- Computing w_{data}, w_{rack}, and w_{remote} for our cluster could provide some benefits in `balAssign`
- Dealing with straggling tasks: one way is to implement speculative execution, but speculative execution is not always the right way to improve performance in a MapReduce system[11]. We propose "speculative reassignment". If there are spare CPU resources, we may be able to calculate other assignments using FlowScheduler in parallel with the running system. Then we can reassign the remaining maps for a better assignment.
- Responding to variances in a hetereogeneous environment. The current Hadoop scheduler has heuristics to perform better in a heterogeneous environment, such as LATE scheduling[11]. Implementing this would help the FlowScheduler.

6 Conclusions

The FlowScheduler does provide better locality than the Hadoop Round Robin Scheduler. For jobs with a high cost of data access in rack and remote, the FlowScheduler is a better option. For jobs with few map tasks, the FlowScheduler is also a better option. Indeed, somewhat unexpectedly, larger jobs that take a long time to run may need fewer maps. For example, Kambatla et. al find that 8 maps is optimal for a 80GB Grep job[10].

The network flow module is a pluggable component, so using a different, faster network flow algorithm is next on the agenda. This should allay some of the concerns about scaling to a high number of map tasks. Getting the most out of the FlowScheduler requires tuning to a specific cluster environment. We hope to extend this project to provide tweakable options for the network flow algorithm based on network topology and job parameters, options for w_{data}, w_{local}, and w_{remote}, and options for load rebalancing and speculative execution. Making the FlowScheduler interoperable with larger scale job schedulers such as Yahoo's CapacityScheduler or Facebook's FairScheduler is also an important step in making the FlowScheduler viable for production use.

7 Acknowledgements

We would sincerely like to thank Dan Abadi for his invaluable advice and help. We would also like to thank Xueyuan Su, who provided many hours of discussion and advice, and whose work this project is based on. We are also very

grateful to Kamil Bajda-Pawlikowski, who was instrumental in helping with implementation details and general project advice. Finally, we would like to thank Stan Eisenstat and the entire Yale Computer Science department for this opportunity. This work was partially sponsored by funds from the Yale Computer Science department.

References

[1] Amazon ec2 instance types, http://aws.amazon.com/ec2/instance-types/.

[2] Apache hadoop project, http://hadoop.apache.org/mapreduce/.

[3] Applications powered by hadoop, http://wiki.apache.org/hadoop/poweredby.

[4] Yahoo! uses largest production hadoop cluster, http://yhoo.it/fvy12n.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms, second edition, 2001.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[7] Jaideep Dhok and Vasudeva Varma. Using pattern classification for task assignment in mapreduce.

[8] Michael J. Fischer, Xueyuan Su, and Yitong Yin. Assigning tasks for efficiency in hadoop: extended abstract. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 30–39, New York, NY, USA, 2010. ACM.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37:29–43, October 2003.

[10] Karthik Kambatla, Abhinav Pathak, and Himabindu Pucha. Slides: Towards optimizing hadoop provisioning in the cloud.

[11] Matei Zaharia, Andrew Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. Technical Report UCB/EECS-2008-99, EECS Department, University of California, Berkeley, Aug 2008.