

# MapReduce on a Chord Distributed Hash Table

Andrew Rosen

Brendan Benshoof  
Anu G. Bourgeois

Robert W. Harrison

## 1 Introduction

- Background
- Goals

## 2 ChordReduce

- System Architecture
- Chord
- CFS
- MapReduce Module

## 3 Experiments

- Experiments
- Results
- Conclusions

# Background

- Google's MapReduce [1] paradigm is integral to data processing.
- Popular platforms for MapReduce, such as Hadoop [2], are designed to be used in datacenters with a degree of centralization.
- Until recently, analysis and optimization of MapReduce has largely remained constrained to that context.

# Goals

- We wanted build a more abstract system for MapReduce.
- We remove core assumptions [3]:
  - The system is centralized.
  - Processing occurs in a static network.
- The resulting system must be:
  - Fault tolerant.
  - Scalable.
  - Completely decentralized.

# Features of ChordReduce

ChordReduce is a decentralized framework for distributed computing:

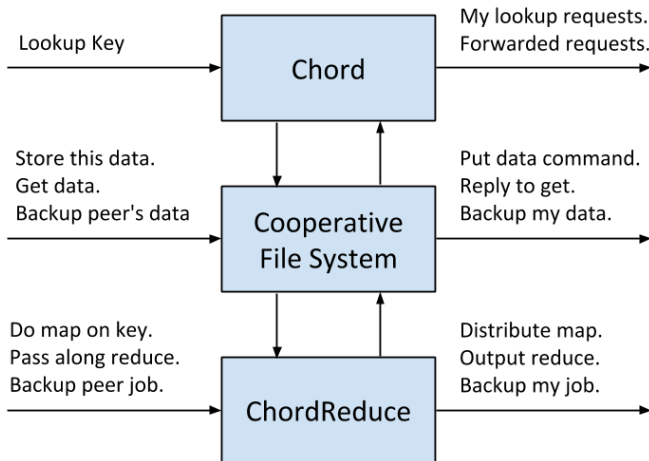
- Scalable.
- Load-Balancing.
  - Data and tasks are evenly distributed across the network.
  - Joining nodes are automatically assigned data and tasks.
- Decentralized:
  - No centralized node is needed to maintain metadata.
  - No central coordinator for tasks.
- Fault tolerant:
  - The loss of multiple nodes does not impact integrity.
  - The network can adjust to churn, the effects of nodes entering and leaving.

# System Architecture

ChordReduce has three layers:

- Chord [4], which handles routing and lookup.
- The Cooperative File System (CFS) [5], which handles storage and data replication.
- The MapReduce layer.

# System Architecture



# Chord

Chord is a peer-2-peer lookup service, where the nodes in the network are arranged in a ring overlay.

- Nodes and files are assigned a  $m$ -bit key.
- Nodes know their predecessors and successors in the ring.
- Nodes are responsible for files with keys between their predecessor's and theirs.
- To speed routing, nodes maintain a table of  $m$  shortcuts, called fingers.
- The fingers allow a high probability  $\log_2 N$  lookup time for any key.



# A Chord Network

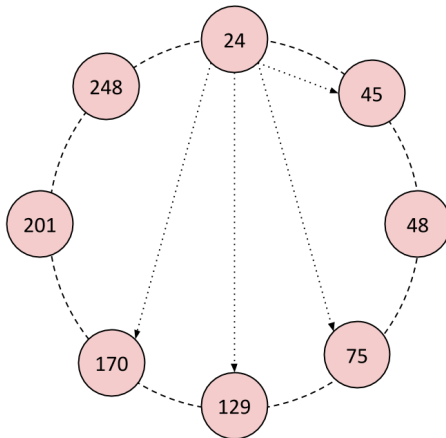


Figure: An 8-node Chord ring where  $m = 8$ . Node 34's fingers are shown.

# CFS

The Cooperative File System runs on top of Chord.

- Files are split up, each block given a key based on their contents.
- Each block is stored according to their key.
- The hashing process guarantees that the keys are distributed near evenly among nodes.
- A keyfile is created and stored where the whole file would have been found.
- To retrieve a file, the node gets the keyfile and sends a request for each block listed in the keyfile.

# Fault Tolerance

- Each node maintains a list of its  $s$  closest successors.
- Nodes back up data they're responsible for to their successors.
- When a node's predecessor fails, the node can immediately take over.
- The network will only lose data if  $s + 1$  successive nodes fail simultaneously.
- The chances of this are  $r^{s+1}$ , where  $r$  is the failure rate.

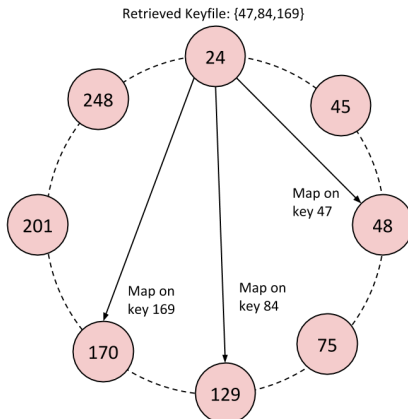
# Starting a MapReduce Job

- Jobs can be started at an arbitrary node, denoted the *stager*.
- The stager retrieves the keyfile and sends a map task for each key.
- This process can be streamlined recursively by bundling keys and sending them to the best finger.
- Once the stager has sent a map to every node, its job is done.

# Data Flow

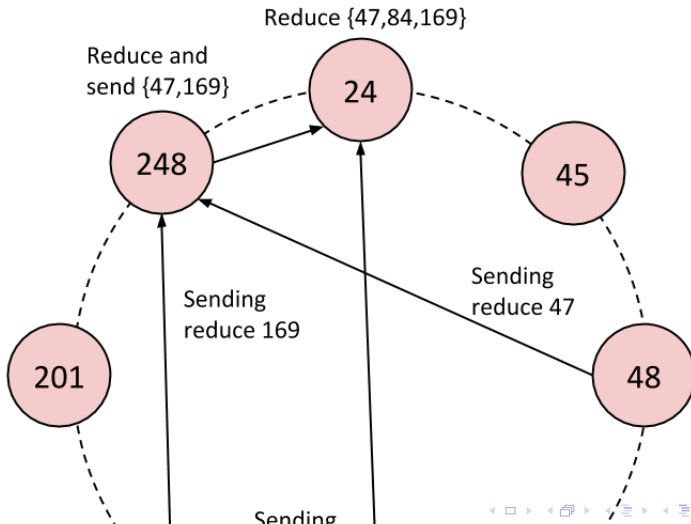
- Results can be sent back via the overlay, or by initiating a direct connection.
- If a node receives multiple reduce results, they are reduced into one before being sent along.

# Data Flow Example



**Figure:** The stager sends a map task for each key in the keyfile. In larger networks, this process is streamlined by recursively bundling keys and sending them to the best finger.

# Data Flow Example



# Fault Tolerance of Map Jobs

- Each node backups their map tasks; removes it when the task is processed.
- If the immediate successor detects the node's failure, it takes over the task.
- If a node detects a new predecessor responsible for a key and map task pair in it's queue, it sends it to the predecessor.
- This allows node to further distribute the work during execution.

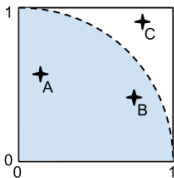


## Fault Tolerance of Reduces

- Individual reduces are backed up in a similar manner; if the original holder of the reduce fails before the reduce is sent, his successor sends his backup.
- Results are sent back to a key, rather than to a specific node.
- This ensures that if node receiving the data fails, his successor will take over.

# Experiment Details

Our initial test was a Monte Carlo approximation of  $\pi$ .



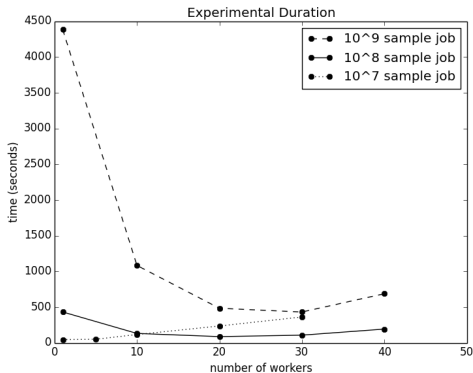
**Figure:** The node chooses random  $x$  and  $y$  between 0 and 1. If  $x^2 + y^2 < 1^2$ , the “dart ” landed inside the circle.

- Map jobs were sent to randomly generated hash addresses.
- The ratio of hits to generated results approximates  $\frac{\pi}{4}$ .
- Reducing the results was a matter of combining the two fields.

# Variables

We ran the experiment using Amazon's Elastic Cloud Compute [6] and varied the following:

- Network size.
- Problem size.
- Rate of churn.



**Figure:** 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.

# Churn Results

| 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:** The results of calculating  $\pi$  by generating  $10^8$  samples under churn. Churn is the chance for each node to join or leave the network. The large speedup is from joining nodes acquiring work during experimental runtime.






# Conclusions

Our experiments established:

- ChordReduce can operate under high rates of churn.
- Execution follows the desired logarithmic speedup.
- Speedup occurs on sufficiently large problem sizes.

This makes ChordReduce an excellent platform for distributed and concurrent programming in cloud and loosely coupled environments.

Questions?

-  J. Dean and S. Ghemawat, “Mapreduce: Simplified Data Processing on Large Clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
-  “Hadoop,” <http://hadoop.apache.org/>.
-  “Virtual hadoop,” <http://wiki.apache.org/hadoop/Virtual>
-  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>
-  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.





Amazon.com, “Amazon EC2 Instances,”  
<http://aws.amazon.com/ec2/instance-types>.