Ishaan Sehgal (netid: isehgal2) • Heesoo Yang (netid: heesooy2) • Asher Dale (netid: asdale2)

**CS425 MP3 Report - MapleJuice: Simplified MapReduce**

We implemented our MapleJuice System using Golang, and for platform-independent messaging we used Google Protocol Buffers. We used our membership service from MP1 for failure detection and also built on top of the SDFS file system from MP2.

Our system has one designated master and other machines are workers. The maple or juice task can be initiated at any machine. This sends an initiation message to the master to kick off the task. We have created a message type in protobuf called MapleJuiceMessage that contains information such as task status, executable filename, and other data. These messages are transmitted using TCP, for reliable data transfer, on a different port than the one for SDFS.

The single master handles a number of coordinating activities. When the master receives an initiation message with the task details, it checks the task message for errors, such as nonexistent input directory. It then performs file splitting (maple only), task partitioning and allocation. For file splitting, the data in the input directory is split (by line) into the user-specified *num_workers* number of files, so that each worker receives an equal amount of work. In our system, each worker corresponds to a single goroutine, so there may be more workers than there are machines. Once the input files are split up, the master sends assignment messages to the workers with the executable and input filenames that the worker can fetch from the SDFS.

The master maintains a map from machineIP to currently-active task numbers (taskMap) so that when a failure occurs, failed tasks can be reassigned to functional machines. Failures are detected using the membership service from MP1. Also, we assign/re-assign tasks to machines that have the fewest number of tasks for load-balancing. When a task is completed, we remove this task from our taskMap and append the result to the appropriate output file(s).

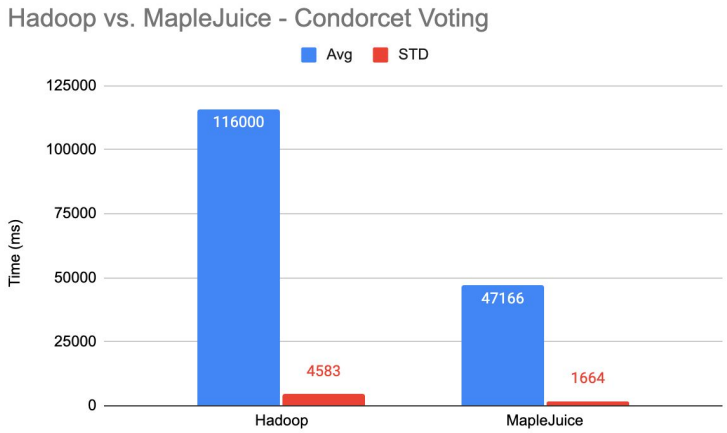
When a machine receives a task assignment from the master, it uses the information in the message to fetch the maple/juice executable and the intermediate file associated with the task (from SDFS) to the machine's local directory. Then, a new goroutine is created to run the actual task/executable. The executable runs repeatedly on 100-line segments of the input file.

The outputs from the executable are captured from STDOUT and stored in the worker's memory as a key-value dictionary. Rather than issuing SDFS commands directly, the worker sends the resulting k-v dictionary to the master for processing. The master will receive task completion messages with the resultant k-v dictionary and write the dictionary to the appropriate output files using SDFS. To reduce memory load, we store and send the dictionary in segments over multiple completion messages and use a flag to denote whether a message contains the last dictionary segment. By having all the results go through the master and only committing the result dictionaries to the SDFS only when the full dictionary is received, we simplify failure handling--since we don't have to worry about figuring out which portion of a task was processed and which have not yet. Instead, we can simply fully restart the entire task.

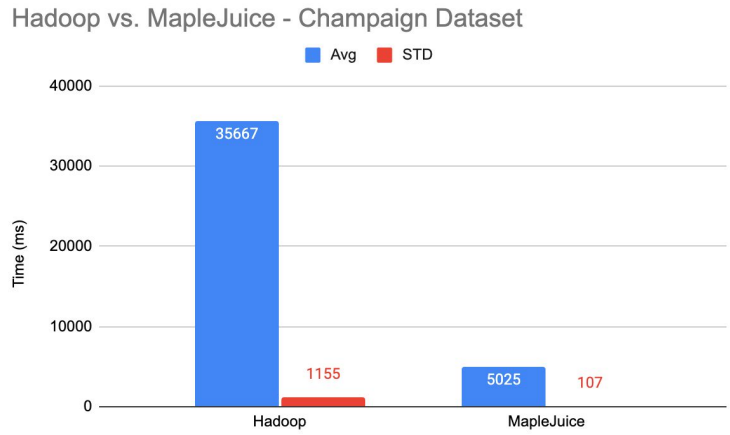
**Applications and comparison versus Hadoop**

For our two applications, we chose the condorcet voting application from HW1 and also used a residential information dataset from the Champaign map database. For the condorcet application, our mapreduce program has 3 consecutive MapReduce stages. We generated our own condorcet dataset & ground truth evaluator using python scripts, included in the repo. For the Champaign dataset, the task is to count the number of residential, mailable and active addresses grouped by zip code. This can be done in a single MapReduce stage. For our system, the MapleJuice programs were programmed in Golang, the same as the system itself, and the output to STDOUT is captured. For the below comparisons, we ran both Hadoop and

our system using 10 worker nodes. The condorcet dataset was 10 million lines (~130MB) and the Champaign dataset was 100,000s of lines (~20MB).




Here, we have summed the runtime for all 3 mapreduce stages and calculated avg & std statistics across 3 trial runs. Though not reflected in the graph, the biggest improvements over Hadoop were in the 2nd and 3rd stages, which use a way smaller input compared to the original dataset used for the 1st stage. This suggests that the speedup is largely due to Hadoop's slow ramp-up time for task allocation. We noticed that due to the small size of the dataset (it fit on only a few HDFS blocks), the YARN scheduler did not always use all nodes in the 10 machine cluster. When we changed the HDFS block size so all 10 worker nodes would be utilized, we found the runtime to be even slower. The 2nd and 3rd stage inputs are only a few dozen bytes, so our system's much improved speed in these stages indicates our system is more lightweight. We suspect that with a much larger dataset, Hadoop would perform better.




For the Champaign dataset, we see an even greater speedup for our MapleJuice system compared to Hadoop. Once again, this highlights the lightweight nature of our system, especially on smaller inputs. Even with small inputs, Hadoop has considerable startup & task allocation times that take up a lot more time, whereas our system can get to work more quickly.