Cutting MapReduce Cost with Spot Market

Huan Liu
Accenture Technology Labs
huan.liu@accenture.com

Abstract

Spot market provides the ideal mechanism to leverage idle CPU resources and smooth out the computation demands. Unfortunately, few applications can take advantage of spot market because they cannot handle sudden terminations. We describe Spot Cloud MapReduce, the first MapReduce implementation that can fully take advantage of a spot market. Even if a massive number of nodes are terminated regularly due to a price increase, Spot Cloud MapReduce can still make computation progress. We show experimentally that it performs well and it has very little overhead.

1 Introduction

An infrastructure cloud, such as Amazon EC2, can make a more efficient use of its infrastructure by aggregating many computation demands from many users. Through statistical multiplexing, the peak and trough from individual users' computation demands could be potentially smoothed out, resulting in a much higher average utilization of the infrastructure.

Unfortunately, there is a limit on what can be achieved with statistical multiplexing alone. Without the ability to shift user demands, the total computation requests could still peak when the demands are high. For example, during the Christmas shopping season, all retailers would be requesting computation capacity. Without an incentive to shift their computation demands, all requests would pile up at the same time. Figure 1 shows the estimated number of instances ("instance" is Amazon's terminology for a virtual machine) launched daily in EC2 between 2007 and 2010 [5]. There are two distinct peaks around Feb. 2010 and Oct. 2010. Even though we do not know how many instances are running daily (because we do not know how many instances are turned off daily), the graph is an indication that the computation demands could change dramatically.

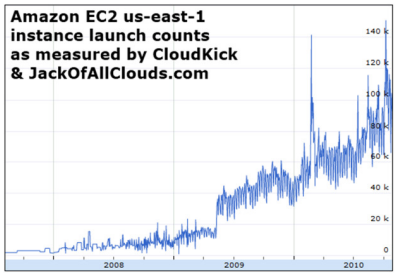

Figure 1: Estimated number of instances launched daily in EC2 between 2007 and 2010.

If we can shift computation demands, there is a greater potential to smooth out the computation requests, leading to a more efficient use of the infrastructure. Although it is difficult to shift real-time applications, such as web applications which serve real-time user demands, other applications, such as batch applications, are more elastic in nature. A batch application, such as a MapReduce[3] computation, often requires the results by a certain deadline, and it could be flexible on when the capacity is provisioned.

A market mechanism is one promising approach to facilitate demand shifting. A cloud provider expresses the current utilization through a price signal. A higher price asks users to hold off their requests if they can wait. Conversely, a lower price attracts more usage by giving the users a price incentive. Amazon EC2 introduced such a mechanism, called spot market, in Dec. 2009. The spot market offers a much lower price in exchange for the ability to turn off an instance anytime when the market price is above the bid price.

Although many MapReduce implementations, such as Hadoop[6], can tolerate failures, they are ill-suited to the spot market environment. They are designed to tolerate infrequent machine failures, but they cannot cope with massive machine terminations caused by a spot price increase. If the primary and backup master nodes fail, no

computation can progress. Even if the master nodes do not run on spot instances, several simultaneous node terminations could cause all replicas of a data item to be lost. In addition to data corruption, it is also shown that adding spot instances to a MapReduce computation can lengthen the computation time [1].

In this paper, we describe Spot Cloud MapReduce (Spot CMR), a MapReduce implementation that works well in a spot market environment. To the best of our knowledge, it is the first MapReduce implementation that could tolerate massive node terminations as could be induced in a spot market. It streams intermediate results to a cloud storage while processing a map task, and when the instance is terminated, it uses the short time available in the shutdown process to flush the buffer and commit the partial results. Thus, it is able to keep on making computation progress even when the instances are constantly turned off.

2 Cloud MapReduce

Spot Cloud MapReduce is built on top of Cloud MapReduce (CMR)[7], but it extends CMR so that it works well in a spot market. Before we describe the changes we add to Cloud MapReduce to make it tolerate large-scale terminations, we first describe the Cloud MapReduce implementation at a high level. For more details, we refer interested readers to the Cloud MapReduce paper[7].

Cloud MapReduce uses various cloud services, such as Amazon S3, SQS, SimpleDB, which greatly simplifies our design. Cloud MapReduce architecture is shown in Fig. 2. There are several SQS queues: one input queue, one master reduce queue, one output queue and many reduce queues.

At the start of a MapReduce job, CMR partitions the input data into M splits, where each split will be processed by a separate map task. CMR enqueues a split message for each split in the input queue. Typically the input data is stored in Amazon S3, and a split message is simply a list of pointers to files in S3, possibly with an associated range to specify a subset of a large file. To facilitate tracking, each split message also has a unique map ID. While the input queue holds the input, the output queue holds the results of the MapReduce computation, i.e., the resulting key-value pairs.

There is only one master reduce queue and it holds many pointers, one for each reduce queue. Similarly to the input queue, which is used to assign map tasks, the master reduce queue is used to assign reduce tasks. There are a large number of reduce queues. The number of them, denoted by R, is a configurable parameter that is set by the user. The reduce queues and the master reduce queue, as well as the entries in the master reduce queue,

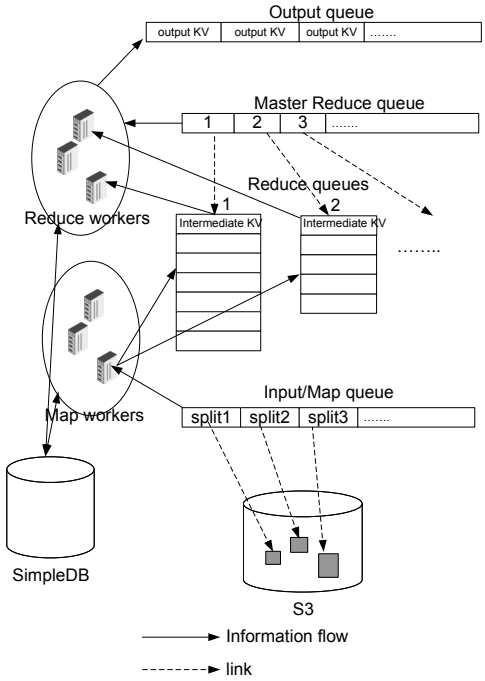

Figure 2: Cloud MapReduce architecture

are created distributedly before the start of the MapReduce job.

A set of map workers, each runs as a separate thread on an EC2 instance, poll the input queue for work. Amazon SQS automatically hides a split message after dequeuing it. The map worker removes the split message from the queue after it has successfully processed the split; otherwise (i.e., when the map worker failed), the message would automatically reappear after a visibility timeout. After a map worker dequeues a split, it parses the input split for key-value pairs, and it invokes the user-defined map function for each key-value pair. While processing an input key-value pair, the user-defined function generates and emits a set of intermediate key-value pairs.

The MapReduce framework collects the intermediate key-value pairs from the map function, then writes them to the reduce queues. A reduce key maps to one of the reduce queues through a hash function. A default hash function is provided, but the users could also supply their own.

CMR uses the network to transfer the intermediate key-value pairs as soon as they are available, thus it overlaps data shuffling with map processing. Overlapping shuffling is used when pipelining MapReduce [2]. Compared to the implementation in [2] where it has to implement pairwise socket connections and buffering/copying mechanism, our implementation using queues is much simpler. Since the map phase is typically long, overlap shuffling has the effect of spreading out traffic. This

can help alleviate the incast problem [8] [4] (switch buffer overflow caused by simultaneous transfer of a large amount of data) if it occurs. In addition, CMR's strategy of immediately writing the intermediate results to the queues is a key enabler for us to tolerate large-scale terminations as could be induced by the spot market. When an instance terminates, there is a minimal amount of intermediate results remaining locally, and we can quickly flush out of the buffer while the instance is in the shutdown procedure. In contrast, other MapReduce implementations, such as Hadoop[6], hold all intermediate results locally until the task finishes. If an instance is terminated, there is not enough time to save the intermediate results.

Once the map workers finish their jobs, the reduce workers start to poll work from the master reduce queue. Once a reduce worker dequeues a message, it is responsible for processing all data in the reduce queue indicated by the message. It dequeues messages from the reduce queue and feeds them into the user-defined reduce function as an iterator. Just like in other MapReduce implementations, the user-defined reduce function writes a set of key-value pairs as the outputs. The reduce workers collect the outputs and write them to the output queue. The output queue can be used either as the final output or as the input to the next MapReduce job.

Besides reading from and writing to the various queues, the workers also read from and write to SimpleDB to update their status. At the end of either a map or reduce task, the workers write a commit message to SimpleDB indicating they have successfully processed the task. For example, upon successful completion, the map task commits a (n, m) pair into SimpleDB to indicate that node n has successfully finished processing map split m. These commit messages are used to determine if a phase has finished. If there is at least one commit message for each map (reduce) task, then we know the map (reduce) stage has finished.

3 Spot Cloud MapReduce

In this section, we describe Spot Cloud MapReduce, a MapReduce implementation that works well in a spot market environment. A spot market presents a more severe failure scenario than the typical failure scenarios a MapReduce implementation is designed for, because many or all nodes could be terminated at the same time. Fortunately, a spot market termination is more graceful than a hardware failure. In a hardware failure, the whole node may go down suddenly without warning. In contrast, in the Amazon spot market, a price increase beyond the bid price would induce a graceful shutdown where the whole shutdown scripts are executed.

Spot Cloud MapReduce is built on top of CMR. We make four changes on top of the CMR implementation. First, we modify the split message format in the input queues. For each split, in addition to the map task id m and the corresponding file location, we also add an offset f, which indicates the position in the file split where we should start processing. At the beginning of the job, when the input queue is created, all split messages enqueued have an offset of 0.

The second change allows us to save the intermediate work when a node is terminated. We take advantage of the graceful termination in the spot market. When a shutdown request is received, the shutdown scripts (e.g., /etc/rc0.d on some Linux distributions) on the host OS are executed. We modify the shutdown scripts. When they are invoked, they first issue a SIGTERM signal to the MapReduce process so that the MapReduce process can saves its states as necessary, then they go to sleep, never executing the rest of the shutdown scripts. In our test, the Amazon hypervisor waits for the shutdown scripts to finish (issuing the halt instruction at the end), and it waits for up to 2 minutes. If the shutdown process still has not finished in that time window, the hypervisor forcefully terminates the instance by simulating a power off event. The shutdown window is too short for other MapReduce implementations to save all intermediate data. However, it is more than enough for Spot CMR since it transfers data to the queues as soon as possible, and it typically has very little data left in its buffer that needs to be flushed.

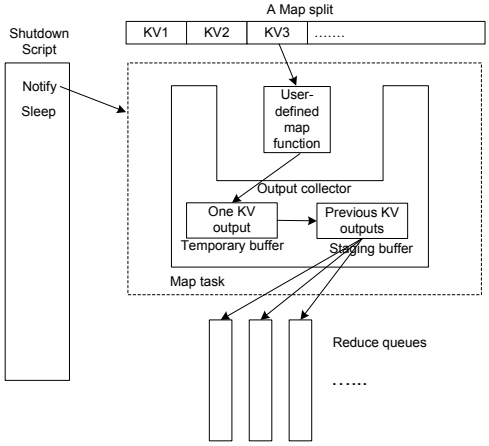

Figure 3: Saving intermediate data when a node is terminated.

Fig. 3 shows how Spot CMR processes a map task. Each map task is given a map split, and the map task parses the map split and passes each key-value pair to the user-defined map function in turn. In addition to passing an input key-value pair, the map task also passes an output collector to the user-defined map function for it to write its output key-value pairs to. Unlike the CMR im-

plementation, the output from one input key-value pair is first saved in a temporary buffer. When the user-defined map function finishes the processing of one key-value pair, its output is appended to a staging buffer which is then asynchronously uploaded to the reduce queues. Streaming output data – sending data from the staging buffer to the reduce queues as soon as possible – is a unique feature of the CMR implementation. Typically the staging buffer holds very little data, making it possible to flush out the data during the shutdown process. In contrast, other MapReduce implementations hold all outputs from a map split locally, and they only upload the outputs after the map split has completed successfully. Streaming output data is possible in CMR because CMR uses filtering at the reduce side to remove duplicate or invalid outputs (e.g., from failed nodes).

When the map process receives a SIGTERM signal from the shutdown scripts, it immediately kills the current user-defined map function, and it throws away the partial outputs in the temporary buffer. The process then begins to flush the staging buffer. In our various tests, we found that in the shutdown time window, at a minimum, we can flush out tens of megabytes of data. Since the amount of data in the staging buffer is typically small (smaller than a few KB), it is enough time to guarantee that we can flush out the buffer.

The third change we make to CMR is to change the commit mechanism to allow a partial commit. We use the following steps in the commit process.

1. Wait for the staging buffer to be successfully flushed.
2. Send a commit message (n, m, f_i, f_{i+1}) to SimpleDB, where n is the node id, m is the map task split, f_i is the beginning offset when this map task started, and f_{i+1} is the offset when the map process received the SIGTERM signal. f_{i+1} indicates the position from where the next map task should resume processing. In Fig. 3, $f_{i+1} = 3$ to indicate that processing should resume from KV3.
3. Add a map split message (m, F_m, f_{i+1}) to the input queue, where m is the map task id, F_m is the original input split, and f_{i+1} is the resume offset.
4. Delete the map split message (m, F_m, f_i) from the input queue.

A node may fail anywhere between the steps, but we can still guarantee the correct processing. Failing between step 1 and 2 invalidates the flushed partial results. Since there is not a commit message, the reduce processes would ignore those partial results. Failing between step 2 and 3 would cause duplicate processing. After a visibility timeout (Amazon SQS's feature to tolerate node failures), the map split message (m, F_m, f_i) would reappear in the input queue and another node would reprocess it. Finally, failing between step 3 and 4 adds an additional redundant split (m, F_m, f_{i+1}), since the old split (m, F_m, f_i) would be reprocessed again.

The last change we make is to change how we determine if there is a successful commit for a map split. Since there could be multiple partial commits, we must make sure that we can find a set of partial commits that cover the whole range. In other words, we need to find a set of commit messages (n_0, m, f_0, f_1), (n_1, m, f_1, f_2), ..., (n_i, m, f_i, f_{i+1}), where $f_0 = 0$ and f_{i+1} is one more than the last key-value pair's offset. Once the set of partial commits are found, the reduce workers use them to filter valid intermediate results from the reduce queues.

Our modifications essentially enable Spot CMR to process the map inputs at a finer key-value pair granularity instead of at the granularity of an input split. In addition, the boundary between two consecutive map processing of one input split is dynamically determined depending on when a node is terminated. Also, there is no need to periodically save the results and roll back when node fails as other checkpoint mechanisms require [9], so the overhead is very low.

Our modifications currently only apply to the map tasks. There is no easy extension to the reduce tasks due to a limitation in Amazon. Amazon SQS does not support FIFO (First In First Out); therefore, even if we remember the offset in the reduce queue, it may point to a different key-value pair when we read it again next time. We are in the process of porting CMR to run on top of Appistry's cloud platform so that it can be deployed in an internal cloud environment. Appistry's queue supports FIFO; hence, we will be able to apply the same techniques we used for the map tasks to the reduce tasks.

4 Experimental results

We evaluate Spot CMR's performance in Amazon EC2. A comprehensive evaluation is beyond the scope of the paper. Instead, we show that Spot CMR can work well in the spot market environment and that it can significantly reduce cost by leveraging spot pricing. Since the current Spot CMR cannot tolerate many terminations in the reduce phase, we only use the spot instances in the map phase, and we use the regular instances in the reduce phase for our evaluations.

We first run the Word Count application on a 50GB of crawled web pages data using 20 *m1.small* instances. We break the input data into 100 splits of 500MB each. Spot CMR parses the input files, and passes the line number as the key and the line content as the value to the user-defined map function. We simulate a spot market scenario where the price increases every 20 minutes, then drops right away. As a result, all 20 instances are killed

every 20 minutes, and they are then restarted right away. Spot CMR takes 93.4 minutes to complete the job. In contrast, if we assume the instances are never turned off (e.g., when the bid price is much higher), CMR takes 84.2 minutes to complete. Spot CMR introduces a slight overhead when nodes are constantly terminated, which includes the time taken: 1) for a Linux OS to boot, 2) for Spot CMR to start and figure out the current progress, and 3) for Spot CMR to save the progress when terminated.

Spot MapReduce stores more data (e.g., more commits) in SimpleDB, which could cause a higher query overhead. With no terminations, CMR consumes 0.012 hours of SimpleDB CPU time as reported by our Amazon billing statement. In comparison, Spot CMR with a failure every 20 minutes consumes 0.013 hours of SimpleDB CPU. Most of the time is consumed in the reduce phase when the reduce workers are querying SimpleDB and are trying to determine if there are enough commits for all map tasks.

We also compared with Hadoop 0.21.0 on the same input data. Assuming no failure, Hadoop took about 119.6 minutes to finish. We also compare with the failure cases. Unfortunately, we cannot terminate a large number of nodes in Hadoop, not only because a master node failure is catastrophic, but also because we cannot afford losing all replicas of a data item. Instead, we randomly terminate 3 instances and restart them every 10 minutes. In this case, Hadoop took 187.5 minutes to finish. As observed in [1], Hadoop's performance is significantly impacted in such a dynamic environment.

We then consider the potential cost savings by using the spot market. We choose a real spot price history in a period of time between Sep. 9th 1pm and Sep. 10th 2010 in Amazon's us-east data center, then we simulate and replay the price changes by manually turn off our nodes when necessary. We choose this period for evaluation because the price fluctuates widely going from the lowest $0.029 to the highest of $0.085. It is shown in Fig. 4.

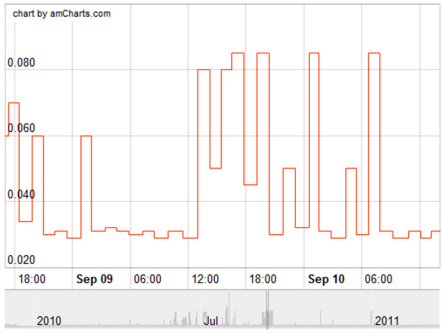

Figure 4: Spot price history in Amazon EC2 us-east data center in Sep. 2010.

For Spot CMR, we bid at the lowest price of $0.029. The total time, including the time waiting for the price to drop, is 22.4 hours because most of the time is spent waiting. However, it only costs $1.16. We can also use Hadoop in the spot market, but bid at a much higher price to avoid termination in that period. We find that this approach would cost $2.60, but it comes with a much quicker turn around time of 119.6 minutes. If we use regular instances for Hadoop, the cost would be $3.4. Clearly, Spot CMR costs much less to process a MapReduce job than Hadoop, although the user must be willing to shift the demands by possibly waiting for a long time.

5 Conclusion

We described the first MapReduce implementation that could continue making progress even when many nodes terminate at the same time in a spot market environment. By adopting a spot market and Spot Cloud MapReduce, an infrastructure cloud provider could shift computation demands to increase the overall utilization of its infrastructure. Since MapReduce jobs represent a large proportion of batch jobs, the amount of demands that we can shift, hence the degree of increase in utilization, could be significant.

References

[1] CHOHAN, N., CASTILLO, C., SPREITZER, M., STEINDER, M., TANTAWI, A., AND KRINTZ, C. See spot run: using spot instances for mapreduce workflows. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 7–7.

[2] CONDIE, T., CONWAY, N., ALVARO, P., AND HELLERSTEIN, J. M. Mapreduce online. In *Proc. NSDI* (2010).

[3] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation* (December 2004).

[4] GRIFFITH, R., CHEN, Y., LIU, J., JOSEPH, A., AND KATZ, R. Understanding tcp incast throughput collapse in datacenter networks. In *Proc. SIGCOMM WREN Workshop* (2009).

[5] GUY ROSEN. Estimated EC2 daily usage. http://www.jackofallclouds.com/2010/12/recounting-ec2/.

[6] Hadoop. http://lucene.apache.org/hadoop.

[7] LIU, H., AND ORBAN, D. Cloud mapreduce: a mapreduce implementation on top of a cloud operating system. In *Proc. IEEE/ACM International Symposium on Cluster Computing and the Grid* (Newport Beach, USA, 2011), CCGRID '11, IEEE Computer Society.

[8] VASUDEVAN, V., PHANISHAYEE, A., SHAH, H., KREVAT, E., , ANDERSEN, D., GANGER, G., GIBSON, G., AND MUELLER, B. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *Proc. SIGCOMM* (2009).

[9] YI, S., KONDO, D., AND ANDRZEJAK, A. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. In *Proc. 3rd IEEE Intl. Conf. on Cloud Computing* (2010), pp. 236–243.