October 29, 2010

SnowFlock-Aware Hadoop Implementation

(http://www.cs.toronto.edu/~mjulia/CSC2231Project/SnowFlockAwareHadoop.html)

Progress Report I: Shahan Khatchadourian and Julia Rubin

Using the help of the SnowFlock research group, we were able to activate SnowFlock and experiment with its execution environment on a cluster of two physical machines that were allocated for us in the systems lab. One of the machines is used as a master and can optionally host clones; the other machine is used for hosting clones only. We were able to run SnowFlock and spawn clones on both these machines.

We also explored several stages in which Hadoop's MapReduce implementation can be extended with the SnowFlock's ability to dynamically spawn new virtual machines. Conceptually, a MapReduce framework has a sequential processing model consisting of the following four stages: (1) Record Reader, (2) Map, (3) Reduce, and (4) Record Writer. We found the following three scenarios of clones' instantiation and destruction most reasonable:

1. *Create clones before the Record Reader stage (e.g., for each input split); destroy them after the Map stage.* HDFS is optimized for large files by using large block sizes, with the default block size of 64MB. Thus, when input splits size is smaller than a block size, blocks can contain more than one input split each. Since each Record Reader processes one input split, there might be cases that several Record Readers read from the same HDFS block. To leverage disk I/O and network utilization when the data is not local, clones can be created after the block has been read into memory, which will allow several Record Readers to work on the same area in memory. Created clones should not be destroyed until after the Map step serializes its output to HDFS, otherwise the output will be lost.

2. *Create clones before the Map stage; destroy them after the Map stage.* Each Map task is associated with a Record Reader that pipes read key-value tuples to the Map. The number of tuples produced by a Record Reader is unknown in advance. However, the number of Maps tasks in a Hadoop cluster is pre-configured and, at runtime, depends on the number of input splits. The Map task processes generated tuples sequentially, by passing each to the user-define Map function. Thus, the elasticity of the Hadoop cluster can be improved if for each input split the generated tuples are batched for processing by cloned Map machines. For example, if a Record Reducer produces one output tuple only, one Map machine is needed. If a Record Reducer produces eight tuples, then two Map machines can be cloned, each processing four tuples. The Map machines can then be destroyed after their outputs are collected by the original Map machine.

3. *Create clones before the Reduce stage; destroy them after the Record Writer stage.* The reduce step is typically used to perform aggregation over tuples having the same key, but the number of keys is unknown in advance, while the number of Reduce tasks in the Hadoop cluster is pre-configured. To improve elasticity, once the number of keys is discovered at runtime, Reduce machines can be cloned, each handling a subset of keys. Created clones should not be destroyed until after the Record Writer step serializes its output, otherwise the output will be lost.

Due to time constraints, we plan to implement the second and third scenarios as part of this project. We consider the first and, possibly, additional scenarios as a future work. Our implementation shall also preserve fault-tolerance behavior of the MapReduce framework. That is, if one of the clones, or the original task, dies, the outputs from other corresponding clones should not be used by the framework and the cluster's JobTracker should reassign that portion of work.

In addition, we discussed the applications and workloads to be used for performance measurements. We consider using canonical MapReduce applications, such as WordCount and Grep as our benchmarking use-cases. We plan to use Wikipedia XML data as input to those applications.

As our next steps, we plan to:

1. Install Hadoop on a SnowFlock-based virtualized environment.
2. Implement a Java wrapper for the SnowFlock Python or C APIs.
3. Implement the extensions to the Apache Hadoop code. We plan to extend the following Hadoop classes: *MapRunnable, MultiThreadedMapRunner, CopyFilesMapper, Fetcher, ShuffleManager, MergeManager, Shuffle, Merge, TaskInProgress, JobConf*, and *Task*.
4. Execute performance tests and analyze the results.