Samuel Cheung
Wentao Yang

Maplejuice (MapReduce)

Design Description

Our MapleJuice design utilizes the past two MPs to perform MapReduce operations and handle failure during computation. It is also much simpler than MapReduce (doesn't handle stragglers, rack awareness capabilities, extensive logging of everything, etc.). We do not support master failure during execution. We use Golang for easier concurrency support.

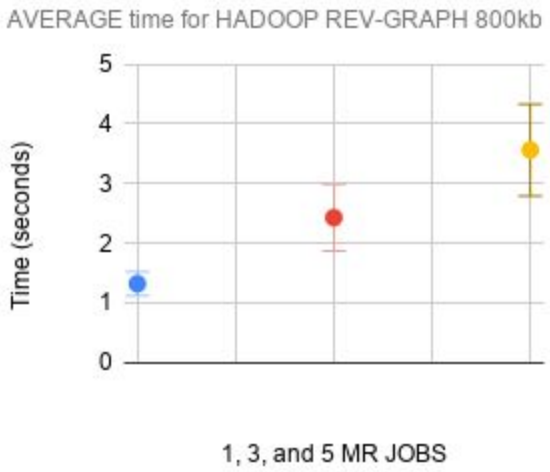
For maple, our programming has the following step. First, a node gets a maple request from standard input and the request is sent to the master. The master would look for all files in the input directory in SDFS (in our case the files with a common prefix since our SDFS only has one directory). Then, it would assign the files evenly based on the number mapper (which is assumed to be less than the number of machines) and send out maple commands to them including required filenames. When the node gets there commands, they would get the files they need from SDFS to local and apply executable (assume to be local to the node) to each of the files. The output of executable would be classified according to keys and sent back to the master. Then the master would append the outputs to local intermediate key files. When all nodes complete (the master records it after receiving outputs), the master would upload the key files to the SDFS (i.e. distribute files to other nodes).

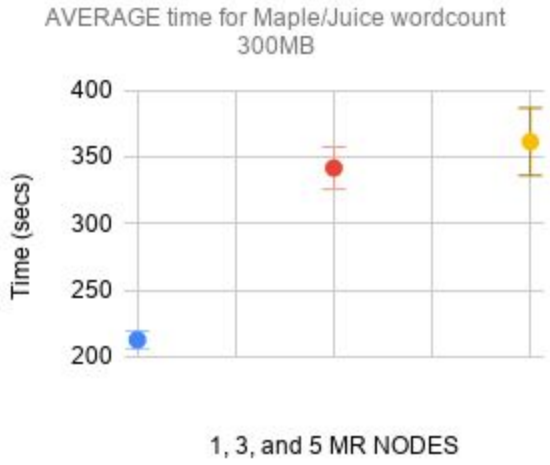
For error handling, since we assume the master does not fail, whenever a node fails, the master would know (from error detector) and restart the maple task on a new node if it has not finished the job. If the node fails during sending outputs to master, the master would discard the incomplete outputs.

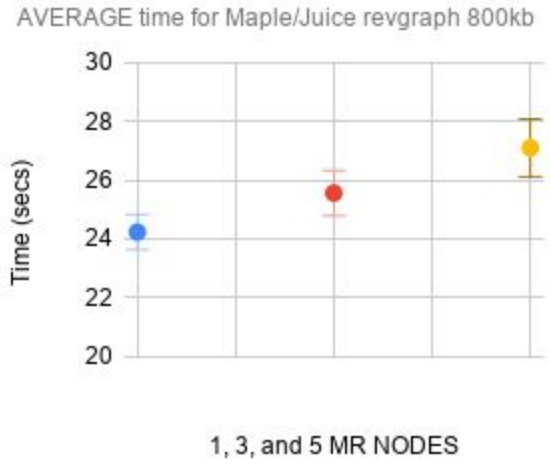
During the Juice phase, the intermediate files are already present in the SDFS for processing. The JUICE command specifies the executable logic performed on the intermediate keys by a specified number of juice nodes. We have the option to do range based or hash-based partitioning of the jobs to the nodes. During the JUICE execution, the master receives the command, partitions the key files to nodes, and keeps track of whether or not a node has completed or not in list of strings. The latter part is important for failure detection and execution timing (for results/plots). If a node gets a job for a file it doesn't have it requests the file using the GET command (holds even if a node receives a job after a failure). If after a failure a node receives a job for a file it is going to get because of replication, it waits until that file replicated before doing the JUICE. If the master detects that a node fails then it retransmits that nodes job to another node depending on whether or not that node has a job (or completed one). After a node is finished processing all of its files it sends back the complete intermediate results for that

node to the master. The master is responsible for combining all intermediate results and storing it in SDFS under the specified destination filename.

Plots/Results







Discussion

Here we will discuss the plots and compare our results to Hadoop. We will briefly describe our Hadoop setup and then compare and contrast the two models. We perform wordcount and reverse graph on an Amazon review dataset which is 300 MB and a chemical network dataset of 800 kB. The set of graphs on the right is for the reverse link graph and the left is for wordcount. We test 1, 3, 5 nodes because it is difficult and time-consuming to setup Hadoop on a machine.

Decreasing the minimum split size we obtain results for a Hadoop clusters of size 1, 3, and 5 on our given VMs. I allocate 1.5 GB of memory per node and replicated files four times. We used a 300MB file to test since it takes around a minute. With one node. This setup leads to

some poor performance with a small file and multiple jobs. But when it's optimized to split data for jobs around 64 Mb to 128 Mb (which we use for our tests) it is significantly faster (5 to 10 times faster). We also allow for reduce to start processing while map is processing. The first set of graphs belong to our Hadoop runs/jobs/tests. The two notable results are the increasing variance as we use more nodes, which we suspect it is because of the relatively more unstable networking with more nodes, and increasing average time taken as we increase the number of nodes which is counter-intuitive and we suspect it is because of increasing transmission cost and duplication cost with more nodes.

Comparing two applications, we can see that the IO costs more than the actual computation because as we drastically increase the task size, there is only a disproportional increase for the time taken since the number of keys(which affects IO cost) does not increase proportionally with task size.

Unfortunately, our map-reduce is a lot slower than Hadoop, mainly because of the IO cost, as everything goes through the SDFS and we are doing duplication and distribution for the SDFS it costs a lot. Also, we admit that we have not fully utilized the concurrency of machines, as a large portion of data processing happens in the master node because we want to ensure the system can handle failure and the map and reduce happens sequentially. Overall, our map-reduce graphs behave similarly to the Hadoop graph, and we can also see we spend more time as we use more nodes because of the increasing transmission and duplication cost, and there are increasing variance in our data primarily because of relatively more unstable TCP transmission with more nodes.