Cooperative Computing: Exploiting Physical Proximity in MapReduce

Pratima Kolan, Fatemah Panahi, Rich Joiner

May 9, 2011

Abstract

In this paper we introduce Cooperative Computing. Cooperative Computing exploits the physical proximity of nodes in distributed systems. It enables nearby nodes to cooperate with each other in a controlled manner when a node gets temporarily overloaded. We envision Cooperative Computing to be applicable to MapReduce and Consistent Hashing. We attempt to incorporate Cooperative Computing to the Hadoop MapReduce framework, which results in encouraging performance improvement in job completion times in certain scenarios.

1 Introduction

We start by defining *Physical Proximity.* Physical proximity of 2 computing nodes/servers has different meanings in different contexts:

- The nodes are located in the same rack in the data center
- The nodes can communicate with each other with a latency less than some threshold

Normally in distributed systems, two replicas of the same content, or 2 servers serving the same content, are not placed physically close to each other. The reason is that this will affect reliability and/or availability of the system as follows:

- Consider the first meaning of the physical proximity where 2 nodes are located in the same rack. This means that if one node goes down in a data center, it is highly probable that the other replica of the content in the same rack will go down. Such correlated failures may happen when a data center shuts down or some rack stops working because of a problem in power supply. The recent Amazon EC2 outage is an example of correlated failures [14].
- Consider the second meaning of physical proximity where the 2 servers can communicate with low latency. This means that 2 servers serving the same content are placed in the same geographic region. This defeats load balancing purposes.

Thus, given that the 2 replicas or the 2 servers are normally placed far from each other, it is costly (in terms of bandwidth or latency) for them to communicate or cooperate with each other. Cooperative Computing attempts to leverage the physical proximity of nodes to each other in *transient* states in the system. In the transient state of overload or stress in the system we try to get help from nearby nodes, which is faster and less expensive in terms of bandwidth/latency. Helping another node means sharing computational resources with that node so that it can finish its task more quickly. Since this optimization comes into play only in transient states, we do not anticipate Cooperative Computing to reduce reliability/availability guarantees that current distributed systems are built upon.

Cooperative Computing can be incorporated with low overhead to distributed systems in which the nodes already maintain some topology and connectivity information about other nodes. The MapReduce computing clusters, and Consistent Hashing mechanisms are 2 examples of where Cooperative Computing can be implemented.

We have incorporated Cooperative Computing to the Hadoop MapReduce framework. Our experimental results show improvements over the state of the art when dealing with large data sets. In many cases this approach has better or the same performance as the original Hadoop MapReduce framework, and provides the same correctness guarantees. Given the size of MapReduce jobs and the large data sets that they work with, we believe that even a few percent improvement in total job running time can have significant impact and save millions of dollars a year [6].

In summary our contributions are first, improving the state of the art approaches for mitigating slow/unresponsive nodes in the distributed system by getting help from nearby nodes. Second, we prototype Cooperative Computing in the Hadoop MapReduce framework. And third, we show that this approach with almost no overhead can result in improvements in job completion time in certain scenarios, and is as good as the state of the art in other scenarios.

The rest of the paper is organized as follows. We first describe applications of Cooperative Computing. Next, we introduce the related works. Then, we discuss our implementation details. Following that we explain our experimental methodology. We then present our results, discuss future work, and conclude the paper.

2 Applications of Cooperative Computing

Cooperative Computing can be added with low overhead to distributed systems that keep and update topology and individual node health information about the system in some form. We believe that this approach will be a natural fit for MapReduce and Consistent Hashing.

2.1 MapReduce

Outliers in MapReduce jobs are common. Outliers are tasks in MapReduce phases that keep the job from completing. As you can see in figure 1 , most of the tasks in the reduce phase finish quickly, whereas a few of the tasks take a long time to finish. In [6] , authors show that 25% of MapReduce phases have more than 15% of their tasks as outliers. Moreover, 80% of the run-time outliers (with uniform probability) are delayed by between 1.5x to 2.5x. Given that a MapReduce job is not finished until *all* its tasks are finished, the outliers can dramatically reduce the cluster performance.

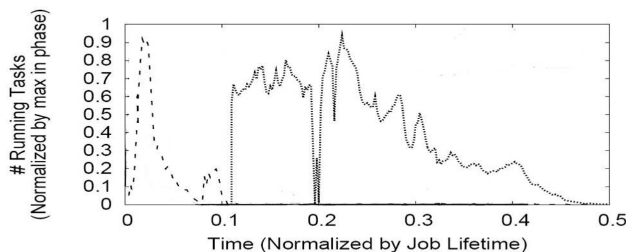

Figure 1: Outliers in MapReduce can be seen in both Map (first curve) and Reduce (second curve) phases.

Currently in the Hadoop MapReduce framework, once an outlier is detected, the task is replicated in another node, in the hope that the replicated task will finish quicker. In [6], authors have

made some improvements to this naive approach such that restarts are only done when they are effective. For example, if a node is slow only because it has more data to process, replicating the task will be a waste of resources.

Cooperative Computing takes a different approach in mitigating the outlier problem. Instead of completely replicating the task at another node in the system, we split the task into sub-tasks that can be completed independently. This has 2 effects: First, it will reduce the load on the stressed/overloaded node so that it can finish its part more quickly. Second, the task is split into sub-parts that can be run in parallel and thus the task will finish faster.

The policy for how the data can be split should be different per application/job. Moreover, only the nodes that are physically proximal to the original node (i.e. in the same rack) are candidates for helping the stressed nodes. This is important because the Hadoop framework schedules tasks near their data to reduce bandwidth cost for transferring the data to the node. Thus, it is likely that the required data for the helping node is available in the same rack as it resides. This minimizes the overhead of transferring the data, and eliminates the possibility of Cooperative Computing stressing the network bandwidth in the system.

Cooperative Computing can be added to the existing Hadoop/MapReduce framework with adding no overhead. This is because in this framework, the NameNode stores the topology information. Thus, there is no overhead for keeping track of the topology. Also, each TaskTracker periodically sends heartbeats to the JobTracker. This heartbeat includes statistics about the task progress that can be used to find outliers in the system.

2.2 Consistent Hashing

We envision Cooperative Computing to be applicable to *Consistent Hashing*. However, we have not yet prototyped this application.

In consistent hashing, nodes are placed randomly in a ring without regards to physical proximity. The purpose for this randomness is to balance the load on the nodes. When a node fails, its successor node takes care of all the data which should be routed to the failed node until it comes up. And once the node comes up, its data is shifted back to it from its successor node. Sometimes this successor node may be physically distant and transferring this data can take time and substantial network resources.

In the context of Cooperative Computing once a slow/overloaded node is detected, it can get help from the nearby nodes. If there is any node physically proximal to the slow node which has sufficient resources, then it can *temporarily* assume some of the data of the slow node and serve requests for that data until the stress from the slow node is removed. By reducing the load on the slow node, first, requests can be responded faster. And second, we hope to prevent a complete failure of this node, in which case significant network bandwidth is wasted to transfer the data to the (distant) successor node and transferring it back to the primary node after it comes up.

In many systems that use Consistent Hashing, such as Amazon's Dynamo [13] , every node already keeps some reachability information about every other node in the ring. Thus, there should be minimal extra bookkeeping work to be done to incorporate Cooperative Computing to these systems.

3 Related Work

The motivation for our work was the concept of Wireless Grid and Ubiquitous Computing [5]. In this environment, wireless devices form an ad hoc grid and they can share different resources such as the computing resources between them. This saves the bandwidth to offload the computation to the cloud or to the service provider. What we propose is a similar concept but applied to MapReduce

or Consistent Hashing cases. We want to enable nearby nodes to cooperate with each other in an ad hoc manner when deemed appropriate by our protocol (ex. when a node gets temporarily overloaded).

The closest work in the literature to our proposal is [6]. In this paper, the authors try to identify outliers and their cause and take actions correspondingly. They identify three causes of outliers as machine characteristics, network characteristics, and imbalance in work of different tasks. They try to deal with outliers based on the cause of slowness. Tasks are restarted in cases where the issue is not workload imbalance. They also propose network aware placement of reduce jobs to avoid congestion. Also, in cases where there is a probability that the output of a map job will be lost (ex. in case of a failure in a nearby node), they try to protect (replicate) the output. Cooperative Computing builds on this work and introduces a different approach of dealing with outliers where we split the task and get help from nearby nodes to finish the task more quickly.

Mariposa [12] is a distributed database system. It implements an economic paradigm where different servers bid for serving parts of a query. The server with the lowest bid will win and server the query. Mariposa is designed for systems that span Wide Area Networks(WAN), and have significantly different constraints from a system that spans a Local Area Network(LAN). Mariposa is similar to Hadoop in that the data and query can be moved in the system in the system in order to provide lower cost alternatives for completing the queries. We view Mariposa a platform similar to Hadoop where our optimization might provide improvements in job completion times. One can split the queries into smaller sub-queries once a server is overloaded to reduce stress on the system and get faster response.

In [8], proximity information is used in order to have a better load balancing scheme. However, the goal of our approach is to offload computation dynamically when the system is overloaded, rather than using proximity information for distributing load.

In *Pastry* [7] and *PON* [9], the researchers try to alleviate the routing latency that is observed in overlay networks because of physical distance of the nodes. Therefore, they make use of physical proximity information and try to design an access scheme that is efficient with regard to both time and space. However, what we propose is to use the topology information only in transient states and in an ad hoc manner in the system.

4 Implementation

We have implemented Cooperative Computing in the form of a new paradigm of speculative task reassignment in the Hadoop MapReduce framework. Figure 2 shows an *abstract* view of our protocol. At issue is the methodology of re-balancing a workload that is observed to be skewed with respect to the amount of work that remains undone across the worker nodes for a MapReduce job. The standard Hadoop approach is to wholly replicate a particular task that is determined to be slow onto another node which appears to have relatively free resources. While our enhancement is designed to divide the work of the slow node rather than duplicating it so that the progress up to that point in time can be preserved, the complexity of such a change proved to be prohibitive in the time allotted; this is discussed further below. However we did successfully enhance the framework to take advantage of the locality of machines with respect to data center racks which led to some interesting changes to the code, which are discussed now.

Each TaskTracker periodically communicates the status of its constituent tasks via a heartbeat message back to the JobTracker (see figure 3 for definition of terms), which then determines whether and how to reassign any work. In the standard Hadoop workflow, a TaskTracker declares its node to have the capacity to take on more work, at which time the JobTracker scans the current

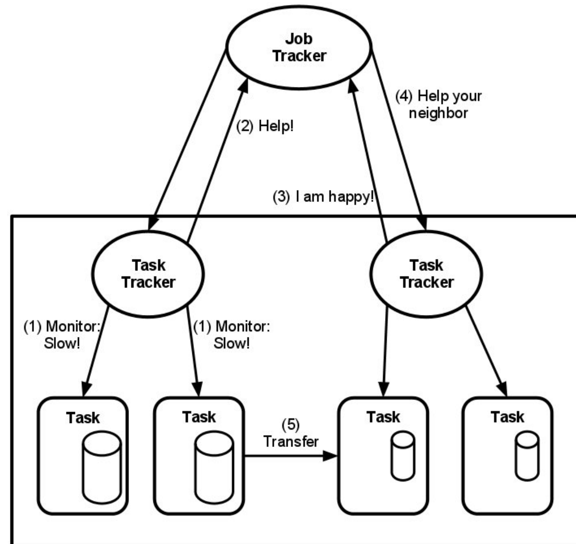

Figure 2: Abstract view of our protocol

MapReduce: a programming paradigm in which a parallelizable workload is distributed amongst a cluster of worker nodes, or the Hadoop module of the same name.
HDFS (or DFS): the Hadoop Distributed File System, which provides data when and where it is needed by MapReduce processes.
Hadoop Common: libraries that are shared by the HDFS and MapReduce portions of Hadoop; previously known as Hadoop Core.
Cluster: the set of machines that are available for acting as HDFS namenodes or datanodes and/or MapReduce masters or workers.
Node: an individual machine in the Hadoop cluster.
NameNode: the coordinating machine for an HDFS system. It controls the metadata for the file system.
DataNode: a machine in the HDFS system containing the file data consisting of a series of named blocks.
JobTracker: a process running on a master node that distributes work and periodically receives updates on the health of the tasks that constitute the currently running jobs.
TaskTracker: a process running on each worker node which monitors and reports on the health of the tasks running on that node. There is typically one task tracker per job executing on each node.
Task: a process that runs on a particular worker (or slave) node. These are categorized into "map" and "reduce" tasks (in addition to the more obscure "combine" and "shuffle" tasks which we do not deal with in this project) according to their function.
Job: a set of map and reduce tasks running within a cluster that can interact and are coordinated to produce an output.
Speculative task: a task which is a replica of one running on a different node that has been determined to be lagging. By replicating the work in multiple places, job completion time will be determined by the quickest of the replicas rather than the slowest of all tasks.

Figure 3: Terminology specific to the the Hadoop system

status information provided by the other TaskTrackers to find a suitable candidate for speculative reassignment. The current implementation of Hadoop considers only the percentage of a particular task that has completed, without regards the any differences that there may be in the size of the workload that is assigned to each task.

Contrastingly in our model, as each heartbeat message is processed the sending TaskTracker is analyzed to determine whether it is running substantially slower than its neighbors (taking into account the amount of data it has left to process and the rate at which it has processed data to this point). If so, the JobTracker actively seeks out a worker node to which a sub-split of the slow node's workload can be reassigned. In contrast to the standard Hadoop logic, we envision the workload being divided between the nodes rather than duplicated. Furthermore, the principal of Cooperative Computing which values locality when transferring data is utilized by considering whether a task is rack-local to the lagging node when determining where to start the speculative task. Since the machine on which the lagging task is present will also serve as a DataNode for the data that is needed to complete that task, performing an in-rack transfer is theorized to be more efficient when possible.

We did encounter roadblocks to our implementation, mostly due to the complexity and interconnectedness of the Hadoop framework. Though great progress has been made towards the splitting of a lagging task (instead of the replication of the entirety of the task), we were unable to complete this portion of the project. Factors which have made this difficult are the limited information which can be communicated natively in Hadoop via inter-process channels, and having to carefully consider the limitations that client applications impose on the resolution of data; that is, some data cannot be split further than it already is according to the specifications of the application.

It is clear that even if our ideas about integrating Cooperative Computing into Hadoop provide efficiency improvements in some cases, this method of recognizing speculative tasks will likely not be possible or efficient in all cases. For example, our scheme is not designed to handle a situation in which a node has completely failed; in this case it's clear that a task must be entirely replicated, and locality can probably not be utilized because the data transfer will need to come from a different DataNode altogether. While our initial implementation takes the course of always using Cooperative Computing in cases of lagging nodes, a truly viable contribution to Hadoop would need to consider the situation and recognize when this is an appropriate action to take, and when other existing methods should be used.

Finally, our enhancement is only implemented to be safe in the scenario in which there is a single job running on the system at a given time. We believe that these circumstances allow us to view the experiments (described below) as a proof of concept while avoiding added complexity that would become intractable to deal with in the project time-frame. As it stands, approximately 300 lines of code were added or changed.

5 Experimental Methodology

5.1 Cluster Setup

In order to work in a closed and controlled environment, we set up a cluster of five computers in the UW Computer Science crash-and-burn lab, and fitted them with the Ubuntu 9.04 operating system and a pre-compiled distribution of DummyNet [11], which is an open-source network simulation package. In this way we were able to approximate the heterogeneous bandwidth characteristics of a miniature data center which consists of two racks containing two and three machines respectively. Bandwidth between the two virtual racks was throttled to simulate a data center over-subscription scenario in which inter-rack switches are clogged by more traffic than they can handle [10]. This setup also allowed us to maintain a controlled environment free of external factors, in contrast with the open computer labs whose machines may have any number of background processes running at any time. Figure 4 shows our testbed.

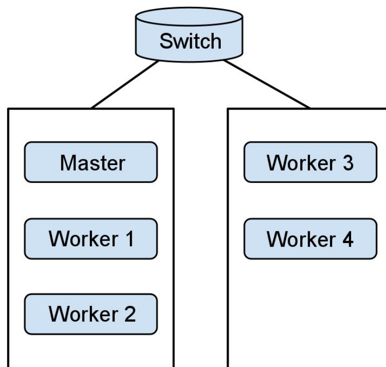

Figure 4: Crash-and-burn lab virtual data center

5.2 Hadoop Configuration

In the course of testing, several settings were manipulated to try and drive the execution towards the use of our enhancement. In order to monitor various code execution paths and determine when and how often our new code was being executed, we enabled `log4j` logging functionality for the involved Hadoop classes and added various logging statements. The normal-operation number of DFS replicas was set to 1 because of the limited size of our cluster. While HDFS has the option of replicating blocks further than this, this setting increases the chance of having to transfer data to facilitate a speculative task. The total number of map and reduce tasks for the system was adjusted between 3 and 40 in an attempt to find what load produced the most interesting results; this range represents settings that are reasonable for a cluster of this size. Ultimately, we ran the test suites with 5 map tasks and 3 reduce tasks, as well as with 40 map tasks and 20 reduce tasks.

Task speculation is enabled by default, but we were able to adjust the frequency at which speculative tasks were kicked off by specifying a smaller threshold for determining whether a task is slow (changing it from 1 standard deviation from the mean to 0.2 standard deviations), and by increasing the cap on the total number of speculative tasks from 10% of all tasks in the system to 50%.

5.3 Applications

The canonical MapReduce application is to count the occurrences of individual words across a set of text files. After some experimentation with more complex/useful/realistic applications, we decided that the simple word count scenario provided an easy-to-manipulate application with which to test our modifications to Hadoop. Since our goal was to provide a proof-of-concept rather than an optimal solution, we considered success to be defined by finding a particular case in which our modifications provide substantially better performance than the standard Hadoop implementation. This contrived case can then be generalized to a class of applications that should theoretically realize performance benefits.

The manipulation of the behavior of MapReduce executions is achieved through the generation of skewed text files as input to the job. Two different types of skew are introduced to affect the map and reduce phases independently.

First, variability in the size of the text files can be used to throw the map phase out of balance. In the typical case of applications that process a collection of text files, Hadoop is able to divide the

data into parts such that an even load is distributed to each worker. However there are certainly applications for which such a data partition is not possible, where the input files must be processed as a whole. (For example, some compression algorithms such as `GZIP` are non-splittable because the compression index is only present at the beginning of the file.) Hadoop provides facilities to specify that input files should not be divided, and we make use of such an application to simulate such a scenario.

Secondly, an imbalance in the distribution of keys passed from a map phase to a reduce phase naturally leads to a skew in the distribution of the reduce workload across nodes. This results from the fact that during reduce, all key-value pairs that share a particular key must be processed on a single node. Related to our word count application, this is achieved by generating input files that contain a great number of instance of one particular word, and relatively few occurrences of all others.

For concrete implementations of the applications described above, we ended up using example programs that are included with the Hadoop distribution. Because the 0.21.0 version of Hadoop that we chose to modify is a transitional version, it contains aspect of both the older `mapred` and the replacement `mapreduce` packages. (We chose this version because it was the latest stable release, and only found out later that this was the case.) Though it made development more complex, this dichotomy did provide us with extra test cases; we tested our implementation using the `WordCount` and `MultiFileWordCount` (which prevents the splitting of input files) applications provided with the `mapreduce` package, as well as a `WordCount` application that is included with Hadoop version 0.19.0 and call into the older `mapred` code.

5.4 Environment Setup

As mentioned previously, we used DummyNet ipfirewall traffic shaper to simulate bandwidth restrictions within the cluster. The master node (which runs both the MapReduce JobTracker and the HDFS NameNode) was located within one virtual rack with two worker nodes, while the remaining two worker nodes were in a rack unto themselves. Intra-rack communication was allowed to proceed at 100MB/s, while inter-rack communication was limited to 10MB/s.

In order to cause increases stress on a single machine during execution of a job, the widely available Linpack benchmarking tool was initiated on a node immediately prior to the job submission. The hope was to create additional imbalance in the system such that the target node would be recognized as falling behind the others. In the tests we have carried out thus far, three simultaneous executions of Linpack were forked before the job began and restarted if they finished before the job itself; to avoid variability in results due to memory leaks or overlapping executions, all instances of Linpack were killed when the job completed.

5.5 Experimental Design

We approached experimentation with the goal of finding potentially advantageous circumstances for Cooperative Computing by comparing the duration of MapReduce runs using the original Hadoop code to identical runs using our modifications. Variables involving the input size and applications were introduced to broaden the scope of our experiments and help pinpoint the differences in the two implementations.

- Each run processes an input set of one of three sizes that we generated via script: small (500,000 words), medium (5,000,000 words) and large (50,000,000 words). Each of these input sets was skewed heavily in both the file sizes and word frequency.

- Each run of the test suite included executions of each of the three applications that are described above.
- A test suite was run without DummyNet enabled to serve as a control.
- A third test suite was run (with DummyNet enabled) with a higher number of tasks to deploy per job specified in the Hadoop configuration.

The experimental runs began by refreshing the entire system, including wiping all Hadoop logs and deleting the entire HDFS file system. The file system was then reinitialized and the specified input set was uploaded. After this cleansing process, the was always one "priming" run of an applications. We had previously observed that the first application run after initializing a Hadoop system tended to be substantially slower, the priming process removes this variability. Then a specified number of repetitions of executions of the three word count applications. After each execution the following items were gathered.

- The JobTracker logs contained the debug information showing how often execution paths of interest were reached.
- The Hadoop command line output, which contains timing statistics for the execution.
- The actual output of the MapReduce job pulled in from HDFS, which verifies correctness of the execution.

6 Results and Discussion

Scripts were written to process the output that was gathered as described above. The results are summarized in the three tables in 5, 6 and 7.

App	Cooperative	Small	Medium	Large
WordCount (mapred)	n	33.97	65.11	345.62
	y	34.66	64.60	372.54
WordCount (mapreduce)	n	30.31	53.77	180.54
	y	30.08	52.97	181.65
MultiFileWordCount	n	23.00	51.34	300.19
	y	23.40	50.86	338.08

Figure 5: Measurements without DummyNet network manipulation, low tasks per job. Average duration (in seconds) is shown for small, medium and large input sizes.

The data in 5 represents the case in which there is no concept of rack-locality because DummyNet was not enabled. Therefore, it stands to reason that our implementation (without successfully completing the re-splitting idea), did not fare as well. As mentioned, to enable our implementation we had to disable Hadoop's normal speculative task assignment algorithms. Observing the substantial loss of efficiency in the "large" executions, we speculate that there are optimizations within the normal Hadoop speculation mechanism that we failed to replicate effectively in our implementation.

App	Cooperative	Small	Medium	Large
WordCount (mapred)	n	34.12	80.46	411.60
	y	35.12	81.58	457.92
WordCount (mapreduce)	n	29.77	52.38	185.20
	y	29.85	52.69	**164.35**
MultiFileWordCount	n	23.00	52.65	312.60
	y	23.31	52.08	337.42

Figure 6: Measurements with rack simulation using DummyNet, low tasks per job. Average duration (in seconds) is shown for small, medium and large input sizes.

App	Cooperative	Small	Medium	Large
WordCount (mapred)	n	92.71	125.19	439.80
	y	103.43	142.19	483.20
WordCount (mapreduce)	n	51.14	78.48	209.80
	y	54.19	76.33	221.20
MultiFileWordCount	n	44.19	73.48	340.00
	y	46.48	71.43	346.20

Figure 7: Measurements with rack simulation using DummyNet, many tasks per job. Average duration (in seconds) is shown for small, medium and large input sizes.

Figure 6 displays a triumph of our implementation over the performance of standard Hadoop (in bold) in the context of the `mapreduce WordCount` application, while the other two applications still show substantial preference for the original Hadoop. We have no definitive answer as to why this is the case, but it may be due to the fact that most of our development was done against `mapreduce WordCount`, and so the execution paths that this application exercises are the ones which we paid the most attention to. It's also possible that, despite our efforts to eliminate environmental variability, there were some external factors at work during the time of day at which the suites were kicked off. If work were to continue on this project, the primary goal would be to try to decipher the cause of this result and to rerun the test suite under similar circumstances.

In all test suites, the differences in execution time between our implementation and standard Hadoop under the "small" and "medium" input sets are small enough to be attributed to noise. This is despite the fact that task speculation did occur with some frequency even in these less intensive runs. Because of the substantial start-up and coordination overhead in any MapReduce job (even after the priming run), it's understandable that longer runs would demonstrate more of a divide. Given the limited time we had for experimentation, measuring runs with larger input than the "large" input set was impractical, but this would be an option if more time were available.

7 Future Work

We plan to investigate the interesting data points in the results sections to see what is the exact reason for significantly higher performance improvement in certain scenarios versus the other scenarions. This will help us to solidify our theories about when and why any performance benefits occur, and perhaps improve the Cooperative Computing design even more.

We would like to finish the implementation to include all aspects of Cooperative Computing (i.e. spliting the task into sub-tasks). It would be also useful to make the Hadoop MapReduce

framework smarter so that it strategically optimizes the use of Cooperative Computing, in lieu of mutual exclusion between methods of task speculation.

It would be great if we are able to test Cooperative Computing in a real Hadoop MapReduce cluster. This helps us to better understand the benefits and limitations of our approach since our small tesbed might not be an accurate representation of a true data center.

8 Conclusion

We introduce the notion of Cooperative Computing where we leverage the physical proximity of nodes in a distributed system to reduce overall job completion time. In Cooperative Computing We divide the task that is running on a slow node to sub-tasks and assign them to the nearby nodes. In this way, nodes close to each other can cooperate to complete a task when a particular node is overloaded.

Our mechanism differs from standard Hadoop task distribution facilities in a couple ways. First, we attempt to partition the data for a task and the task itself, rather than replicating each of these. Cooperative Computing also utilizes proximity for the sake of efficiency. In the course of implementation, we landed upon an alternate method of distributing speculative tasks that may be interesting to explore further: being proactive with regards to slow tasks rather than to TaskTrackers that can take on more work.

We prototype Cooperative Computing in the Hadoop MapReduce framework with no overhead on the system and without sacrificing any correctness guarantees that Hadoop provides. Initial results show that in many cases our system performs the same or better than the original Hadoop implementation. Cooperative Computing performs encouraging performance improvements in certain scenarios.

We envision that Cooperative Computing can be incorporated into consistent hashing with minimal overhead. Prototyping this system remains as future work.

References

[1] L. A. Barroso and U. Hölzle. "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines", M. D. Hill, Ed. San Rafael, CA, USA: Morgan & Claypool, 2009

[2] Jeffrey Dean , Sanjay Ghemawat, "MapReduce: simplified data processing on large clusters," Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation, p.10-10, December 06-08, 2004, San Francisco, CA

[3] Ganesh Ananthanarayanan , Srikanth Kandula , Albert Greenberg , Ion Stoica , Yi Lu , Bakes Saha , Edward Harris, "Reining in the outliers in map-reduce clusters using Mantri," Proceedings of the 9th USENIX conference on Operating systems design and implementation, p.1-16, October 04-06, 2010, Vancouver, BC, Canada

[4] Thomas Sandholm , Kevin Lai, "MapReduce optimization using regulated dynamic prioritization," Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, June 15-19, 2009, Seattle, WA, USA

[5] "Wireless grid", Wikipedia, `http://en.wikipedia.org/wiki/Wireless_grid` (Retrieved April 12th, 2011)

[6] Ganesh Ananthanarayanan , Srikanth Kandula , Albert Greenberg , Ion Stoica , Yi Lu , Bikas Saha , Edward Harris, "Reining in the outliers in map-reduce clusters using Mantri", Proceedings of the 9th USENIX conference on Operating systems design and implementation, p.1-16, October 04-06, 2010, Vancouver, BC, Canada

[7] Antony I. T. Rowstron , Peter Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems", Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, p.329-350, November 12-16, 2001

[8] Haiying Shen , Cheng-Zhong Xu, "Hash-based proximity clustering for efficient load balancing in heterogeneous DHT networks", Journal of Parallel and Distributed Computing, v.68 n.5, p.686-702, May, 2008

[9] Gennaro Cordasco, Alberto Negro, Alessandra Sala, Vittorio Scarano, "PON: Exploiting Proximity on Overlay Networks," ipdps, pp.469, 2007 IEEE International Parallel and Distributed Processing Symposium, 2007

[10] Mohammad Al-Fares, Alexander Loukissas, Amin Vahdat, "A scalable, commodity data center network architecture," Proceedings of the ACM SIGCOMM 2008 conference on Data communication, August 17-22, 2008, Seattle, WA, USA [doi¿10.1145/1402958.1402967]

[11] Luigi Rizzo, "DummyNet," `http://info.iet.unipi.it/~luigi/dummynet/`

[12] Michael Stonebraker , Paul M. Aoki , Witold Litwin , Avi Pfeffer , Adam Sah , Jeff Sidell , Carl Staelin , and Andrew Yu, "Mariposa: a wide-area distributed database system", The VLDB Journal The International Journal on Very Large Data Bases, v.5 n.1, p.048-063, January 1996

[13] Giuseppe DeCandia , Deniz Hastorun , Madan Jampani , Gunavardhan Kakulapati , Avinash Lakshman , Alex Pilchin , Swaminathan Sivasubramanian , Peter Vosshall , Werner Vogels, "Dynamo: amazon's highly available key-value store", Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, October 14-17, 2007, Stevenson, Washington, USA

[14] "Amazon EC2 outage calls availability zones into question", NetworkWorld, `http://www.networkworld.com/news/2011/042111-amazon-ec2-zones.html` (Retrieved May 9th, 2011)