MapReduce on a Chord Distributed Hash Table

Andrew Rosen Brendan Benshoof Matt Erwin Robert W. Harrison Anu G. Bourgeois
Department of Computer Science
Georgia State University
Atlanta, Georgia
rosen@cs.gsu.edu

***Abstract*—MapReduce platforms are designed for datacenters, which are highly centralized in nature. These platforms require a centralized source to maintain and coordinate the network, which leads to a single point of failure. We present ChordReduce, a generalized and completely decentralized MapReduce framework which utilizes the peer-to-peer protocol Chord. ChordReduce's robustness, scalability, and lack of a single point of failure allows MapReduce to be easily deployed in a greater variety of contexts, including cloud and loosely coupled environments.**

***Index Terms*—MapReduce; P2P; Parallel Processing; Peer-to-Peer Computing; Cloud Computing; Middleware;**

I. Introduction

Google's MapReduce [1] paradigm has rapidly become an integral part in the world of data processing and is capable of efficiently executing numerous Big Data programming and data-reduction tasks. By using MapReduce, a user can take a large problem, split it into small, equivalent tasks and send those tasks to other processors for computation. The results are sent back to the user and combined into one answer. MapReduce has proven to be an extremely powerful and versatile tool, providing the framework for using distributed computing to solve a wide variety of problems, such as distributed sorting and creating an inverted index [1].

Popular platforms for MapReduce, such as Hadoop [2], are explicitly designed to be used in large datacenters [3] and the majority of research has been focused there. These MapReduce platforms are highly centralized and tend to have single points of failure[4] as a result. A centralized design assumes that the network is relatively unchanging and does not usually have mechanisms to handle node failure during execution or, conversely, cannot speed up the execution of a job by adding additional workers on the fly. Finally deploying these systems and developing programs for them has an extremely steep learning curve.

We were motivated to start exploring a more abstract deployment for MapReduce, one that could be deployed in a much wider variety of contexts, from peer-2-peer frameworks to datacenters. Our framework cannot and does not rely on many common assumptions, such as a dedicated and static network of homogeneous machines. Nor do we assume that failed nodes will recover [3]. Finally, our framework needed to be easy to deploy and simple to use.

We used Chord [5], a peer-2-peer distributed hash table, as the backbone for developing our system. Our paper presents these contributions:

- We define the architecture and components of ChordReduce and demonstrate how they fit together to perform MapReduce jobs over a distributed system without the need of a central scheduler or coordinator, avoiding a central point of failure[1]. We also demonstrate how to create programs to solve MapReduce problems using ChordReduce (Section *ChordReduceNum*).
- We used built a prototype system implementing ChordReduce and deployed it on Amazon's Elastic Cloud Compute. We tested our deployment by solving Monte-Carlo computations and word frequency counts on our network (Section *ExperimentNum*)[2].
- We prove that ChordReduce is scalable and highly fault tolerant, even under high levels of churn and can even benefit from churn under certain circumstances. Specifically, we show it is robust enough to reassign work during runtime in response to nodes entering and leaving the network (Section *ResultsNum*).
- We contrast ChordReduce with similar architectures and identify future areas of fruitful research using ChordReduce (Section *Future/Related Work*).

II. Background

ChordReduce takes its name from the two components it is built upon. Chord[5] provides the backbone for the network and the file system, providing scalable routing, distributed storage, and fault-tolerance. MapReduce runs on top of the Chord network and utilizes the underlying features of the distributed hash table. This section provides background on Chord and MapReduce.

A. Chord

Chord [5] is a peer-to-peer (P2P) protocol for file sharing and distributed storage that guarantees a high probability $\log_2 N$ lookup time for a particular node or file in the network. It is highly fault-tolerant to node failures and churn, the constant joining and leaving of nodes. It scales extremely well and the network requires little maintenance to handle individual nodes. Files in the network are distributed evenly among its members.

[1] TODO: defend why the starting node isn't considered to be this in corresponding section

[2] The point to make here is that we developed and tested it (results of tests is the next section). Do we want to mention specifically what tests we run or leave it more abstract.

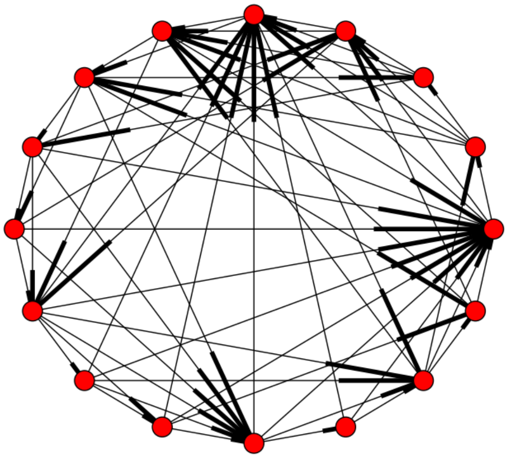

Fig. 1: A Chord ring 16 nodes where $m = 4$. The bold lines are incoming edges. Each node has a connection to its successor, as well as 4 fingers, some of which are duplicates.

As a distributed hash table (DHT), each member of the network and the data stored on the network is mapped to a unique m-bit key or ID, corresponding to one of 2^m locations on a ring. The ID of a node and the node itself are referred to interchangeably.

In a traditional Chord network, all messages travel in one direction - upstream, hopping from one node to another with a greater ID until it wraps around. A node in the network is responsible for all the data with keys *above or upstream* his predecessor, up through and including its own ID. If a node is responsible for some key, it is referred to being the successor of that key.

Robustness in the network is accomplished by having nodes backup their contents to their s (often 1) immediate successors, the closest nodes upstream. This is done because when a node leaves the or fail, the most immediate successor would be responsible for the content its content.

Each node maintains a table of m shortcuts to other peers, called the finger table. The ith entry of a node n's finger table corresponds to the node that is the successor of the key $n + 2^{i-1} \mod 2^m$. Nodes route messages to the finger that is closest to the sought key without going past it, until it is received by the responsible node. This provides Chord with a highly scalable $\log_2(N)$ lookup time for any key [5].

As nodes enter and leave the ring, the nodes use their maintenance procedures to guide them into the right place and repair any links with failed nodes. Full details on Chord's maintenance cycle are beyond the scope of this paper and can be found here [5].

B. *MapReduce*

At its core, MapReduce [1] is a system for division of labor, providing a layer of separation between the programmer and the more complicated parts of concurrent processing. The programmer sends a large task to a master node, who then divides that task among slave nodes (which may further divide the task). This task has two distinct parts: Map and Reduce. Map performs some operation on a set of data and then produces a result for each Map operation. The resulting data can then be reduced, combining these sets of results into a single set, which is further combined with other sets. This process continues until one set of data remains. A key concept here is the tasks are distributed to the nodes that already contain the relevant data, rather than the data and task being distributed together among arbitrary nodes.

The archetypal example of using MapReduce is counting the occurrence of each word in a collection of documents, called WordCount. These documents have been split up into blocks and stored on the network over the distributed file system. The master node locates the worker nodes with blocks and sends the Map and Reduce tasks associated with WordCount. Each worker then goes through their blocks and creates a small word frequency list. These lists are then used by other workers, who combine them into larger and larger lists, until the master node is left with a word frequency list of all the words in the documents.

Insert psuedocode or picture, code for word count here!

The most popular platform for MapReduce is Hadoop [2]. Hadoop is an open-source Java implementation developed by Apache and Yahoo! [6]. Hadoop has two components, the Hadoop Distributed File System (HDFS) [7] and the Hadoop MapReduce Framework [8]. Under HDFS, nodes are arranged in a hierarchical tree, with a master node, called the NameNode, at the top. The NameNode's job is to organize and distribute information to the slave nodes, called DataNodes. This makes the NameNode a single point of failure [4] in the network, as well as a potential bottleneck for the system [9].

To do work on Hadoop, the user stores their data on the network. This is handled by the NameNode, which equally apportions the data among the DataNodes. When a user wants to run some analysis on the data or some subset the data, then that function is sent by the NameNode to each of the DataNodes that is responsible for the indicated data. After the DataNode finishes processing, the result is handled by other nodes called Reducers which collect and reduce the results of multiple DataNodes.

III. ChordReduce

Popular platforms for MapReduce, such as Hadoop, are extremely powerful, but have some inherent limitations. These platforms are designed to be deployed in a data center. Their architecture relies on multiple nodes with specific roles to coordinate the work, such as the NameNode and JobTracker. These nodes perform necessary scheduling and distribution tasks and help provide fault-tolerance to the network as

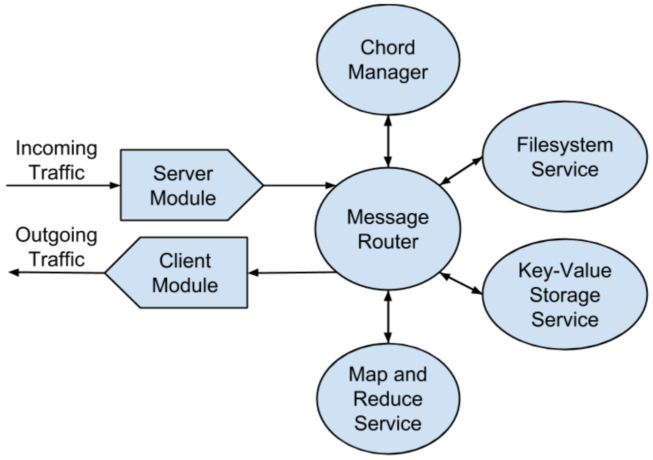

Fig. 2: The basic architecture of a node in ChordReduce. MapReduce runs as a service on top of each node.

a whole, but in doing so become single points of failure themselves.

ChordReduce is designed as a more abstract framework for MapReduce, able to run on any arbitrary distributed configuration. ChordReduce leverages the features of distributed hash tables to handle distributed file storage, fault tolerance, and lookup. We designed ChordReduce to ensure that no single node is a point of failure and that there is no need for any node to coordinate the efforts of other nodes during processing.

Our central design philosophy was to implement additions to the Chord protocol by leveraging the existing features of Chord. By treating each task or target computation as an object of data, we can distribute them in the same manner as files and rely on the protocol to route them and provide robustness.

ChordReduce is similar to blank and blank

insert architecture layer diagram and *insert work flow diagram. Chord tree of stage, map , reduce.*

A. File Storage

The design of a distributed file system is closely tied to the design to the implementation of MapReduce [10] [7]. Our system uses CFS [11], short for Cooperative File System, to store files.

Everything in Chord, be it data or a node, is given a hash identifier or key. The ID of a node is the hash of their IP address and port, while the key for a file is the hash of its filename. In the initial version of Chord, the entire file would be stored in the node with the ID equal or closest upstream to the file's key.

Dabek et. al found that by splitting the file into blocks and storing each block in a different node greatly improved the system's load balancing when compared to storing the entire file on a single node[11]. ChordReduce implements the same system. Files are split into approximately equally sized blocks. Each block is treated as an individual file and is assigned a key equal to the hash of it's contents. The block is then stored at the node responsible for that key.

The node which would normally be responsible for the whole file instead stores a *keyfile*. The keyfile is an ordered list of the keys corresponding to the files' block and is created as the blocks are assigned their respective keys. When the user wants to retrieve a file, they first obtain the keyfile and then request each block specified in the keyfile.

B. Decentralized MapReduce and Data Flow

In ChordReduce's implementation of MapReduce, each node takes on responsibilities of both a worker and master, much in the same way that a node in a P2P file-sharing service acts as both a client and a server. To start a job, the user contacts a node at a specified hash address and provides it with the tasks. This address can be chosen arbitrarily or be a known node in the ring. The node at this hash address is designated as the *stager*.

The job of the stager divide the work into *data atoms*, which define the smallest individual units that work can be done on. This might represent block of text, the result of a summation for a particular intermediate value, or a subset of items to be sorted. The specifics of how to divide the work are defined by the user in a *stage* function. The data atoms also contain the Map and Reduce functions defined by the user.

If the user wants to perform a MapReduce job over data on the network, the stager locates the keyfile for the data and creates a data atom for each block in the file. Each data atom is then sent to the node responsible for their corresponding block. When the data atom reaches it's destination node, that node retrieves the necessary data and applies the Map function. The results are stored in a new data atom, which are then sent back to the stager's hash address (or some other user defined address). This will take $\log_2 n$ hops traveling over Chord's fingers. At each hop, the node waits a predetermined minimal amount of time to accumulate additional results (In our experiments, this was 100 milliseconds). Nodes that receive at least two results merge them using the Reduce function. The results are continually merged until only one remains at the hash address of the stager.

MapReduce jobs don't rely on a file stored on the network, such as a Monte-Carlo approximation, create data atoms specified by the user stage function in the stage function. The data atoms are then each given a random hash and sent to the node responsible for that hash address, guaranteeing they are evenly distributed throughout the network. From there, the execution is identical to the above scenario.

Once all the Reduce tasks are finished, the user retrieves his results from the node at the stager's address. This may not be the stager himself, as the stager may no longer be in the network. The stager does not need to collect the results himself, since the work is sent to the stager's hash address, rather than the stager itself. Thus, the stager could quit the network after staging, and both the user and the network would be unaffected by the change.

Similar precautions are taken for nodes working on Map and Reduce tasks. Those tasks are backed up by a node's successor, who will run the task if the node leaves before finishing its work (e.g. the successor loses his predecessor). The task is given a timeout by the node. If the backup node detects that the responsible node has failed, he starts the work and backs up again to *his* successor. Otherwise, the data is tossed away once the timeout expires. This is done to prevent a job being submitted twice.

An advantage of our system is the ease of development and deployment. The developer does not need to worry about distributing work evenly, nor does he have to worry about any node in the network going down. The stager does not need to keep track of the status of the network. The underlying Chord ring handles that automatically. If the user finds they need additional processing power during runtime, they can boot up additional nodes, which would automatically be assigned work based on their hash value. If a node goes down while performing an operation, his successor takes over for him. This makes the system extremely robust during runtime.

All a developer needs to do is write three functions: the staging function, Map, and Reduce. These define how to split up the work into manageable portions, the work to be performed on each portion to obtain results, and how to combine these results into a single result, respectively.

C. Fault-Tolerance

Due to the potentially volatile nature of a peer-to-peer network, ChordReduce has to be able to handle an arbitrary amount of churn. When a node fails or leaves Chord, the failed node's successor will become responsible for all of the failed nodes keys. As a result, each node in the ChordReduce network relies on their successor to act as a backup.

To prevent data from becoming irretrievable, each node periodically sends backups to its successor. In order to prevent a cascade of backups of backups, the node only passes along what it considers itself responsible for. What a node is responsible for changes as nodes enter and leave the network. If a node's successor leaves, the node sends a backup to his new successor. If the node fails, the successor is able to take his place almost immediately. This scheme is used to not only backup files, but the Map and Reduce tasks and data atoms as well.

This procedure prevents any single node failure or sequences of failures from harming the network. Only the failure of multiple neighboring nodes poses a threat to the network's integrity. Furthermore, since a node's ID in the network does not map to a geographical location, any failure that affects multiple nodes simultaneously will be spread uniformly throughout, rather than hitting successive nodes. This means if successive nodes to fail simultaneously, they did so independently.

This concept can be extended to provide additional robustness. Suppose that each node has failure rate $r < 1$ and that the each node backs up their data with s successive nodes downstream. If one of these nodes fail, the next successive node takes its place and the next upstream node becomes another backup. This ensures there will always be s backups. The integrity of the ring would only be jeopardized if $s + 1$ successive nodes failed almost simultaneously, before the maintenance cycle would have a chance to correct for the failed nodes. The chances of such an event would be $r^s + 1$, as each failure would be independent *I think the citation is CFS*.

A final consequence of this is load-balancing during runtime. As new nodes enter the network, they change their successor as the maintenance ycle guides them into the correct location in the ring. When a node n changes his successor, n asks if the successor is holding any data n should be responsible for. The successor looks at all the data n is responsible for and sends it to n. The successor maintains this data as a backup for n. Because Map tasks are backed up in the same manner as data, a node can take the data and corresponding tasks he's responsible for and begin performing Map tasks immediately.

IV. Experiments

In order for ChordReduce to be a viable framework, we had to show these three properties:

1) ChordReduce provides significant speedup during a distributed job.
2) ChordReduce scales.
3) ChordReduce handles churn during execution.

Speedup can be demonstrated by showing that a distributed job is generally performed more quickly than the same job handled by a single worker. More formally we need to establish that $\exists n$ such that $T_n < T_1$, where T_n is the amount of time it takes for n nodes to finish the job.

To establish scalability, we need to show that the cost of distributing the work grows logarithmically with the number of workers. In addition, we need to demonstrate that the larger the job is, the number of nodes we can have working on the problem without the overhead incurring diminishing returns increases. This can be stated as

$$T_n = \frac{T_1}{n} + k \cdot \log_2(n)$$

where $\frac{T_1}{n}$ is the amount of time the job would take when distributed in an ideal universe and $k \cdot \log_2(n)$ is network induced overhead, k being an unknown constant dependent on network latency and available processing power.

Finally, to demonstrate robustness, we need to show that ChordReduce can handle arbitrary node failure in the ring and that such failures minimally impair the overall speed of the network.

A. Experimental Deployment

We built a fully functional prototype of ChordReduce in python. One difficulty with Hadoop is the installation an configuration of the software on arbitrary machines; our design is intended to be as quick to deploy as possible.

We ran our experiments using Amazon's Elastic Compute Cloud (EC2) service. Amazon EC2 allows users to purchase an arbitrary number of virtual machines and pay for the machines by the hour. Each node was an individual EC2 small instance

[12] with a Ubuntu 12.04 image preconfigured with Git and a small startup script which retrieves the latest version of the code when the node is booted..

Instrumentation Node: Once started, nodes retrieve the latest version of the code and run it as a service, automatically joining the network. We can choose any arbitrary node as the stager and tell it to run the MapReduce process. We found that the network was robust enough that we could take a node we wanted to be the stager out of the network, modify its MapReduce test code, have it rejoin the network, and then run the new code without any problems. Since only the stager has to know how to create the Map tasks, the other nodes do not have to be updated and execute the new tasks they are given. However, this process was extremely tedious and time consuming.

We created an additional node to help configure the experiment, which we call the 'configurer'[3]. We avoid calling it the 'manager' or 'coordinator' as we don't want to create the false impression that the configurer directly affects the experiments. The configurer's roll is to aid in the generation and collection of data.

The configurer does not actively participate in computing or distributing the work. We use it to modify the experimental variables in each node without having to shutdown the node or instance. We also use it to simulate the effects of churn on the network.

ChordReduce is a fully functional Chord implementation in Python. Our installation was designed to be as simple as possible. It consists of downloading our code [13] and running chord.py. A user can specify a port and IP of a node in the ring they wish to join. The node will automatically integrate into the ring with this minimal information. The ring as implemented is stable and well organized. We created various services to run on top the network, such as a file system and distrubuted web server. Our file system is capable of storing whole files or splitting the file up among multiple nodes the ring. Our MapReduce module is a service that runs on top of our Chord implementation, similar to the file system (Fig. 2). We avoided any complicated additions to the Chord architecture; instead we used the protocol's properties to create the features we desired in our MapReduce framework.

B. Setup

To stress test our framework, we ran a Monte-Carlo approximation of π. This process is largely analogous to having a square with the top-right quarter of a circle going through it (Fig. 3), and then throwing darts at random locations. Counting the ratio of darts that land inside the circle to the total number of throws gives us an approximation of $\frac{\pi}{4}$. The more darts thrown, i.e. the more samples that are taken, the more accurate the approximation[4].

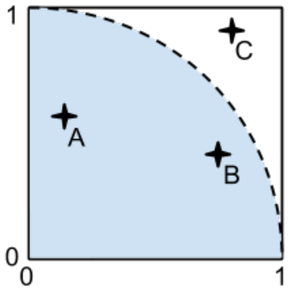

Fig. 3: The "dartboard." The computer throws a dart by choosing a random x and y between 0 and 1. If $x^2 + y^2 < 1^2$, the dart landed inside the circle. A and B are darts that landed inside the circle, while C did not.

We chose this experiment for a number of reasons. The job is extremely easy to distribute. This also made it very easy to test scalability. By doubling the amount of samples, we can double the amount of work each node gets. We could also test the effectiveness of distributing the job among different numbers of workers.

Each Map job is defined by the number of throws the node must make and yields a result containing the total number of throws and the number of throws that landed inside the circular section. Reducing these results is then a matter of adding the respective fields together.

We ran our experiments on groups of 1, 10, 20, 30, and 40 workers, which generated a 10^8 sample set and a 10^9 sample set. Additionally, we gathered data on a 10^7 sample set using 1, 5, 10, 20, 30 workers. To test churn, we ran an experiment where each node had an equal chance of leaving and joining the network and varied the level of churn over multiple runs.

We also utilized a subroutine we wrote called $plot$, which sends a message sequentially around the ring to establish how many members there are. If $plot$ failed to return in under a second, the ring was experiencing structural instability.

V. Results

Fig. 4 and Fig. 5 summarize the experimental results of job duration and speedup. Our default series was the 10^8 samples series. On average, it took a single node 431 seconds, or approximately 7 minutes, to generate 10^8 samples. Generating the same number of samples using ChordReduce over 10, 20, 30, or 40 nodes was always quicker. The samples were generated fastest when there were 20 workers, with a speedup factor of 4.96, while increasing the number of workers to 30 yielded a speedup of only 4.03. At 30 nodes, the gains of distributing the work were present, but the cost of overhead ($k \cdot \log_2(n)$) had more of an impact. This effect is more pronounced at 40 workers, with a speedup of 2.25.

Since our data showed that approximating π on one node with 10^8 samples took approximately 7 minutes, collecting 10^9 samples on a single node would take 70 minutes at minimum.

[3] Or possibly instrumentation node

[4] This is not intended to be a particularly good approximation of π. Each additional digit of accuracy requires increasing the number of samples taken by an order of magnitude.

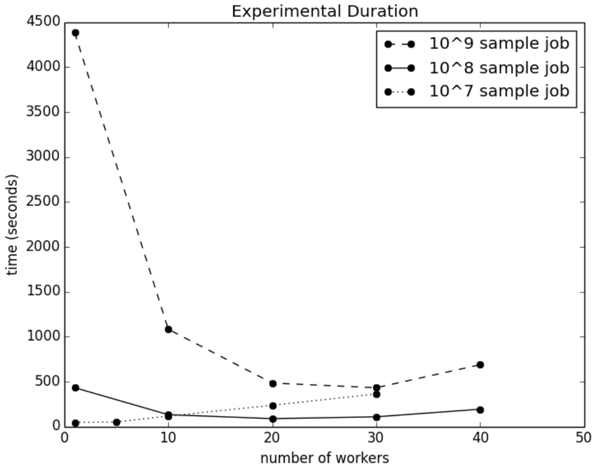

Fig. 4: For a sufficiently large job, it was almost always preferable to distribute it. When the job is too small, such as with the 10^7 data set, our runtime is dominated by the overhead. Our results are what we would expect when overhead grows logarithmically to the number of workers.

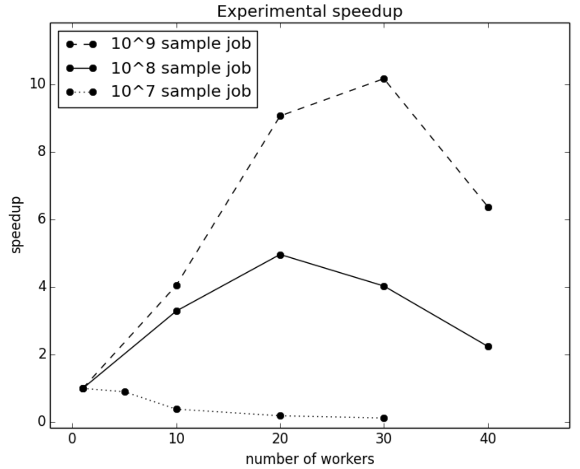

Fig. 5: The larger the size of the job, the greater the gains of distributing with ChordReduce. In addition, the larger the job, the more workers can be added before we start seeing diminishing returns. This demonstrates that ChordReduce is scalable.

Fig. 5 shows that the 10^9 set gained greater benefit from being distributed than the 10^8 set, with the speedup factor at 20 workers being 9.07 compared to 4.03. In addition, the gains of distributing work further increased at 30 workers and only began to decay at 40 workers, compared with the 10^8 data set, which began its drop off at 30 workers. This behavior demonstrates that the larger the job being distributed, the greater the gains of distributing the work using ChordReduce.

The 10^7 sample set confirms that the network overhead is logarithmic. At that size, it is not effective to run the job concurrently and we start seeing overheard acting as the dominant factor in runtime. This matches the behavior predicted by our equation, $T_n = \frac{T_1}{n} + k \cdot \log_2(n)$. For a small T_1, $\frac{T_1}{n}$ approaches 0 as n gets larger, while $k \cdot \log_2(n)$, our overhead, dominates the sample. The samples from our data set fit this behavior, establishing that our overhead increases logarithmically with the number of workers.

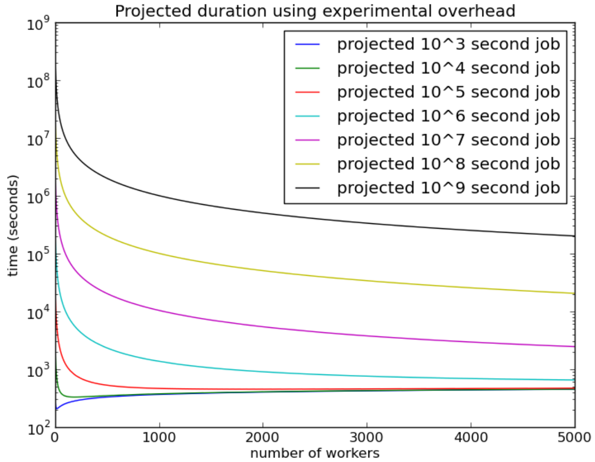

Fig. 6: The projected runtime using ChordReduce for differently sized jobs. Each curve projects the expected behavior for job that takes a single worker the specified amount of time.

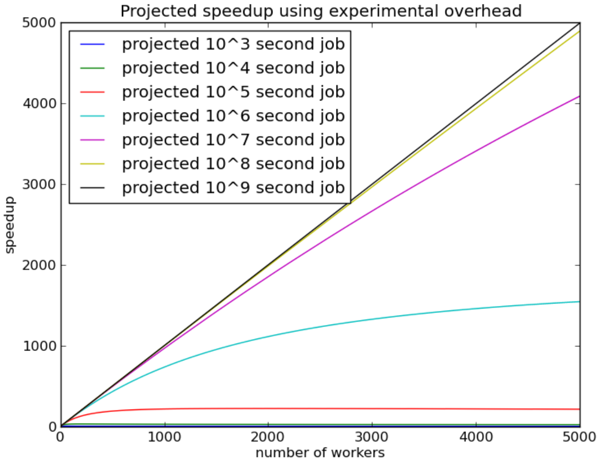

Fig. 7: The projected speedup for different sized jobs.

Since we have now established that $T_n = \frac{T_1}{n} + k \cdot \log_2(n)$, we can estimate how long a job that takes an arbitrary amount of time to run on a single node would take using ChordReduce. Our data points indicated that the mean value of k for this problem was 36.5. Fig. 6 shows that for jobs that would take more than 10^4 seconds for single worker to complete, we can expect there would still be benefit to adding an additional worker, even when there are already 5000 workers already in the ring. Fig. 7 further emphasizes this. Note that as the

Churn rate per second	Average runtime (s)	Speedup vs 0% churn
0.8%	191.25	2.15
0.4%	329.20	1.25
0.025%	431.86	0.95
0.00775%	445.47	0.92
0.00250%	331.80	1.24
0%	441.57	1.00

TABLE I

jobs become larger, the expected speedup from ChordReduce approaches linear behavior.

Table I shows the experimental results for different rates of churn. These results show the system is relatively insensitive to churn. We started with 40 nodes in the ring and generated 10^8 samples while experiencing different rates of churn, as specified in Table I. At the 0.8% rate of churn, there is a 0.8% chance each second that any given node will leave the network followed by another node joining the network at a different location. The joining rate and leaving rate being identical is not an unusual assumption to make [14] [15].

Our testing rates for churn are an order of magnitude higher than the rates used in the P2P-MapReduce simulation [14]. In their paper, the highest rate of churn was only 0.4% per minute. Because we were dealing with fewer nodes, we chose larger rates to demonstrate that ChordReduce could effectively handle a high level of churn.

Our experiments show that for a given problem, ChordReduce can effectively distribute the problem, yielding a substantial speedup. Furthermore, our results showed that the larger the problem is, the more workers could be added before diminishing returns were incurred. During runtime, we experienced multiple instances where *plot* would fail to run and the stager would report socket errors, indicating that it had lost connection with a node in the ring. Despite this turbulence, every node managed to reestablish connection with each other and report back all the data. This further demonstrated that we were able to handle the churn in the network.

VI. Related Work

We have identified two papers that focus on combining P2P concepts with MapReduce. Both papers are similar to our research, but differ in crucial ways, as described below. *One thing we're pretty sure of is that neither utilizes a blocking scheme like CFS [11], but I'll have to check that*

A. P2P-MapReduce

Marozzo et al. [14] investigated the issue of fault tolerance in centralized MapReduce architectures such as Hadoop. They focused on creating a new P2P based MapReduce architecture built on JXTA [16] called P2P-MapReduce. P2P-MapReduce is designed to be more robust at handling node and job failures during execution.

Rather than use a single master node, P2P-MapReduce employs multiple master nodes, each responsible for some job. If one of those master nodes fails, another will be ready as a backup to take its place and manage the slave nodes assigned to that job. This avoids the single point of failure that Hadoop is vulnerable to. Failures of the slave nodes are handled by the master node responsible for it.

Experimental results were gathered via simulation and compared P2P-MapReduce to a centralized framework. Their results showed that while P2P-MapReduce generated an order of magnitude more messages than a centralized approach, the difference rapidly began to shrink at higher rates of churn. When looking at actual amounts of data being passed around the network, the bandwidth required by the centralized approach greatly increased as a function of churn, while the distributed approach again remained relatively static in terms of increased bandwidth usage.

They concluded that P2P-MapReduce would, in general, use more network resources than a centralized approach. However, this was an acceptable cost as the P2P-MapReduce would lose less time from node and job failures [14].

While P2P-MapReduce is decentralized, it still relies on a very definite master/slave hierarchy for organization, computations, and scaling. During simulation, 1% of the entire network was assigned as master nodes. This means for a simulation of 40000 nodes, 400 were required to organize and coordinate jobs, rendering them unable to do any processing. In addition, a loosely-consistent distributed hash table (DHT) such as JXTA can be much slower and fails to maintain the same level of guarantees as an actual DHT, such as Chord [17].

B. MapReduce using Symphony

Lee et al.'s work [18] draws attention to the fact that a P2P network can be much more than a way to distribute files and demonstrates how to accomplish different tasks using Map and Reduce functions over a P2P network. Rather than using Chord, Lee et al. used Symphony [19], another DHT protocol with a ring topology. To run a MapReduce job over the Symphony ring, a node is selected by the user to effectively act as the master. This ad-hoc master then performs a bounded broadcast over a subsection the ring. Each node repeats this broadcast over a subsection of that subsection, resulting in a tree with the first node at the top. Map tasks are disseminated evenly throughout the tree and their results are reduced on the way back up to the ad-hoc master node. This allows the ring to disseminate Map and Reduce tasks without the need for a coordinator responsible for distributing these tasks and keeping track of them, unlike Hadoop.

Their experimental results showed that the latency experienced by a centralized configuration is similar to the latency experienced in a completely distributed framework. However, there are no mechanisms in place to handle churn in the network. If a node joins during a MapReduce job, it will be unable to contribute any of its resources to the problem. If a node in the bounded broadcast tree fails, or worse the ad-hoc master node fails, the data that node is responsible for is lost.

VII. Conclusion and Future Work

We presented ChordReduce, a framework for MapReduce that is completely decentralized, scalable, load balancing, and highly tolerant to churn and node failure at any point in the network. We implemented a fully functional version of ChordReduce and performed detailed experiments to test its performance. These experiments confirmed that ChordReduce is robust and effective. ChordReduce is based on Chord, which is traditionally viewed as a P2P framework for distributing and sharing files. Instead, we demonstrated that it can also be used as a platform for distributed computation. Chord provides $\log_2 n$ connectivity throughout network and has built in mechanisms for handling backup, automatically assigning responsibility, routing, and load balancing.

Using Chord as the middleware for ChordReduce establishes its effectiveness for distributed and concurrent computation. The effectiveness of Chord opens up new approaches for tackling other distributed problems, such as supporting databases and machine learning for Big Data, and exascale computations. We intend to further optimize the performance of ChordReduce and extend the middleware to other applications.

A. *Optimizations*

Caching

Appendix

```
jobid = "pi"

##calculate pi mapreduce job
from map_reduce import Data_Atom
## assume contents is a line of text
import random
def map_func(atom):
    print "map pi"
    jobid = atom.jobid
    resultin = 0L
    total = 0L
    print "running a map"
    points = atom.contents
    random.seed(int(atom.hashkeyID.key,16))
    for p in range(0,points):
        x = random.random()
        y = random.random()
        if (x**2.0)+(y**2.0) <= 1.0:
            resultin+=1
        total+=1
    results = (resultin, total)
    atom = Data_Atom("", atom.hashkeyID, results)
    atom.jobid = jobid
    return atom

def reduce_func(atom1, atom2):
    if atom2.jobid == atom2.jobid:
        jobid = atom2.jobid
    else:
        raise Exception("unmatched jobs in reduce")
    "the form of this is probably wrong"
    a1 = atom1.contents[0]
    a2 = atom2.contents[0]
    b1 = atom1.contents[1]
    b2 = atom2.contents[1]
    results = (a1+a2, b1+b2)
    atom = Data_Atom("", atom1.hashkeyID, results)
    atom.jobid = atom1.jobid
    return atom

def stage():
    samples = 100000000 #midsize run
    jobs = 500
    atoms = []
    last = 0
    for i in range(0,jobs):
        atoms.append(Data_Atom("pi",None,samples/jobs))
    print "DONE STAGE"
    return atoms
```

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] "Hadoop," http://hadoop.apache.org/.

[3] "Virtual hadoop," http://wiki.apache.org/hadoop/Virtual

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[5] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 149–160, August 2001. [Online]. Available: http://doi.acm.org/10.1145/964723.383071

[6] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 165–178.

[7] D. Borthakur, "The Hadoop Distributed File System: Architecture and Design," 2007.

[8] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with mapreduce: a survey," *ACM SIGMOD Record*, vol. 40, no. 4, pp. 11–20, 2012.

[9] K. V. Shvachko, "HDFS Scalability: The Limits to Growth," *login*, vol. 35, no. 2, pp. 6–16, 2010.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[11] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-Area Cooperative Storage with CFS," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 202–215, 2001.

[12] Amazon.com, "Amazon EC2 Instances," http://aws.amazon.com/ec2/instance-types.

[13] A. Rosen, B. Benshoof, and M. Erwin, "ChordReduce," https://github.com/BrendanBenshoof/ChordReduce-Experimental.

[14] F. Marozzo, D. Talia, and P. Trunfio, "P2P-MapReduce: Parallel Data Processing in Dynamic Cloud Environments," *Journal of Computer and System Sciences*, vol. 78, no. 5, pp. 1382–1402, 2012.

[15] H. Shen and C.-Z. Xu, "Locality-Aware and Churn-Resilient Load-Balancing Algorithms in Structured Peer-to-Peer Networks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 6, pp. 849–862, 2007.

[16] L. Gong, "JXTA: A Network Programming Environment," *Internet Computing, IEEE*, vol. 5, no. 3, pp. 88–95, 2001.

[17] C. Nocentini, P. Crescenzi, and L. Lanzi, "Performance Evaluation of a Chord-Based JXTA Implementation," in *Advances in P2P Systems, 2009. AP2PS '09. First International Conference on*, 2009, pp. 7–12.

[18] K. Lee, T. W. Choi, A. Ganguly, D. Wolinsky, P. Boykin, and R. Figueiredo, "Parallel Processing Framework on a P2P System Using Map and Reduce Primitives," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011, pp. 1602–1609.

[19] G. S. Manku, M. Bawa, P. Raghavan *et al.*, "Symphony: Distributed Hashing in a Small World." in *USENIX Symposium on Internet Technologies and Systems*, 2003, p. 10.