International Journal on Recent Researches In Science, Engineering & Technology
(Division of Computer Science and Engineering)
A Journal Established in early 2000 as National journal and upgraded to International journal in 2013 and is in existence for the last 10 years. It is run by Retired Professors from NIT, Trichy. It is an absolutely free (No processing charges, No publishing charges etc) Journal Indexed in JIR, DIIF and SJIF.
Research Paper
Available online at: www.jrrset.com

ISSN (Print) : 2347-6729
ISSN (Online) : 2348-3105

Volume 3, Issue 1, January 2015.

JIR IF : 2.54
DIIF IF : 1.46
SJIF IF : 1.329

CATEGORIZATION OF HADOOP COMMUNITY MODEL FOR MULTI-TIER SUPPORT SCHEME

C.Sunil Kumar[1], N.Soundararajan[2], B.Sakthishree[3]

[1]Professor, Department of Computer Science and Engineering, Mahendra Engineering College, Mahendhirapuri, Namakkal District, Mallasamudram, Tamilnadu, India

[2,3]Assistant Professor Department of Computer Science and Engineering, Mahendra Engineering College, Mahendhirapuri, Namakkal District, Mallasamudram, Tamilnadu, India

Abstract: Current distributed processing systems, for example, MapReduce and Spark, enable software engineers to utilize just restricted operations characterized by the structure. Based on the limitation, algorithms that don't fit with the system can't be proficiently communicated. The confinement emerged from the need of adaptation to fault-tolerance. That is, these structures recoup lost information by re-computing them from accessible information when fault happens. To guarantee the mechanism works effectively, just operations gave by the framework can be utilized. Then again, there is another fault-tolerance strategy called checkpointing. Since, it accomplishes fault-tolerance by saving memory substance, there is no such limitation to operations. In any case, the cost of saving a memory picture is high. To overcome this exchange, propose a light-weight checkpointing technique called continuation-based check pointing, which empowers low fault-tolerance with no limitation. It saves the data that is important for restarting, which essentially diminishes the cost of check pointing. A distributed computing structure called Feliss by utilizing our continuation-based checkpointing strategy, which incorporates an enhanced MapReduce without the above restriction and a message passing interface (MPI) subset. Assessed Feliss with different applications and demonstrated that order-of-magnitude speedup can be accomplished with applications that can't be communicated productively with current structures.

Key Words: distributed computing system, MapReduce, Spark, fault-tolerance, message passing interface (MPI), continuation-based check-pointing, Feliss, light-weight checkpointing technique.

Introduction

It is getting to be plainly normal to gather information from sensors, the Web, business exchanges, etc., and examine these "Big Data" to deliver helpful data. To examine the information, Hadoop/MapReduce is ordinarily utilized. The works proficiently to the program fits well with MapReduce. Something else, the performance is extremely poor. It is because of the restriction of the structure. Since, MapReduce gives restricted operations (e.g. map and reduce), different MapReduce capacities must be joined to execute practical algorithms.

MapReduce essentially requires its information and output to be on HDFS, which is an appropriated document framework, gave by Hadoop. Thusly, these joined MapReduce capacities must convey through HDFS document I/O instance. It is hard to bypass the record I/O on because the dataflow is predefined and access to other distributed information from map or decrease is not permitted. Due to this restriction, machine learning and data mining algorithms require iterative calculation, are known to run slowly with MapReduce [1].

To tackle this issue, different structures have been proposed. Spark is viewed as one of the most encouraging structures. Spark can execute machine learning algorithms significantly speedier than MapReduce by maintaining a strategic distance from such record I/O. Spark works on a distributed information structure called resilient distributed dataset (RDD), operations, for example, map, join, and filter are given.

The fact of Spark is greatly enhanced from MapReduce, operations that can be utilized are as yet restricted, which makes it hard to proficiently execute certain types of algorithms. A case of such an algorithm is "Gaussian elimination", which iteratively refreshes just a piece of a framework. Spark does not give an operation that modifies a RDD, because RDDs are changeless; another RDD should be made to update the past one. In the way, updating a piece of a matrix requires duplicating the whole lattice including the non-updated part, which prevents effective usage. Such limitation of MapReduce and Spark comes from their re-computing based execution of fault-tolerance.

On account of MapReduce, the framework re-executes map or reduce works on part of the input document to recuperate lost information if a fault happens. On account of Spark, the framework recollects the list of operations connected to RDDs and re-computes the lost part from where the information remains. RDDs should be changeless to influence this component to work accurately. In the two cases, software engineers should not utilize distributed information other than what is characterized by the framework (i.e. RDD and HDFS). That is, if such client characterized information is not fault-tolerant, recomputation can't recover them. In the event that the information is altered during calculation, re-calculation modifies them once more, which outcomes in a conflicting execution state.

Then again, there is another well known system to accomplish fault-tolerance: checkpointing. With checkpointing, the execution state is saved occasionally as a "checkpoint". If a fault happens, the saved checkpoint is reestablished and execution is continued. Hence, there is no limitation of operations when utilizing checkpointing. In any case, checkpointing has the accompanying issues. On account of framework level checkpointing, the whole memory of a procedure is saved as an execution state. It requires a lot of time to save memory substance. This high overhead is the reason MapReduce and Spark don't utilize checkpointing. On account of utilization level checkpointing, a software engineer expressly determines which factors to save as an execution state. This takes essentially less time than framework level checkpointing in light of the fact that exclusive the data that is important to restart is saved. However, indicating each factor to save is a difficult for developers and inclined to error; checkpointing does not work effectively if a software engineer neglects to save factors that are expected to restart.

To overcome this exchange off between the existing fault tolerance strategies, propose a light-weight checkpointing technique called continuation-based checkpointing, which empowers low overhead fault-tolerance with no restriction. It is a sort of application level checkpointing technique; however software engineers don't need to explicitly determine which factors to save. Rather, software engineers compose a program with the goal that whatever is left of the

calculation is every so often changed over to an information structure, called a continuation, and stored into a pool.

Related Works

Research on checkpointing has a long history. For framework level checkpointing, BLCR has been broadly utilized as of late, which acts as a kernel module of Linux. Libckpt [2] fills in as a client level library, and backings incremental checkpointing, which saves just the changed part after the past checkpoint. Our present structure does not support incremental checkpointing and intend to execute for future work. In Condor [3] checkpointing is executed as a client level library. It utilizes checkpointing to relocate a procedure for asset management in a distributed computing condition. Applying checkpointing for such an object is additionally for future work.

Transparent application level checkpointing with the help of a compiler was proposed in [4]. It analyzes live factors that should be checkpointed in the interest of a software engineer. Its inspiration was like our own, however they couldn't decrease the checkpoint size of complex information structures, for example, std::map, on the grounds that data of the information structure is required (e.g. to recover std::map, pointer data is not required). On account of serialization, a serialization library (or a serialization code gave by a software engineer) productively serializes such information structures utilizing the data. Moreover, the execution cost of making another compiler is substantially bigger than that of library layer usage, for example, our own.

Charm++ [5] accomplishes checkpointing on the programming language layer. Objects of Charm++ are area autonomous and can move to different processors. This characteristic is utilized for checkpointing. These articles are like our continuations. Feliss is not the same as Charm++ in that Feliss does not require utilizing a unique language.

These checkpointing strategies are utilized together with MPI. For instance, Open MPI, which is a generation level MPI usage, supports checkpointing utilizing BLCR [6]. Concerning information investigation systems, there has been many attempts `to expand the speed of substantial scale machine learning. Be that as it may, current frameworks have restrictions. Haloop [7] and Twister [8] can productively execute iterative MapReduce. However, they don't support other execution designs. Piccolo [9] and Distributed GraphLab [10] are distributed computing systems that are particular to particular information structures, for example, a table or a graph.

They accomplish fault-tolerance by checkpointing these information structures. These systems have limitation in which just limited distributed information structures are supported. Sparkler [11] stretches out Spark to enhance the performance of expansive scale lattice factorization issues. Be that as it may, it just enhances the execution of such sort of issues; the limitation still remains. SystemML [12] incorporates a machine learning algorithm written in an area particular language into a Hadoop/MapReduce program. This gives adaptability to the client, however it displays poor performance due to the restriction of Hadoop; if the information measure is little, its performance is much more awful than R, however it has advantage that it can deal with bigger information than R. M3R [13] enhances its performance by making a Hadoop/MapReduce-perfect layer that surrenders fault-tolerance and out-of-core calculation.

Proposed System

Remote procedure call (RPC) as the primitive of dispersed execution, which is a component to call a capacity on a remote server. Sending and accepting information between servers should likewise be possible with RPC. Remote procedure call is a conventional system, however existing RPC requires a developer to compose an interface explanation in IDL. It is gathered

by an IDL compiler to deliver stub codes. A customer and a server program are composed utilizing the stub. Since such a method is cumbersome, designed and executed another RPC framework that does not require IDL.

To influence such RPC conceivable, the customer needs to advise the server which capacity to call. For this reason, it utilized the symbol name of the capacity simply like the serialization of a capacity pointer in our continuation-based checkpointing technique. Arguments of the capacity are serialized by Boost and sent to the server together with the symbol name of the capacity.

Fault-Tolerance with Continuation-based Checkpointing

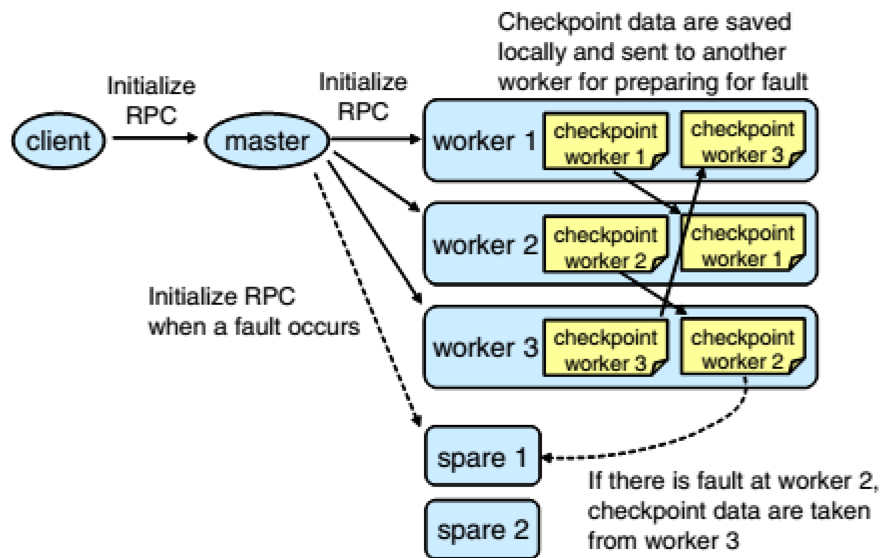

Figure.1 Overview of distributed checkpointing

For distributed checkpointing, the accompanying servers are determined at instatement: master, workers that really do the calculation, and spare servers that are utilized when a fault happens. Figure 1 gives the review of the distributed checkpointing framework. A customer program initially associates with the master and introduces RPC. At that point the master interfaces with the laborers and introduces RPC.

A program makes a continuation and determines it as the primary continuation. The master places it into the continuation pool of one of the workers, which begins the program execution. During the execution of the continuations, a few continuations are put into other workers' continuation pools with RPC, which brings about circulated execution.

The master occasionally takes checkpoints at determined interims. At the point when a checkpoint is taken, the master stops execution of the considerable number of workers. That is, in the wake of completing right now executing continuation, every one of the workers prevent getting continuations from the continuation pool. In the wake of affirming that every one of the workers has stopped, the master requests the workers to take a checkpoint. Every worker serializes the continuation pool and saves it onto the local disk. The serialized information is also sent to another worker to plan for server fault. In the wake of affirming that every one of the workers wrapped up a checkpoint, the master restarts the workers.

In the event that there is a long running continuation on a worker, different workers must wait for that worker to stop before beginning checkpointing. Asynchronous checkpointing, for example, the Chandy-Lamport algorithm can take care of this issue. Actualizing such an algorithm is for future work.

The master occasionally checks if the specialists are running effectively. In the event that a worker is not working, the master restarts every one of the master from the checkpoint. More

particularly, the master stops the entire worker forms and restarts them; they read previously taken checkpoint information and proceed with execution from that state. The checkpoint information is read from the local disk if the worker server is as yet working. Something else, a spare server is utilized rather; it reads the checkpoint information from the worker that has a duplicate of it.

Accept a fault does not happen at the master. This assumption is sensible because the quantity of workers is significantly bigger than that of the master, which is one, and the fault probability is corresponding to the quantity of servers. This assumption is embraced by most distributed systems, for example, Hadoop and Spark.

Variable Wrapper and Utility Functions

In the event that a server restarts from a checkpoint, addresses of the factors may change when they are recovered from checkpoint information. This is not an issue inside a server because the pointers that point to the factors are also changed by their addresses during the deserialization procedure. Then again, it becomes problematic when a developer needs a pointer that focuses to another server's variable.

To resolve of this issue, present "variable wrapper", this wraps factors. A variable wrapper fills in as an ID of a pointer; there is a mapping table between the ID and the pointer in every server. Since the table exists inside a server, the mapping can be kept up effectively after restart. Software engineers can utilize the ID as the pointer to another server's variable. Additionally, it utilized the variable wrapper for different purposes. The first is synchronization. The execution time of a continuation must be sufficiently short. Be that as it may, if the execution is hindered by synchronization, it may turn out to be too long. To take care of this issue, the framework gives a synchronization method that does not piece execution.

With this method, the variable wrapper works like a "future". That is, the variable wrapper has two expresses: a pointer is set or not set to it. To read the pointer of the variable wrapper, a continuation is determined; it is executed after the pointer is set to the variable wrapper. This empowers synchronization without blocking. Another usefulness added to the variable wrapper is saving the substance of the pointer onto the local disk after serialization, which is utilized to help out-of-core calculation. The information saved on the disk are duplicated to another worker together with checkpoint information when a checkpoint is taken. The framework gives communicate and decrease capacities with a tree-based correspondence algorithm, which utilizes variable wrappers as remote pointers.

Improved MapReduce and MPI

Software engineers can compose distributed fault-tolerant projects with the functionalities clarified above: RPC, continuations, and utility capacities together with factor wrappers. Nonetheless, these functionalities are somewhat primitive. Along these lines to gave MapReduce over the checkpointing layer. The MapReduce executed was enhanced so the restriction of Hadoop/MapReduce is expelled; it doesn't require the info and yield to be on HDFS, and access to discretionary disseminated information is permitted.

Other than that, the MapReduce API is like that of Hadoop/MapReduce and Google's unique MapReduce; map, reduce, combine, and partition capacities are indicated as the arguments of the mapreduce work. These customer gave capacities are work objects, which are cases of classes that have operator() as a part work. Such capacity items can encapsulate states. The states can be pointers; access to other information from these capacity objects is permitted. This empowers more adaptable and speedier execution than with the conventional MapReduce usage.

Our MapReduce usage utilizes the queues clarified above for information and output of MapReduce. Various MapReduce capacities can be associated through the queues. One MapReduce can go the information through memory to the next. Furthermore, the last MapReduce can start execution before the former MapReduce totally wraps up.

Consequently, in such a case, our usage runs quicker than conventional executions. In one MapReduce, the map-processing and decrease processing parts are additionally associated through queues. Sorted map outcomes are sent to the diminish part, and the lessen part combine sorts outcomes because of numerous maps, which outcomes in so called shuffle stage. Since the lessen part can't begin before the map part totally completes, the map outcomes won't not fit in memory. Such information is spilled out to the local disk utilizing the variable wrapper portrayed previously. The spilled map outcomes are arranged with an external sort algorithm in the shuffle stage.

Hadoop/HDFS can be utilized as input and output. To utilize HDFS as input, the framework gives a capacity that reads pieces of a record from HDFS and places them into the input queues of MapReduce. Load balancing is done here like the first MapReduce. It is additionally mindful of the region of the record; it puts local chunks beyond what many would consider possible into the queues.

Additionally, it executed a MPI subset over the checkpointing layer, which empowers us to utilize existing MPI libraries and algorithms written in MPI. To utilize MPI, the continuation of the best level capacity that utilizations MPI should be sent to every one of the workers. In the capacity, a software engineer can compose a MPI program as usual. To utilize both MapReduce and MPI together, a software engineer can simply assemble the mapreduce work. They can communicate through pointers to factors, variable wrappers, queues, etc.

Result and Discussion

To affirm the effectiveness of our continuation-based checkpointing strategy, it contrasted its performance and Berkeley Lab Checkpoint/Restart (BLCR), which acknowledges checkpointing by storing the whole memory picture of a procedure onto a disk.

Figure. 2 demonstrates the assessment outcomes. The program utilized for the assessment holds std::map<string, string> as information, which stores 20 million components. It assessed their checkpointing times and checkpoint record sizes utilizing one server. Feliss could take a checkpoint 5.3 times quicker than BLCR. The checkpoint document measure was 5.6 times littler than that of BLCR, which added to the speed of the checkpoint.

One purpose behind this is std::map utilizes an information structure called a red-black tree, which utilizes numerous pointers inside. Due to this memory overhead, it requires considerably more memory than the really stored information measure. Serialization changes such information structures to contain just the really stored information, which significantly decreased the information measure. Information investigation applications as a rule require such complex information structures, which is unique in relation to conventional HPC programs that generally utilize dense information structures, for example, an exhibit.

Next, assessed the checkpointing and restarting times of utilizations on Feliss. The applications utilized for assessment were: word count (WC), logistic regression (LR), LU deterioration (LU), latent semantic analysis (LSA), and latent Dirichlet allocation (LDA). WC utilizes MapReduce. LU is actualized with Gaussian elimination utilizing MPI. LSA utilizes both MapReduce and MPI together with Parallel ARPACK to acknowledge solitary value disintegration that is required in LSA. Feliss has such favorable position that it can use existing great library written in MPI, which is troublesome with Spark or Hadoop. LDA utilizes different MapReduce capacities, which are proficiently associated.

The input information of WC, LSA, and LDA was English Wikipedia content of around 8 GB. The input information of LR is 20 GB of vectors. The input information of LU is a lattice of size 4320 x 4320.

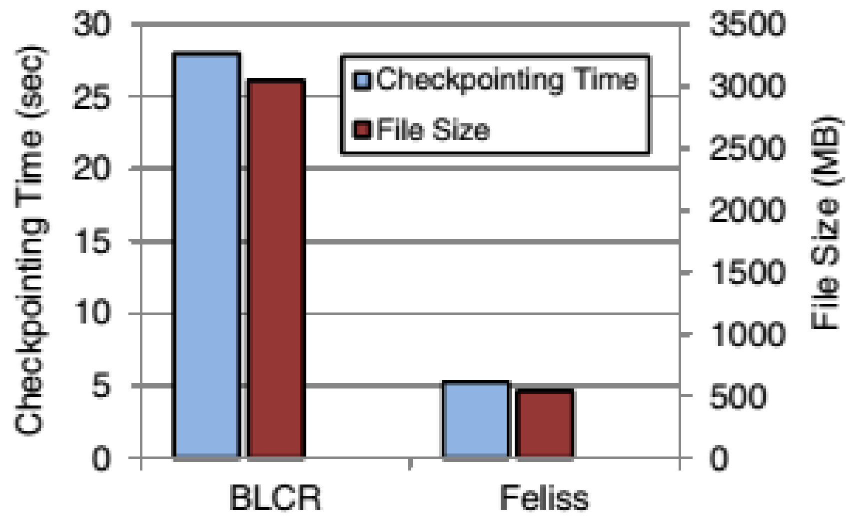

Figure 2. Comparison with BLCR

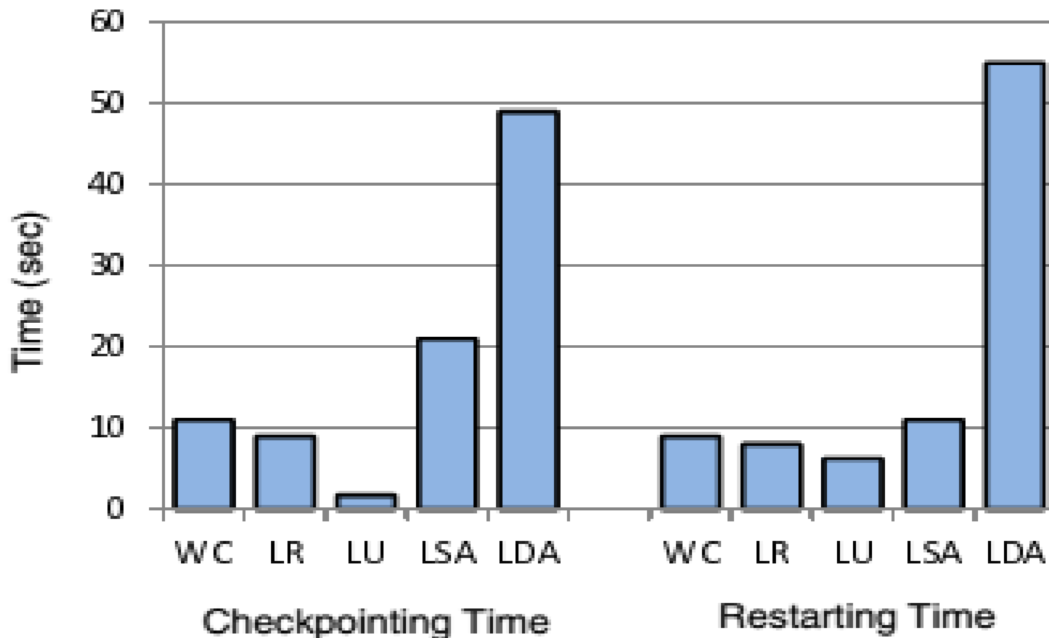

Figure. 3 Checkpointing and Restarting Times of Applications

Quantified (1) the execution time without checkpointing, (2) execution time with one checkpointing, and (3) execution time with one checkpointing and a fault; one of the procedures was killed soon after taking the checkpoint, and the program restarted from the checkpoint after that. It utilized (2) - (1) as checkpointing time, and (3) - (2) as restarting time. It assessed them on 18 servers (72 CPUs). Figure 3 demonstrates the assessment outcomes. Both checkpointing and restarting times were sufficiently enough on genuine applications. The checkpointing and restarting times of LDA were bigger than those of alternate applications because LDA utilizes more memory than the others.

To decide the performance of Feliss, initially assessed its scalability. Figure 4 demonstrates the execution speed of the above applications with different quantities of CPUs. It demonstrates relative speed standardized by the speed with 1 CPU. The checkpoint was taken once during the execution. The input information was the same as in the past assessment.

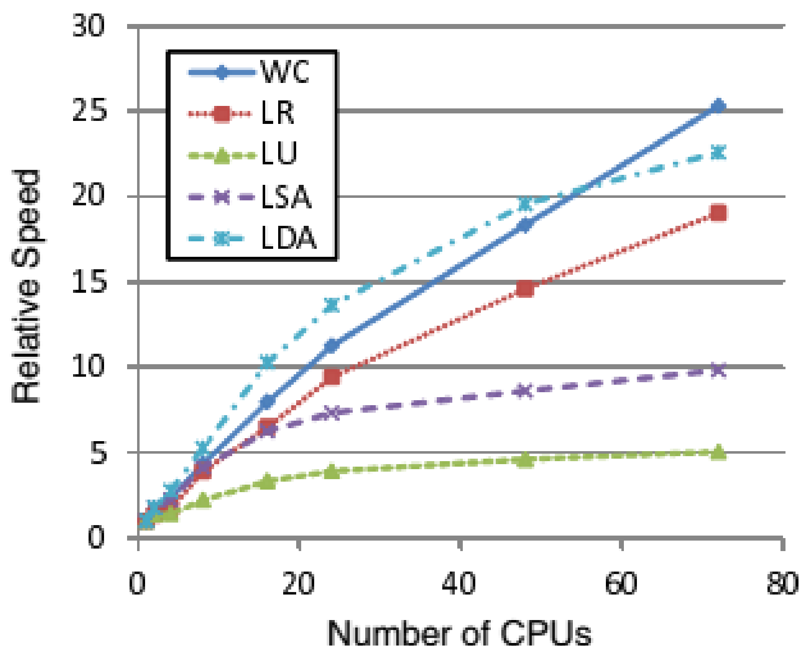

Figure. 4 Scalability of Feliss

Conclusion

A light-weight checkpointing method called continuation-based checkpointing, and utilized it to implement a flexible distributed computing framework called Feliss. It implemented and evaluated various applications on Feliss and confirmed that it can execute them more efficiently than current frameworks. Future work includes extension of checkpointing such as asynchronous checkpointing, incremental checkpointing, and using checkpointing for resource management. Completion of MPI implementation is also for future work. To utilized Feliss as a platform of data mining research. It will add other abstractions to Feliss to make it easy to express various data mining algorithms.

References

[1] M. McCauley, J. Ma, M. Chowdhury, M. Zaharia, T. Das, A. Dave, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for inmemory cluster computing," in NSDI, 2012.

[2] M. Beck, J. S. Plank, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in Usenix Winter Technical Conference, 1995, pp. 213–223.

[3] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of UNIX processes in the Condor distributed processing system," University of Wisconsin - Madison Computer Sciences Department, Tech. Rep. UWCS-TR-1346, 1997.

[4] G. Rodriguez, I. Cores, M. Martin, and P. Gonz'lez, "Reducing application-level checkpoint file sizes: Towards scalable fault tolerance solutions," in 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2012, pp. 371 –378.

[5] L. Shi, G. Zheng, and L. V. Kale, "FTC-Charm++: an inmemory checkpoint-based fault tolerant runtime for Charm++ and MPI," in CLUSTER, 2004, pp. 93–103.

[6] J. Squyres, J. Hursey, T. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for Open MPI," in IPDPS, 2007, pp. 1 –8.

[7] B. Howe, Y. Bu, M. Balazinska, and M. D. Ernst, "HaLoop: efficient iterative data processing on large clusters," Proc. VLDB Endow., vol. 3, no. 1-2, pp. 285–296, 2010.

[8] S.-H. Bae, H. Li, J. Ekanayake, B. Zhang, T. Gunarathne, J. Qiu, and G. Fox, "Twister: A runtime for iterative MapReduce," in HPDC'10, 2010, pp. 810–818.

[9] J. Li, and R. Power "Piccolo: Building fast, distributed programs with partitioned tables," in OSDI, 2010, pp. 293–306.
[10] C. Guestrin, D. Bickson, Y. Low, J. Gonzalez, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," Proc. VLDB Endow., vol. 5, no. 8, pp. 716–727, 2012.
[11] S. Tata, B. Li, and Y. Sismanis, "Sparkler: supporting largescale matrix factorization," in EDBT, 2013, pp. 625–636.
[12] Y. Tian, R V. Sindhwani,. Krishnamurthy, A. Ghoting, E. Pednault, B. Reinwald, S. Tatikonda, and S. Vaithyanathan, "SystemML: Declarative machine learning on MapReduce," in ICDE, 2011, pp. 231–242.
[13] D. Cunningham, A. Shinnar, V. Saraswat, and B. Herta, "M3R: increased performance for in-memory Hadoop jobs," Proc. VLDB Endow., vol. 5, no. 12, pp. 1736–1747, 2012.