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 implement RM-AM model, a cluster will have one server working as the global resource manager (RM), in charge of Maple & Juice task scheduling and tracking. Specifically, we pick the same server as namenode in SDFS. All datanodes will act as application manager (AM), undertaking tasks and submitting results to SDFS for next phase. Basically, we use RPC to execute each task from RM to AMs.

In Maple phase, we use Go channel to schedule tasks. In Juice phase, 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.

**Inside MapleJuice**: In Maple phase, 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.

**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.

**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**

1. Application: WordCount

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

2. Application: Reverse Web-Link

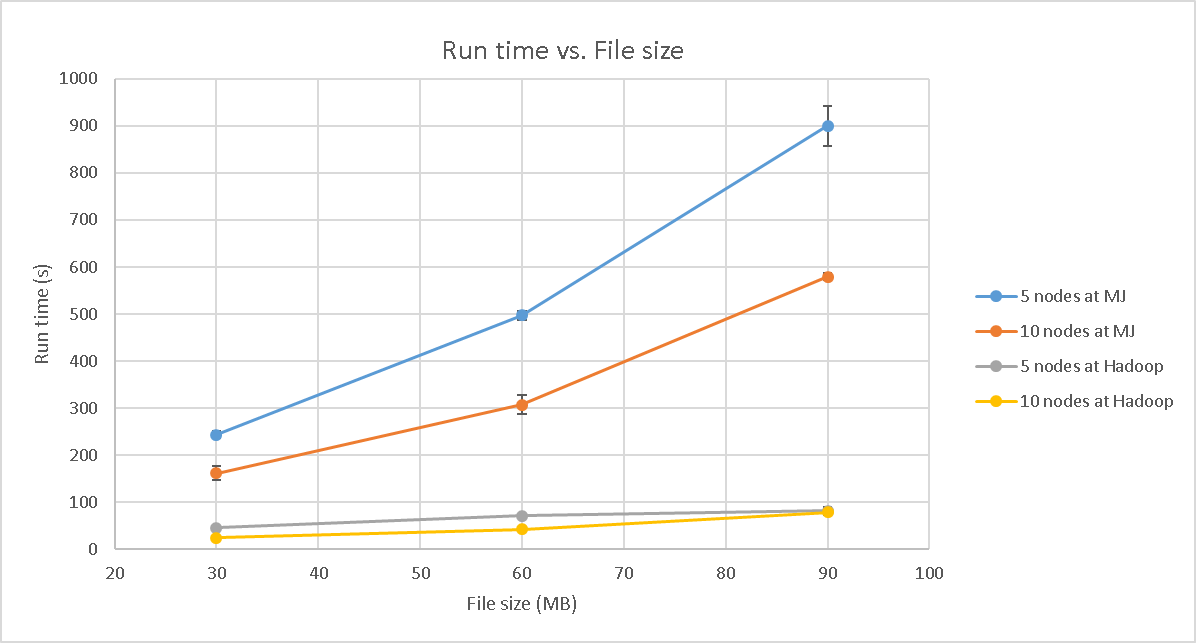
Dataset: We used large directed social-network datasets on SNAP and split it to 30s/60s/90s of 1 MB file.

3. Result Analysis (Comparison with Hadoop)

On average, our system performs not good enough in larger dataset (over 50MB file). By our analysis, the significant overhead comes from the access to the intermediate files (thru 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.



**Table1. WordCount RunTime (MapleJuice vs Hadoop)**

****

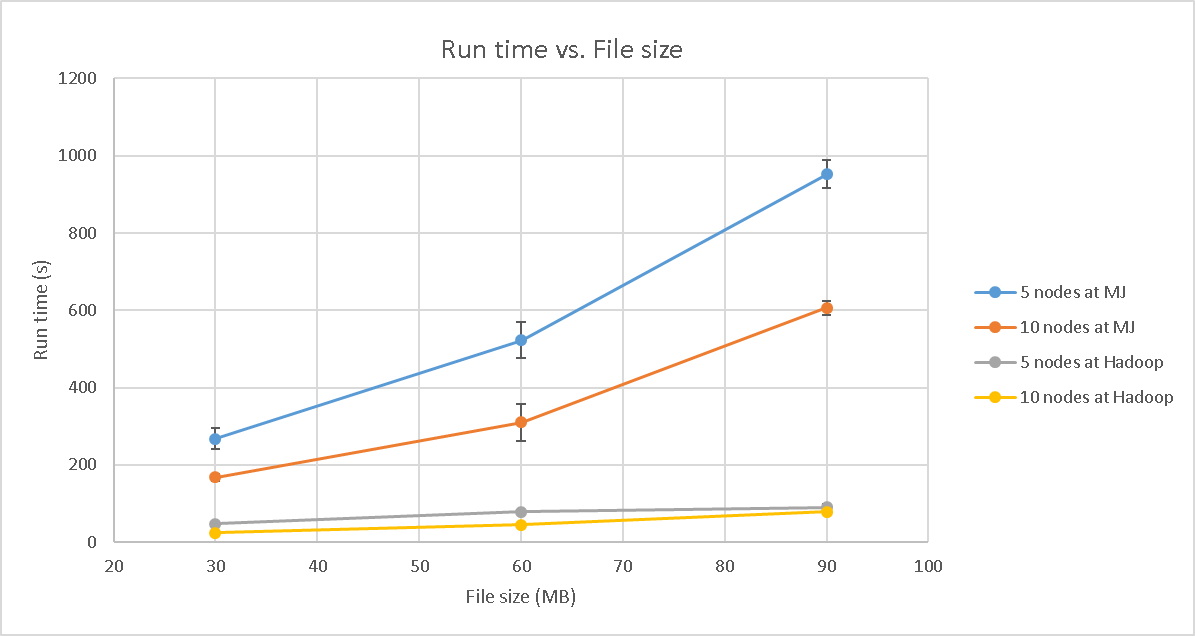
**Figure1. Average RunTime (WordCount)**

Our MapleJuice system. In our SDFS, the replication factor is set to 4, which .

As the plot shows, Hadoop’s average completion time per MB decreases with larger dataset while our implementation suffers from the increment of dataset. The communication protocol of our framework highly relies on Remote Procedure Call, which



**Table2. WebLink RunTime (MapleJuice vs Hadoop)**



**Figure2. Average RunTime (WebLink)**