Evaluating MapReduce Frameworks for Iterative Scientific Computing Applications

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

***Abstract*—Scientific Computing deals with solving complex scientific problems by applying resource-hungry computer simulation and modeling tasks on-top of supercomputers, grids and clusters. Typical scientific computing applications can take months to create and debug when applying de facto parallelization solutions like Message Passing Interface (MPI), in which the bulk of the parallelization details have to be handled by the users. Frameworks based on the MapReduce model, like Hadoop, can greatly simplify creating distributed applications by handling most of the parallelization and fault recovery details automatically for the user. However, Hadoop is strictly designed for simple, embarrassingly parallel algorithms and is not suitable for complex and especially iterative algorithms often used in scientific computing. The goal of this work is to analyze alternative MapReduce frameworks to evaluate how well they suit for solving resource hungry scientific computing problems in comparison to the assumed worst (Hadoop MapReduce) and best case (MPI) implementations for iterative algorithms.**

***Keywords*—MapReduce; Distributed computing; Scientific computing; Hadoop; Spark; HaLoop; MPI; Twister;**

I. Introduction

Performing precise scientific experiments or simulations typically requires the utilization of supercomputers, grids or computer clusters and is closely dependent on the advances of High Performance Computing (HPC) and parallel programming fields. The involved large-scale scientific applications must be able to utilize multi-core and multi-node systems and also to survive machine or network failures when they are expected to run for extended time periods. The de facto solution for creating distributed solutions is Message Passing Interface (MPI). However, it typically requires very low level programming in C or Fortran languages and debugging MPI applications and assuring their fault tolerance is a very complex task, which can increase exponentially with the complexity of the application.

Frameworks like MapReduce can simplify this work by providing near automatic parallelization and fault recovery for algorithms adapted to the MapReduce model. It would be very beneficial to exploit such frameworks for solving scientific computing problems but Hadoop is designed for simple, embarrassingly parallel data processing tasks and not for more complex algorithms. Among others [1]–[3], we have also studied [4], [5] using Hadoop MapReduce for scientific computing algorithms and our results confirm that it has great difficulties handling iterative applications. To solve this issue we have investigated a number of alternatives approaches.

First approach is to use alternative non-iterative algorithms instead of the most efficient iterative algorithms and adapt those to MapReduce. For example, instead of using iterative Partitioning Around Medoids (PAM) kMedoid clustering method, we can use the non-iterative Clustering Large Applications (CLARA) method [6] which is more suitable for the MapReduce model. Another example is solving systems of linear equations. Instead of using the Conjugate Gradient (CG) [7] linear system solver, we can use the Monte Carlo [8] method for finding the matrix inverse of linear system's matrix form. However, the results of our experiments [9] have shown that while the alternative Monte Carlo linear system solver is more suitable for MapReduce, it can significantly reduce the parallel efficiency and accuracy of the solver. Moreover, finding suitable alternative algorithms or restructuring the original algorithm requires expert knowledge of the involved algorithms and thus would not simplify the work.

Another approach is to use alternative distributed computing models, which are more suitable for scaling up iterative scientific computing applications and which are able to provide similar advantages as MapReduce. Frameworks based on these models should be able to take care of most of the parallelization tasks such as data partitioning and distribution, task scheduling and synchronization, and fault tolerance. They should also be able to greatly simplify the creation of such applications for users who do not want to spend excessive time on their design and debugging.

Bulk Synchronous Parallel (BSP) [10] caught our attention when Google published that they use this model in their large scale graph computation framework Pregel instead of MapReduce. We have investigated utilizing BSP based frameworks for scaling up scientific computing applications [11] and while the results show that BSP implementations can be as efficient as MPI implementations for iterative applications and show great potential for simplifying the scaling of scientific computing applications, the current BSP implementations do not provide the same advantages as MapReduce.

We are considering a third approach in this work: using alternative MapReduce frameworks that are designed to provide better support for iterative applications, like Spark [12], Twister [13] or HaLoop [14]. Each of these frameworks have

978-1-4799-5313-4/14/$31.00

their own way of extending the MapReduce model to support more complex algorithms. While they may give up some advantages of Hadoop MapReduce to achieve this, they show great performance improvements from Hadoop for these kinds of algorithms. Our goal is to accurately evaluate how well they perform in comparison to MPI implementations and Hadoop.

Rest of the paper is outlined as follows. Section II describes each of the Mapreduce implementations. Section III outlines our evaluation methodology and experiment setup. Section IV analyses the experiment results Section V describes related work and section VI concludes the paper with a summary.

II. MapReduce

MapReduce [15] was developed by Google as a distributed computing model and a framework for performing reliable computations on a large number of commodity computers. An application following the MapReduce model consists of two methods: Map and Reduce. Its input is a list of key and value pairs and each of the pairs are processed separately by Map tasks which output the result as one or more key-value pairs. The Map output is partitioned by keys into groups which are in turn divided between different reduce tasks. Input to a Reduce method is a key and a list of all values assigned to this key. Reduce then performs user defined aggregation function on the value list and outputs the result as key and value pairs.

MapReduce framework takes care of everything else from data partitioning, distribution and communication to task synchronization and fault tolerance, greatly simplifying the writing of distributed applications as the user only has to define the content of the Map and Reduce tasks. Near automatic parallelization is achieved by simply executing a number map and reduce tasks concurrently across machines in the cluster and partitioning the input data between them.

While Google's MapReduce framework is proprietary there exists a number of freely usable MapReduce implementations such as Hadoop [16]. It has been shown [17] that Hadoop MapReduce is suitable for many data processing applications from information retrieval and indexing to solving graph problems, like finding graph components, barycentric clustering, enumerating rectangles and triangles. It has also been tested for solving embarrassingly parallel scientific computing problems [1], [3], [9] and it performed well for algorithms such as Marsaglia polar method, integer sort and Monte Carlo methods. However, it has also been shown that Hadoop has significant problems with more complex applications which utilize more complex methods.

After we investigated using MapReduce in science we noticed that most of the methods that MapReduce has difficulties with are iterative. However, there are different types of iterative algorithms. It prompted us to study [4] if MapReduce model is unsuitable for all iterative algorithms or only a certain subset. We divided iterative algorithms into separate classes based on how difficult it is to adapt them to the MapReduce model and what is their resulting structure like this: 1) Algorithms requiring a single MapReduce execution. 2) Algorithms requiring constant number of sequential MapReduce executions. 3) Algorithms that are iterative and require a single MapReduce execution at each iteration. 4) Algorithms that require multiple MapReduce executions at each iteration.

Such classification allows us to judge which algorithms are more easily adaptable to the MapReduce model and what kind of effect does belonging to a specific class have on its parallel efficiency and scalability. We adapted algorithms from each class to Hadoop and studied their efficiency and scalability. Our results [4] confirmed that Hadoop generally performs very well the embarrassingly parallel algorithms that make up the first class and for algorithms belonging to second class, when the number of MapReduce jobs is small. However, it performed much worse for the more complex algorithms belonging to the third and fourth class.

When executing multiple Hadoop jobs in a sequence, Map and Reduce tasks are stateless and the data flow is strictly one directional. There are no means to keep the state of Map and Reduce tasks in memory across multiple MapReduce executions. As a result, an iterative application consisting of 10 identical MapReduce jobs must be configured 10 times, must read input 10 times, send intermediate data to Reduce 10 times and write any changes back to the HDFS. Thus, an algorithm that requires less number of MapReduce jobs for the same task is generally more suitable for MapReduce but there are always exceptions. For example, algorithms that require sending a huge amount of intermediate data between Map and Reduce tasks might be much less efficient than iterative algorithms that require little or no intermediate data to be communicated.

Google's success with MapReduce and the popularity of Hadoop have shown that it greatly simplifies processing large datasets, and a number of alternative MapReduce frameworks have been designed for iterative applications. The following subsections describe three of such frameworks and the table I gives an overview of their differences from Hadoop.

A. Spark

Spark [12] is an open source framework for large scale data analytics. It supports both in-memory and on-disk computations in a fault tolerant manner by introducing resilient distributed datasets (RDDs) that can be kept either in memory across the machines in the cluster or on disk. In both cases the data kept in the RDD is partitioned across machines in the cluster and can be replicated to provide data recovery in case of failures. A computation is defined by applying different Spark defined operations on the RDD's such as map, group-by and reduce. Spark also supports joins, unions, co-grouping and cartesian products on RDD's and thus extends the framework capabilities beyond the simple model of MapReduce.

Fault tolerance is achieved by keeping track of operations applied to RDD's and in case of machine or network failures, lost RDD partitions are reconstructed by backtracking the applied operations and rebuilding the RDD partition from the

	Support for iterations	Task fault tolerance	Data fault tolerance	Framework issues	Advantages
Hadoop	No support for iterative tasks.	Re-executes failed Map or Reduce tasks.	All input and output data is stored in HDFS, partitioned into smaller blocks and replicated.	Long job configure time ($\geq$ 17 sec). Huge number of configuration options make optimizations difficult.	Very actively developed. Optimized for huge scale data processing.
HaLoop	Loop-aware task scheduling ensures that Map and Reduce tasks processing same partitions are scheduled on the same physical machine.	Re-executing failed Map or Reduce tasks. No real fault tolerance for the long running iterative MapReduce tasks.	Uses HDFS for input/output data replication.	No developments past prototype. Only works with Hadoop 0.20.0. Intermediate data between Map and Reduce tasks is stored to disk. Issues with recovering cached data.	Can directly reuse existing Hadoop MapReduce code. Uses caching for loop-invariant data and convergence evaluation.
Spark	Caching data between different MapReduce like task executions by introducing RDDs that can be explicitly kept in memory across the machines in the cluster.	Provides an API for checkpointing but leaves the decision of which data to checkpoint to the user.	RDD has enough information on how it was derived from other RDDs to rebuild just the missing partition. RDDs only support coarse-grained transformations, where a single operation (applied to many records) is logged.	Difficult to control the distribution of data between machines.	Also supports join, Cartesian product, cogroup and union. User chooses when and which data to keep in memory or on disk and whether to use replication.
Twister	Enables long running map and reduce tasks which do not have to be terminated between MapReduce executions.	Twister client can detect the fault, and try to restore the computation from the last iteration.	Only guarantees restoring input data that can be reloaded from file system or inherited static parameters. Any transient information stored in Map and Reduce tasks will be lost.	No distributed file system, partitioning data is a manual process. Assumes that the intermediate data produced after the map stage will fit in to the distributed memory.	Supports fully in-memory computations

TABLE I: Comparison of the chosen MapReduce frameworks

latest intermediate partitions. In contrast to fault recovery by checkpointing, there is no overhead when there are no failures.

B. Twister

Jaliya Ekanayake et al. have studied [18] using MapReduce for data intensive science and concluded that the use of iterative algorithms by scientific applications limits the applicability of the existing (at the time) MapReduce implementations like Hadoop. However, they also strongly believed that MapReduce implementations specifically designed to support such applications would be very valuable tools. Thus they proposed a new MapReduce framework for iterative scientific applications, Twister [13].

Twister distinguishes between static data that does not change and normal data that may change at each iteration. It also provides better support for iterative algorithm by enabling long running map and reduce tasks which do not have to be terminated between MapReduce executions. Twister uses no distributed file system and partitioning the input data is a manual process. It also only guarantees restoring input data that can be reloaded from the file system or static parameters inherited from the main program and any transient information stored in Map and Reduce tasks will be lost in case of failures.

C. HaLoop

HaLoop [14] is built on top of Hadoop version 0.20.0 and it directly extends the MapReduce framework by supporting iterative execution of Map and Reduce tasks, adding various data caching mechanisms and making the task scheduler loop-aware. The authors separate HaLoop from Twister [13] by claiming that it is more suited for iterative algorithms because using memory cache and long running MapReduce tasks makes Twister more prone to failures.

HaLoop can reuse existing Hadoop Map and Reduce code without modifications by specifying how they are executed in iterative manner in an encapsulating iterative job configuration. Users have to define a couple of additional classes from distance measurement for loop ending condition to caching configuration. While creating the applications themselves is not difficult as long as the user has experience with Hadoop, debugging and creating more complex job chains can be a daunting task as the documentation is sparse and the framework itself is no longer in active development.

III. Evaluation configuration

We chose three algorithms for benchmarking: Partitioning Around Medoids (PAM), Clustering Large Applications (CLARA) and Conjugate Gradient linear system solver (CG). We implemented these algorithms in four MapReduce frameworks (Hadoop, HaLoop, Twister and Spark) and in MPI. To be able to adequately measure the overhead of MapReduce implementations in comparison to MPI without comparing the programming language specific overhead, we chose a Java based MPI implementation MpiJava. The following subsections describe the chosen MPI implementation, three algorithms and the also the system configuration.

A. mpiJava

MpiJava [19] provides an object oriented Java interface to MPI that can work with any native MPI library written in C or C++ that implements the MPI standard API, such as OpenMPI [20] or MPICH2 [21]. Java Native Interface (JNI) is used to access MPI commands from the native MPI library and also to pass data between Java and the native language.

We chose MPICH2 as the native MPI library. It is a high-performance MPI implementation which supports a wide range of platforms (desktop systems, shared-memory systems, multicore architectures) and high-speed networks (10 Gigabit Ethernet, InfiniBand, Myrinet) [21]. However, it's latest versions do not provide fault tolerance. When using MpiJava the algorithms still run on Java and only the actual MPI messages are transfered through the underlying native libraries. Thus, the performance of concurrent Java tasks is not affected

and we can directly compare the performance of the data synchronization and the parallelization process in general.

B. Benchmarking algorithms

1) Partitioning Around Medoids (PAM): Partitioning Around Medoids [22] (PAM) is the most naive version of the k-medoid clustering method. The general idea of the k-medoid clustering method is to represent each cluster with it's most central element, the medoid, and to reduce all comparisons between the clusters and other objects into comparisons between the medoids of the clusters and the objects.

To cluster a set of objects into k clusters, PAM first chooses random objects as the initial medoids. For each object in the dataset, it calculates the distance from every medoid and assigns the object to the closest medoid, dividing the dataset into k clusters. At the next step, the medoid positions are recalculated for each of the clusters, choosing the most central object as the new medoid. This two step process is repeated until there is no change from the previous iteration. The whole iteration can be adapted as a single MapReduce job:

- Map:
  - Find the closest medoid and assign the object to it.
  - Input: (cluster id, object)
  - Output: (new cluster id, object)
- Reduce:
  - Find which object is the most central and assign it as a new medoid the cluster.
  - Input: (cluster id, (list of all objects in the cluster))
  - Output: (cluster id, new medoid)

The resulting MapReduce job is repeated until medoid positions of the clusters no longer change.

2) Clustering Large Applications (CLARA): Clustering Large Applications [22] (CLARA) is also an iterative k-medoid clustering algorithm, but in contrast to PAM, it only clusters small random subsets of the dataset to find candidate medoids for the whole dataset. This process is repeated multiple times and the best set of candidate medoids is chosen as the final result. The results of the iterations are independent of each other and do not have to be executed in a sequence.

Everything can be reduced into two different MapReduce jobs. First job chooses a number of random subsets from the input data sets, clusters each of them concurrently using PAM and outputs the results. The second MapReduce job calculates the quality measure for each of the results of the first job, by checking them on the whole data set concurrently inside one MapReduce job. These two MapReduce jobs are outlined as follows. First CLARA MapReduce job:

- Map:
  - Assign a random key to each object.
  - Input: (key, object)
  - Output: (random key, object)
- Reduce:
  - The order of the objects is random after sorting. Read first n objects and perform PAM clustering on the n objects to find k different candidate medoids.
  - Input: (key, list of objects)
  - Output: (key, list of k medoids)

Second CLARA MapReduce job:

- Map:
  - For each object, calculate the distance from the closest medoid. For each object, this is done for each candidate sets, and one output is generated for each.
  - Input: (cluster, object)
  - Output: (candidate set id, distance from the closest medoid) [One output for each candidate set]
- Reduce:
  - Sum the distances with the same candidate set id.
  - Input: (candidate set id, list of distances)
  - Output: (candidate set id, sum(list of distances))

The result of the second job is a list of calculated sums for each candidate set, each representing the total sum of distances from all objects and their closest medoids. The candidate set of medoids with the smallest sum of distances is chosen as the best clustering result.

3) Conjugent Gradient linear system solver (CG): CG [7] is an iterative algorithm for solving algebraic systems of linear equations. It uses the matrix form: $Ax = b$ of a linear system, where A is a known matrix, b is a known vector and x is the solution vector. It first performs an initial inaccurate guess of the solution x and then iteratively improves its accuracy by applying gradient descent. Adapting CG to MapReduce is relatively complex task as it is not possible to directly adapt the whole algorithm to the MapReduce model. The matrix and vector operations used by CG at each iterations can be reduced to the MapReduce model instead.

The input matrices are generally large, but can typically fit into collective memory of computer clusters. The computational complexity of the algorithm is not high and the performed task at every iteration is relatively small which means the ratio between communication and computation is unusually high, especially in comparison to CLARA and PAM. This makes CG a good candidate as an additional benchmarking algorithm for iterative MapReduce frameworks.

C. Experiment setup

All the experiments were performed in Amazon EC2 public cloud on 32+1 instances. The Amazon EC2 instance type was m1.large with 7.5GB memory, 2 x 420 GB storage and 2 cores with 2 EC2 Compute Units each. EC2 computing unit represents the capacity of allocated processing power and 1 EC2 unit is considered to be equivalent to an early-2006 1.7 GHz Xeon processor based on benchmarking results. Amazon Inc did not publish the full details of the underlying hardware at the time of running the experiments. The experiments were

conducted in the same Amazon EC2 availability zone (us-east-1b) to affirm that the experiment results were affected as little as possible. The software environment was set up in Ubuntu 12.04 server operating system. MPI tests were performed using mpiJava 1.2.7, MapReduce tests were performed using Hadoop 1.0.3, Twister 0.9, Spark 0.8.0 and HaLoop revision 408. MpiJava internally used MPICH2 1.4.1p1 for the MPI communication and HaLoop used a modified Hadoop 0.20.0.

None of the framework configurations were optimized in detail to avoid giving any of them an unfair advantage. Optimizations were only performed to assure that the algorithm executions were parallelized in a balanced manner in each of the frameworks and that the number of working processes was equal to the number of cores in the cluster. This included lowering the HDFS block size to 12 MB, changing the number of input files as needed to force computations in Map tasks to be of equal size and specifying the number of reducers to be equal to the number of available cores.

IV. Evaluation results

A. Partitioning Around Medoids (PAM)

Table II provides the runtime results for the PAM kMedoid algorithm implementations. We ran all four implementations with data sets consisting of 13 333, 40 000 and 80 000 2-dimensional points and measured their runtime. The maximum relative standard deviation was 2.6% for MPI, 12% for HaLoop, 13% for Spark, 24.8% for Twister and 10% for Hadoop. MPI implementation was the fastest in all cases being clearly the most efficient solution. Out of the four MapReduce solutions Twister performed the best, and was rather close to MPI.

HaLoop and Spark clearly performed better than Hadoop in every test case but were significantly slower than either MPI or Twister. For the first two datasets, Spark was several times faster than HaLoop, but it achieved better parallel speedup in on the largest dataset and managed to achieve a better runtime when executed on 16 and 32 nodes. Figure 1 provides a comparison of PAM implementations when dealing with the largest dataset to better illustrate the differences between them.

B. Clustering Large Applications (CLARA)

Table III provides the runtime results for the CLARA kMedoid clustering algorithm when the data sets consisted of 1 333 333, 2 666 666 and 4 000 000 2-dimensional points. The maximum relative standard deviation was 0.5% for MPI, 11.2% for HaLoop, 13.4% for Spark, 10.4% for Twister and 13.2% for Hadoop. Compared to PAM, these results show how the implementations can manage a larger dataset when the algorithm is not iterative. In all the MapReduce implementations CLARA required 2 MapReduce jobs, first to randomize the data and to apply sequential PAM on smaller sampled datasets and second to choose the best clustering. MPI again showed the best results, achieving almost perfect

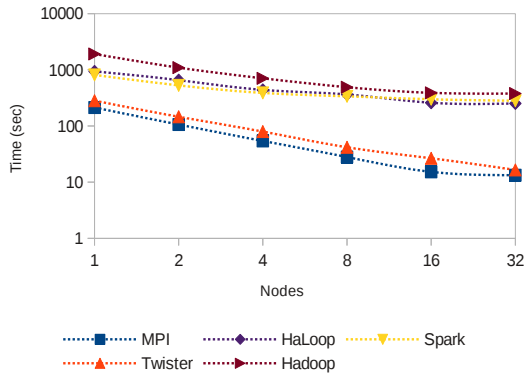

Figure 1. PAM runtime comparison when clustering 80000 objects

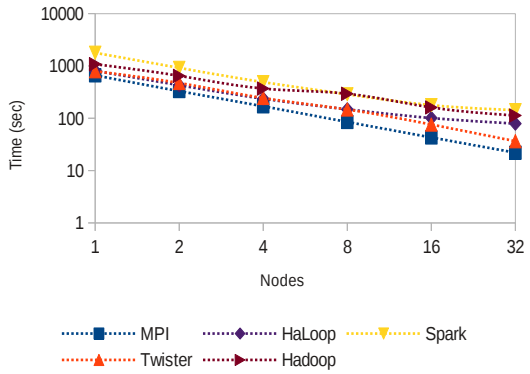

Figure 2. CLARA runtime comparison when clustering 3 million objects

parallel speedup. Both HaLoop and Twister also performed quite well, being constantly better than Hadoop but it were still constantly slower than MPI. However, Twister scaled much better than HaLoop and was twice as fast on 32 nodes for the largest dataset. Spark was clearly inefficient in comparison to the other four solutions, being the only one slower than Hadoop. Already on a single node cluster it was clearly slower than the other frameworks and also was constantly the slowest framework on the biggest data set. The differences of the CLARA implementations when dealing with largest dataset are illustrated on figure 2.

C. Conjugent Gradient linear system solver (CG)

Table IV provides the runtime results for the CG algorithm. The dataset consisted of a matrix of 4 million, 16 million and 64 million elements. In comparison to PAM and CLARA, most of the data does not have to be repartitioned between nodes between iterations and can stay local to the original processes. Of course, this does not apply to Hadoop MapReduce as it has no way of keeping data in memory between iterations. The maximum relative standard deviation was 5.4% for MPI, 15.2% for HaLoop, 20.6% for Spark, 12.5% for Twister and

	MPI			HaLoop			Spark			Twister			Hadoop		
Nodes	13333	40000	80000	13333	40000	80000	13333	40000	80000	13333	40000	80000	13333	40000	80000
1	12.7	57.7	213	645	730	938	34	220	811	13.1	62.1	282	1624	1732	1912
2	6.7	29.0	106	427	485	655	29	147	528	7.15	33.0	146	984	1018	1094
4	3.7	14.7	54.1	313	341	439	24	111	388	4.06	18.7	79.4	634	652	706
8	1.9	7.5	28.0	234	256	364	22	97	337	2.33	11.9	41.8	465	472	490
16	1.2	3.9	15.23	231	235	259	21	88	299	2.66	7.21	26.6	369	378	389
32	1.0	3.8	13.31	198	209	252	30	86	281	3.34	5.05	16.5	359	362	378

TABLE II: Running time (s) of PAM clustering results with different frameworks and data set sizes

	MPI			HaLoop			Spark			Twister			Hadoop		
Nodes	1.3 mil	2.7 mil	4 mil	1.3 mil	2.7 mil	4 mil	1.3 mil	2.7 mil	4 mil	1.3 mil	2.7 mil	4 mil	1.3 mil	2.7 mil	4 mil
1	222	442	660	294	569	806	612	1197	1783	273.6	530.3	792.4	617	836	1081
2	111	221	330	173	294	428	319	618	918	166.6	323.5	472.2	436	473	647
4	58	113	169	113	228	236	181	344	490	81.8	164.0	244.0	253	349	369
8	29	57	85	87	110	148	106	197	287	50.1	98.6	147.0	180	235	294
16	15	29	43	74	83	101	68	122	178	24.0	49.0	75.9	138	132	161
32	8	15	22	72	72	79	57	61	143	15.8	24.1	36.4	92	101	113

TABLE III: Running time (s) of CLARA clustering results with different frameworks and data set sizes

18.2% for Hadoop.

MPI CG was the fastest implementation for the two smalled data set sizes. Moreover, as long as the dataset fits into the memory of one machine, there is actually no speedup from using MPI on multiple nodes as sending messages between processes adds up to a significant overhead. As the number of nodes was increased for the larger dataset, Twister actually performed better than MPI. Considering it was still slower than the MPI's single node execution, it's only shows that Twister has less communication overhead and thus scales better. At the same time it's runtime is degraded by other framework issues, such as object serialization overhead.

Spark also performed well, but with first two datasets there is actually no gain from using multiple nodes and most of the time was actually spent on configuring the Spark processes, loading data to RDD, deploying them and cleaning up after computation was done. Except for the last dataset where Spark actually achieved parallel speedup up to 8 nodes. HaLoop again performed constantly better than Hadoop, but compared to MPI, Twister and Spark the results were still extremely slow, showing that it gains little if any benefit from HaLoop map input caching when large amount of data still needs to processed in reduce tasks and thus need to be transported across machines. The differences between the CG implementations when dealing with largest dataset are illustrated on figure 3.

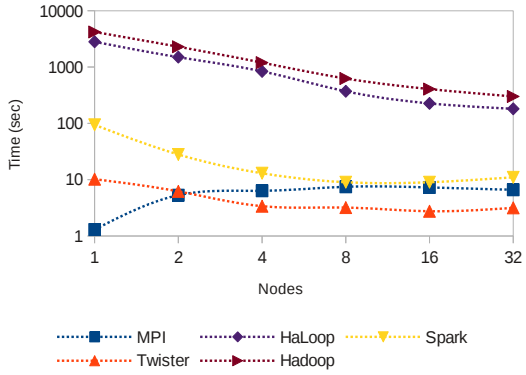

Figure 3. CG runtime comparison with data set size of 64 million elements

D. Summary of the Evaluation

From the runtime experiments it is clear that Spark, Twister and HaLoop can deal with iterative algorithms better than Hadoop but not as well as MPI, with Twister being clearly the fastest of them. While this is somewhat expected, the difference is larger than we envisioned for both HaLoop and Spark and it shows that they have great scope for improvements. Another important thing to note is that while the HaLoop results are constantly in same relation with both MPI (always several times slower) and Hadoop (always around 1.5 times faster), Spark results seem to be greatly affected by the characteristics of the benchmarking algorithms and their dataset composition.

In the case of PAM, there is little benefit from caching the input data in memory in both Spark and HaLoop frameworks as the small objects are constantly repartitioned between clusters, which involves regrouping and transporting them between concurrently running tasks at every iteration.

In the case of CLARA, Spark has great difficulties, which can be attributed to the peculiarities of the data set composition as CLARA dataset consists of millions of very small objects (2D points) which are stored in Spark RDD's. It was the same in the case of PAM, but its dataset is much smaller so the problem did not arise. This problem is especially evident when considering that Spark already has difficulties handling the CLARA dataset when running on a single node and from the fact that is has no trouble with the CG algorithm which requires even more memory to store the input data in memory. In addition, the CG objects themselves (matrices) are much larger. Our hypothesis is that Spark RDD's are inefficient when

	MPI			HaLoop			Spark			Twister			Hadoop		
Nodes	4 mil	16 mil	64 mil	4 mil	16 mil	64 mil	4 mil	16 mil	64 mil	4 mil	16 mil	64 mil	4 mil	16 mil	64 mil
1	0.10	0.35	1.30	154	851	2809	9.0	10.2	93.7	1.71	3.29	10.19	380	1008	4191
2	0.44	1.61	5.30	151	370	1498	9.3	9.2	28.0	1.39	2.14	6.19	299	533	2291
4	0.47	1.71	6.34	125	200	838	10.3	9.6	12.7	1.73	1.85	3.38	298	388	1197
8	0.53	1.65	7.46	130	176	371	11.3	11.7	9.0	1.44	1.57	3.19	298	303	625
16	0.60	1.73	7.27	130	127	226	11.3	11.3	9.0	1.51	1.57	2.75	297	297	406
32	0.65	1.82	6.60	134	130	181	13.0	12.0	11.3	1.80	2.15	3.16	298	298	300

TABLE IV: Running time (s) of Conjugate Gradient results with different frameworks and data set sizes

they consist of a large number of very small objects. We will investigate it in detail in our future work.

The CG experiment was the closest Spark could get to MPI results. When processing the largest dataset, Spark results are directly comparable to MPI results (11 vs 6.6 seconds) on 32 nodes if we take into account that most of the time in Spark was actually spent on framework overhead of scheduling Spark processes, initiation and cleanup, as is evident from the fact that runtime never falls below 9 seconds. Similarly, the MPI runtime is also greatly affected by its communication overhead which depends on the size of the messages and the number of processes, as seen from the increase in MPI CG runtime in table IV for the largest dataset as the cluster grows.

While Twister is the fastest of the investigated MapReduce frameworks, it is not as usable as Spark when stability or fault tolerance is required. Twister had the most issues and crashes that greatly complicated executing the performance tests in any automated manner and it's fault recovery system is not ideal for longer running tasks, as only the Map input and data passed by the main program is recovered. Any data kept in memory across iterations would still be unrecoverable.

V. Related Work

Yingyi Bu et al. have investigated [23] using HaLoop instead of Hadoop for large-scale iterative data analysis by implementing PageRank, descendant query and k-means algorithms in both frameworks and analyzing the results on different datasets. HaLoop performed better than Hadoop in each of their test cases showing that it is more suitable than Hadoop for these types of application. However, without comparing HaLoop to MPI or other widely used lower level distributed computing solutions, it stays unclear how efficient HaLoop really is in comparison to the de facto tools used in the scientific computing field. A number of studies have compared [5], [24] Twister to Hadoop MapReduce and MPI for typical scientific algorithms and showed that Twister can greatly reduce the overhead of iterative MapReduce applications.

Bulk Synchronous Parallel (BSP) model is promising to provide the same advantages as MapReduce. While the initial implementations (Oxford BSPlib [25], BSPonMPI [26]) were not taken into a wide spread use, a new wave of BSP frameworks have been initiated after Google published using the BSP model for their new large scale graph processing framework Pregel [27]. It was developed to address MapReduce's inability of supporting iterative graph processing algorithms.

Pregel uses checkpointing to provide implicit fault tolerance. It stores the state of the application in persistent storage between supersteps at user configured intervals. Google's implementation is proprietary and can only be used in-house to solve various graph related problems but a number of Pregel like frameworks have since been created, such as Apache Giraph [28], Stanford GPS [29] and Hama [30].

The first two frameworks follow Pregel very closely. Apache Giraph [28] leverages the existing Hadoop infrastructure to execute MapReduce like jobs that use BSP model internally and the computation is divided into a sequence of supersteps. Stanford GPS [29] includes several features not present in either Google Pregel or Apache Giraph. It provides support for algorithms that include global as well as vertex-centric computations, the ability to repartition the graph during processing and partitioning adjacency lists of high-degree vertices to reduce the amount of communication. Pregel, Apache Giraph and Stanford GPS are specifically designed for graph algorithms, and while it is almost always possible to represent other types of distributed applications as graph computations by restructuring them, it would also require extra effort from the user and may also lower the parallel efficiency of the result.

Hama [30] was originally envisioned as a framework that provides Hadoop-compatible interfaces to different computation engines such as MapReduce and Pregel. It has been actively developed as an open source framework but has decided to drop the support for MapReduce and only retains the BSP model in its latest versions. Authors of the Hama have measured how it [30], [31] performs in comparison to Hadoop MapReduce and MPI. The show predictably that Hama performs better than Hadoop MapReduce. Despite being in active development for a number of years Hama still does not include a fully working fault tolerance mechanisms.

VI. Conclusions

The goal of this work was to evaluate how suitable the alternative MapReduce frameworks are for more complex scientific algorithms and how well they perform in comparison to MPI and Hadoop. We compared the performance of three iterative MapReduce frameworks Spark, Twister and HaLoop to both MPI and Hadoop by implementing three benchmarking algorithms on each of them and measuring their runtime under 6 different cluster and 3 data set sizes.

The results show that almost in all cases Twister, HaLoop and Spark are more efficient than Hadoop, confirming that they

are able to deal with iterative applications much better. At the same time, the results also show that both Spark and HaLoop can not directly compete with the MPI implementations of the same algorithms. In every single test case MPI implementation was faster. Twister performed better than Spark and HaLoop in every case but was still slower than MPI.

We must also take into account that there are many advantages in using MapReduce-like frameworks that can significantly simplify adapting algorithms for parallel processing. From the ease of partitioning and distributing data through the Hadoop Distributed File System (HDFS) to near automatic fault tolerance and parallelization. Still, the efficiency of the result can not be ignored for scientific computing applications. Executing long running scientific experiments and simulations requires a large number of computing resources and whether we are using supercomputers, grids or cloud resources, the cost and limit of such resources are crucial. Thus the chosen higher level distributed computing framework should at least be comparable to MPI and not be a several times less efficient.

It is evident from the results that only Twister has a comparable performance to MPI for all the benchmarks and thus is a good candidate for iterative applications when performance is important. However, Twister loses many of the advantages typically attributed to MapReduce frameworks because it does not use a distributed file system and it is generally unstable. Spark's results depended strongly on the algorithm characteristics and thus can only be applied in some cases. HaLoop results were only marginally improved over Hadoop in comparison to MPI and as it is in a prototype state, it is not really usable in real applications.

ACKNOWLEDGMENT

This research is supported by AWS in Education Grant, European Regional Development Fund through EXCS, Estonian IT Academy, Estonian Science Foundation grant PUT360 and Target Funding theme SF0180008s12.

REFERENCES

[1] C. Bunch, B. Drawert, and M. Norman, "MapScale: A Cloud Environment for Scientific Computing," University of California, Computer Science Department, Tech. Rep., 2009.

[2] M. Kim, H. Lee, and Y. Cui, "Performance evaluation of image conversion module based on mapreduce for transcoding and transmoding in smccse," in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, 2011, pp. 396–403.

[3] T. Dalman, T. Dörnemann, E. Juhnke, M. Weitzel, W. Wiechert, K. Nöh, and B. Freisleben, "Cloud mapreduce for monte carlo bootstrap applied to metabolic flux analysis," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 582 – 590, 2013.

[4] S. N. Srirama, P. Jakovits, and E. Vainikko, "Adapting scientific computing problems to clouds using mapreduce," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 184–192, 2012.

[5] P. Jakovits, S. N. Srirama, and E. Vainikko, "Mapreduce for scientific computing - viability for non-embarrassingly parallel algorithms," in *Applications, Tools and Techniques on the Road to Exascale Computing*, vol. 22. IOS Press, 2012, pp. 117–124.

[6] P. Jakovits and S. N. Srirama, "Clustering on the cloud: Reducing clara to mapreduce," in *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, ser. NordiCloud '13. New York, NY, USA: ACM, 2013, pp. 64–71.

[7] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," Pittsburgh, PA, USA, Tech. Rep., 1994.

[8] C. Robert and G. Casella, *Monte Carlo statistical methods*. Springer Verlag, 2004.

[9] P. Jakovits, I. Kromonov, and S. Srirama, "Monte carlo linear system solver using mapreduce," in *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, dec. 2011, pp. 293 –299.

[10] T. Cheatham, A. Fahmy, D. C. Stefanescu, and L. G. Valiant, "Bulk synchronous parallel computing – a paradigm for transportable software," in *In Proc. IEEE 28th Hawaii Int. Conf. on System Science*. Society Press, 1995, pp. 268–275.

[11] P. Jakovits, S. N. Srirama, and I. Kromonov, "Viability of the bulk synchronous parallel model for science on cloud," in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, 2013, pp. 41–48.

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *2nd USENIX conf. on Hot topics in cloud computing*, ser. HotCloud'10, 2010, p. 10.

[13] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. ACM, 2010, pp. 810–818.

[14] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," in *36th International Conference on Very Large Data Bases*, Singapore, September 14–16, 2010.

[15] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, January 2008.

[16] Apache Software Foundation, "Hadoop," February 2014. [Online]. Available: http://wiki.apache.org/hadoop/

[17] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science and Engineering*, vol. 11, no. 4, pp. 29–41, July/Aug 2009.

[18] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientific analyses," in *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, dec. 2008, pp. 277 – 284.

[19] B. Carpenter, "mpiJava: A Java Interface to MPI," in *First UK Workshop on Java for High Performance Network Computing, Europar 98*, 1998.

[20] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 353–377, 2004.

[21] Argonne National Laboratory, "MPICH2 - a high-performance and widely portable implementation of the Message Passing Interface (MPI) standard," February 2014. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpich2/

[22] L. Kaufman and P. Rousseeuw, *Finding Groups in Data An Introduction to Cluster Analysis*. New York: Wiley Interscience, 1990.

[23] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "The haloop approach to large-scale iterative data analysis," *The VLDB Journal*, vol. 21, no. 2, pp. 169–190, Apr. 2012.

[24] J. Ekanayake, X. Qiu, T. Gunarathne, S. Beason, and G. Fox, "High Performance Parallel Computing with Cloud and Cloud Technologies," Indiana University, Tech. Rep., 2009. [Online]. Available: http://grids.ucs.indiana.edu/ptliupages/publications/CGLCloudReview.pdf

[25] J. M. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao *et al.*, "Bsplib: The bsp programming library," *Parallel Computing*, vol. 24, no. 14, pp. 1947 – 1980, 1998.

[26] W. J. Suijlen, "BSPonMPI," February 2014. [Online]. Available: http://bsponmpi.sourceforge.net/

[27] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*. ACM, 2010, pp. 135–146.

[28] Apache Software Foundation, "Giraph," February 2014. [Online]. Available: http://giraph.apache.org/

[29] S. Salihoglu and J. Widom, "Gps: A graph processing system," Stanford University, Technical Report. [Online]. Available: http://ilpubs.stanford.edu:8090/1039/

[30] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," *Cloud Computing Technology and Science, IEEE International Conference on*, pp. 721–726, 2010.

[31] A. S. Foundation, "Hama Benchmarks," February 2014. [Online]. Available: http://wiki.apache.org/hama/Benchmarks