CS425 MP4 Report

Taipeng Liu(taipeng2) & Xiang Li(xiangl14)

1. Design

Algorithm: Our "MapleJuice" system is a MapReduce-like parallel computing framework, which is built on top of reliable Distributed File System and full-membership failure-detection module. We borrow all names (i.e., RM, AM, namenode, datanode) from Hadoop and they do quite the same works, and implement RM-AM model, a cluster will have one server working as the resource manager (RM), which is responsible to Maple & Juice task scheduling, distribution and tracking. Typically, the server running RM also works as namenode in SDFS. All datanodes act as application managers (AMs), which undertake tasks and submit results to SDFS for next phase. As before, we use Go for this MP and RPC API to distribute each task from RM to AMs.

Inside MapleJuice: In Maple phase, we use Go channel to schedule tasks. Each AM will be assigned a fair amount of work (file number) to run user-specific executable, which outputs a series of key-value pairs. We'll cache the output on local disk without put them into SDFS, which will be a huge amount of overhead. RM maintains crucial data structure about the intermediate files on each Mapler and assign Juice tasks to each Juicer. In Juice phase, each AM fetches intermediate files from remote local disks and append to one single result file in SDFS. We realize hash & range partition based on the available working nodes' number, which is user-selectable. Also, user can decide whether to delete the input files.

How our MapleJuice works: First, client/user should put their executables into SDFS, along with the dataset. Then client will RPC call RM to perform Maple function for the input data. After maple finishes, intermediate results will be stored separately on the cluster. In Juice phase, client will RPC call RM to shuffle tasks based on key and assign tasks. After all tasks are finished and returned, the client will be notified.

RM is critical since it assigns and monitors each task's process, also records each working nodes' task list. Also, it should handle the failure situation and reassign failed tasks. Failure could happen anytime. First, we must make sure the result remains intact. Hence, if a node fails during writing the result, we should get rid of this round and restart juice phase. Besides, namenode will reassign failed tasks to idle nodes. Our protocol can tolerate namenode's failure too.

How to deal with failure: We implement a task keeper in both map phase and reduce phase. The task keeper is responsible for checking all tasks are finished. If not, for example, a datanode fails when running a map task, the task keeper will check out a working map which records all nodes and their received tasks. Then task keeper will re-distribute to other alive nodes all tasks which are distributed to the failed node. Task keeper returns when all tasks are done.

2. Relation with previous work

We make full use of MP3's SDFS as we need frequent distributed file access in this application. Besides, all new feature highly replies on MP2's reliable failure detection protocol as it maintains an updated membership-list. Of course, MP1's remote grep is useful for debug.

3. Performance Measurement

a. Application: Word Count

Dataset: We used Gutenberg's book datasets and split it to 30s/60s/90s of 1MB file.

Num of node	MapleJuice						Hadoop					
	30MB		60MB		90MB		30MB		60MB		90MB	
	Avg.(s)	SD.(s)	Avg.(s)	SD.(s)	Avg.(s)	SD.(s)	Avg.(s)	SD.(s)	Avg.(s)	SD.(s)	Avg.(s)	SD.(s)
5	244	7.81025	497	10.14889	899.6667	43.36281	45.54533	1.507747	71.81933	3.654937	82.05267	8.420517
10	161.6667	14.84363	307.6667	20.74448	579.6667	7.505553	24.46	1.665975	43.086	4.699974	78.96167	6.183115

Table1. WordCount RunTime (MapleJuice vs Hadoop)

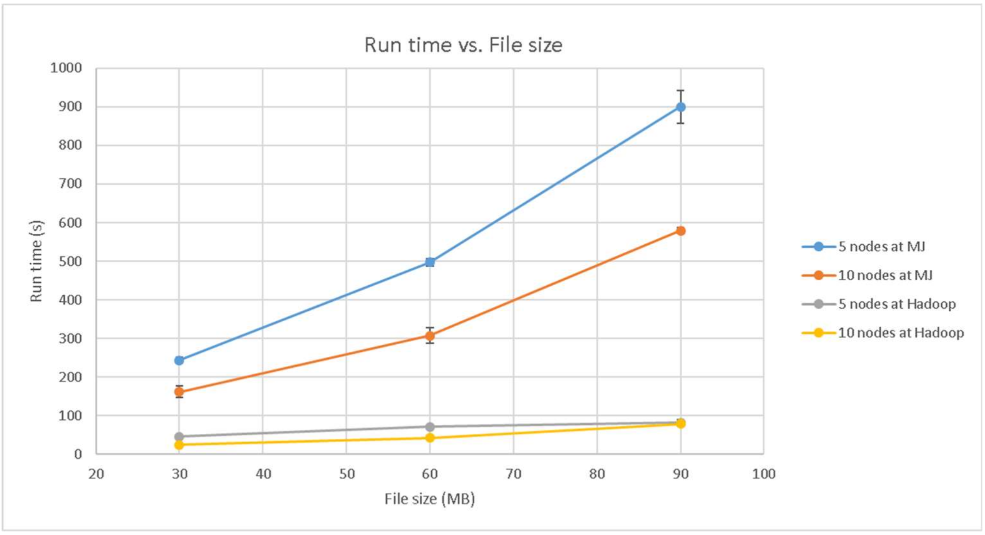

Figure1. Average RunTime (WordCount)

b. Application: Reverse Web-Link

Dataset: We used large directed social-network datasets on SNAP and split it to 30s/60s/90s of 1 MB file.

Num of node	MapleJuice						Hadoop					
	30MB		60MB		90MB		30MB		60MB		90MB	
	Avg.(s)	SD.(s)	Avg.(s)	SD.(s)	Avg.(s)	SD.(s)	Avg.(s)	SD.(s)	Avg.(s)	SD.(s)	Avg.(s)	SD.(s)
5	268	26.21068	522.3333	46.2313	952.3333	36.22614	47.38333	2.062445	78.53267	4.633579	89.52267	6.190223
10	167.3333	9.609024	309.6667	47.81562	606	17.34935	24.576	4.316544	44.71767	3.116057	78.87067	8.125423

Table2. WebLink RunTime (MapleJuice vs Hadoop)

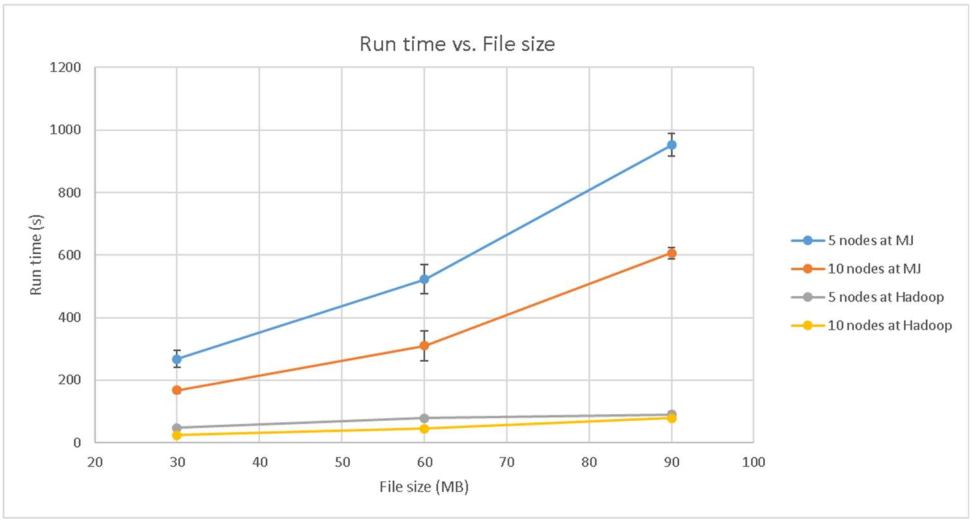

Figure2. Average RunTime (WebLink)

c. Analysis

Overall, our performances are not good enough in larger dataset (over 50MB file). From our analysis, the significant overhead comes from the access to the intermediate files (through RPC to remote disk) and key-partition in RM due to the huge amount of keys. On the other hand, we tested our application on 5 & 10 working nodes. The result meets our expectation, more working nodes have better parallelism performance.

One more important factor which can significantly affects our result is the number of replicas. We reuse the code in MP3, so the system will keep 4 replicas of each sdfs file. But in Hadoop configuration, we set replica equal to 1. It is expected that Hadoop runs faster than our system, because for each write operation, we write to more 3 files and wait for all of them return.

As the plot shows, Hadoop's average completion time per MB decreases with larger dataset while our implementation suffers from the increment of dataset.