NEWT - A fault tolerant BSP framework on Hadoop YARN

Ilja Kromonov, Pelle Jakovits, Satish Narayana Srirama
Institute of Computer Science, University of Tartu, J. Liivi 2, Tartu, Estonia
{kromon, jakovits, srirama}@ut.ee

***Abstract*—The importance of fault tolerance for the parallel computing field is ever increasing, as the mean time between failures is predicted to decrease significantly for future highly parallel systems. The current trend of using commodity hardware to reduce the cost of clusters forces users to ensure that their applications are fault tolerant. When it comes to embarrassingly parallel data-intensive algorithms, MapReduce has gone a long way in simplifying the creation of such applications. However, this does not apply to iterative communication-intensive algorithms common in the scientific computing domain. In this work we propose a new programming model inspired by Bulk Synchronous Parallel (BSP) for creating new a fault tolerant distributed computing framework. We strive to retain the advantages that MapReduce provides, yet efficiently support a larger assortment of algorithms, such as the aforementioned iterative ones.**

I. Introduction

In recent years cloud-based platforms have emerged as alternatives to supercomputers and grids for high performance computing needs. With the illusion of infinite resources, cloud computing allows one to loan computation time on demand with a flexible pay-as-you-use billing model. However, applications are placed in an environment associated with a high risk of hardware failure. This is amplified by the use of commodity equipment by many cloud service providers to lessen the cost of data center components, meaning that fault tolerance is of utmost importance for any long-running applications in this environment.

For these reasons, Hadoop MapReduce framework has found widespread use in the cloud-based distributed computing field. It provides fault tolerance by replicating both data and computation. Originally introduced by Google in 2004 [1], it excels at solving data-heavy embarrassingly parallel problems, however, has trouble with more sophisticated algorithms [2] as MapReduce was simply not designed to support them. Its processes were designed to be stateless, ensuring that all input blocks are eligible for each of the available processes without affecting the outcome. This concept ensures that failure of one of the nodes does not affect the sequential consistency of the program and is at the core of the MapReduce fault tolerance mechanism.

For more sophisticated algorithms one can use Message Passing Interface (MPI) - an established standard, which throughout the years has become the de facto way of writing parallel programs. While allowing for a large degree of flexibility, MPI code tends to be error prone and difficult to debug and maintain, especially with the possibility of deadlocks and race conditions. Unfortunately, as of MPI version 3, fault tolerance is still not part of the MPI standard. Managing faults is left to specific implementations, typically requiring extra work to make applications fault tolerant.

Another alternative is using frameworks based on the Bulk Synchronous Parallel [3] (BSP) computing model, designed for parallel iterative algorithms. BSP computations consist of a series of supersteps, each divided into three stages:

- Concurrent computation (using only local data)
- Communication
- Barrier synchronization

One of the main advantages of this scheme is elimination of race conditions and deadlocks, by avoiding circular data dependencies. The resulting structure of programs presents an easily obtainable overview of the implemented algorithm's granularity, giving a estimate of the expected parallel runtime and performance.

For these reasons we consider the Bulk Synchronous Parallel model to be an ideal basis for a parallel computing framework, which can have all the properties that make it as viable for integration with the cloud environment as one based on MapReduce, while providing a message passing paradigm familiar to MPI programmers, without many of the issues involved. Unfortunately, the existing BSP solutions either do not provide the fault tolerance required by long running applications or are designed for specific type of applications, such as graph computations and thus require extra effort for adapting other kind of applications to them.

Thus, we propose a BSP-inspired programming model which enables transparent stateful fault-tolerance for programs that follow this model and provides better support for a wider range of algorithms than the current solutions To validate the approach, we created a framework following this model and implemented a number of typical iterative scientific computing algorithms on it.

II. Proposed Solution

The goals of the proposed solution are to provide automatic fault recovery, retain the program state after fault recovery, provide a convenient programming interface and support (iterative) scientific computing applications.

Before defining the model, the first thing to note is that in more complex iterative programs, each iteration may consist of more than one distinct BSP superstep. To accommodate the continuation of the recovered program at the correct stage of the iteration, without storing the entire address space, it

makes sense to write it as a finite state machine (FSM). In the resulting FSM each such stage is equivalent to one of the states. This leads us to view programs under the BSP model as suitable for an abstract computer, which consists of a mutable state, message queues, label to function map, label of the next function and a communicator for sending messages. Following pseudocode describes the inner workings of such a machine:

$state \leftarrow initialState$
$next \leftarrow initialLabel$
while $true$ **do**
  $next \leftarrow execute(next, state, comm)$
  $barrier(comm)$
  if $next == none$ **then**
    $break$
  end if
end while

The $execute$ call runs the function defined by label $next$ and returns the label of the next function in the sequence. The mutation of state and sending/receiving of messages (through communicator $comm$) is achieved as a side-effect of these functions. There is a need for communication primitives that cover the semantics of sending and receiving messages. These primitives are made accessible through the communicator. The $barrier$ initiates communication and synchronizes all machines as per the BSP model.

The given generic program structure allows for the state, label of the next stage and incoming message queue to be stored into a persistent storage between calls of $execute$ for later recovery. This initialization of the recovered process can be achieved seamlessly by replacing the initial state with the state from the latest checkpoint and the processes that did not fail can complete the recovery by simply replacing their current state. A program has to define the state and a mapping of labels to functions, which describe the program flow. The return value of each of these functions is the label of the next function.

To validate this approach, we created a proof-of-concept prototype implementing the given model, which is built on top of Hadoop YARN [4] for resource management, scheduling and Hadoop Distributed File System (HDFS), and Apache MINA [5] for interprocess communication.

III. Experiments and Conclusions

We implemented Conjugate Gradient (CG) and Partitioning Around Medoids (PAM) algorithms on the prototype. We compared the prototype to BSPonMPI - a BSPlib implementation, which we determined in previous work [6] to perform as good as MPI for the given algorithms.

	k-medoids clustering			conjugate gradient	
p	BSPonMPI	NEWT	p	BSPonMPI	NEWT
1	597.05	578.05	1	136.11	123.69
2	326.42	344.17	2	76.52	81.24
4	174.97	196.84	4	41.98	64.32
8	91.70	108.04	8	24.91	53.18
16	110.67	114.20	16	27.54	75.19

TABLE I. Running time comparison between NEWT and BSPonMPI (in seconds).

In the scalability trials each of the algorithms was given input of size that was kept constant (a sparse system of 8000000 linear equations for CG and 80000 points across 32 clusters for PAM) and only varied the number of processes p. The results in table I for NEWT include a 10-15 second overhead that is induced by YARN for initialization and allocation of process containers. Despite this, the scaling on a coarse-grained parallel algorithm, such as PAM, is nearly identical to the BSPonMPI implementation, suggesting that structuring the algorithm according to the model does not impose a significant overhead.

We also measured how long it takes to write and restore checkpoints to HDFS. For PAM it took between 5 to 30ms to write checkpoints consisting of 300000 2D points. In case of CG, the average was 30s for a 400MB checkpoint. Reading PAM checkpoints took approximately 300ms and 400 megabyte CG checkpoint from HDFS took 13 seconds on average. Apart from the time it takes to read the checkpoints from HDFS, the recovery overhead includes the requesting of new containers from the YARN resource manager and the restarting of socket connections between processes.

When it comes to fine-grained parallel algorithms, such as CG, where the computation of one superstep on average took under 100 milliseconds, the communication part of the runtime significantly outweighs the computation part as the number of parallel processes grows, resulting in subpar scaling on the current version of the prototype. Since the BSPonMPI implementation performs much better, the likely culprit is the latency from the current prototype's implementation of message passing and barrier synchronization, where one possible solution is building the framework top of an MPI implementation that is compatible with the YARN environment.

It indicates that the currently chosen communication library Apache MINA may not be the best candidate for such applications. The currently ongoing work of supporting MPI on a YARN cluster may address this issue, such as Hamster [7], but this requires additional investigation.

Acknowledgment

This work is supported by European Regional Development Fund through EXCS, IT Academy, Estonian Science Foundation grant ETF9287 and Target Funding SF0180008s12.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] S. N. Srirama, P. Jakovits, and E. Vainikko, "Adapting scientific computing problems to clouds using mapreduce," *Future Gener. Comput. Syst.*, vol. 28, no. 1, pp. 184–192, Jan. 2012.

[3] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[4] Apache Software Foundation. (2013, Jun.) Hadoop YARN. [Online]. Available: http://hadoop.apache.org/

[5] ——. (2013, Jun.) MINA. [Online]. Available: http://mina.apache.org/

[6] P. Jakovits, S. Srirama, and I. Kromonov, "Viability of the bulk synchronous parallel model for science on cloud," in *High Performance Computing & Simulation. International Conference on*, 2013, (In print).

[7] (2013, Jun.) Hamster: Hadoop and mpi on the same cluster. [Online]. Available: https://issues.apache.org/jira/browse/MAPREDUCE-2911