Future Generation Computer Systems 28 (2012) 184–192

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Adapting scientific computing problems to clouds using MapReduce

Satish Narayana Srirama, Pelle Jakovits *, Eero Vainikko

Distributed Systems Group, Institute of Computer Science, University of Tartu, J. Liivi 2, 50409 Tartu, Estonia

ARTICLE INFO

Article history:
Received 19 January 2011
Received in revised form
10 May 2011
Accepted 28 May 2011
Available online 12 June 2011

Keywords:
Scientific computing
Cloud computing
MapReduce
Hadoop
Iterative algorithm
Twister

ABSTRACT

Cloud computing, with its promise of virtually infinite resources, seems to suit well in solving resource greedy scientific computing problems. To study this, we established a scientific computing cloud (SciCloud) project and environment on our internal clusters. The main goal of the project is to study the scope of establishing private clouds at the universities. With these clouds, students and researchers can efficiently use the already existing resources of university computer networks, in solving computationally intensive scientific, mathematical, and academic problems. However, to be able to run the scientific computing applications on the cloud infrastructure, the applications must be reduced to frameworks that can successfully exploit the cloud resources, like the MapReduce framework. This paper summarizes the challenges associated with reducing iterative algorithms to the MapReduce model. Algorithms used by scientific computing are divided into different classes by how they can be adapted to the MapReduce model; examples from each such class are reduced to the MapReduce model and their performance is measured and analyzed. The study mainly focuses on the Hadoop MapReduce framework but also compares it to an alternative MapReduce framework called Twister, which is specifically designed for iterative algorithms. The analysis shows that Hadoop MapReduce has significant trouble with iterative problems while it suits well for embarrassingly parallel problems, and that Twister can handle iterative problems much more efficiently. This work shows how to adapt algorithms from each class into the MapReduce model, what affects the efficiency and scalability of algorithms in each class and allows us to judge which framework is more efficient for each of them, by mapping the advantages and disadvantages of the two frameworks. This study is of significant importance for scientific computing as it often uses complex iterative methods to solve critical problems and adapting such methods to cloud computing frameworks is not a trivial task.

1. Introduction

Scientific computing is a field of study that applies computer science to solve typical scientific problems. It should not be confused with just computer science. Scientific computing is usually associated with large scale computer modeling and simulation and often requires a large amount of computer resources. Cloud computing [1] suits well in solving these scientific computing problems, with its promise of provisioning virtually infinite resources.

In the adaptation of resource-intensive applications for the clouds, the applications must be reduced to frameworks that can successfully exploit the cloud resources, which is the approach we are studying in our Scientific Computing on the Cloud (SciCloud) project [2]. Generally, cloud infrastructure is based on commodity computers, which are cost effective but are bound to fail regularly. This can cause serious problems as the software has to adapt to failures. To deal with hardware or network failures in a distributed system, the best course usually is to replicate important data and retry computations which fail. There are also distributed computing frameworks that provide fault tolerance by design and one such framework is the MapReduce [3] framework.

MapReduce was first developed by Google as a parallel computing framework to perform distributed computing on a large number of commodity computers. Since then, it has gained popularity as a cloud computing framework on which to perform automatically scalable distributed applications. Google MapReduce implementation is proprietary and this has resulted in the development of open source counterparts like Hadoop [4] MapReduce. Hadoop is a Java software framework inspired by Google's MapReduce and Google File System [5] (GFS). The Hadoop project is being actively developed by Apache and is widely used both commercially and for research, and as a result has a large user base and adequate documentation.

While the automatic scalability is very attractive when working with distributed applications, the structure of a MapReduce application is very strict. It is not trivial to reduce complex algorithms to the MapReduce model and there is no guarantee that the resulting

* Corresponding author. Tel.: +372 55638589.
E-mail addresses: srirama@ut.ee (S.N. Srirama), jakovits@ut.ee, jakovits@smail.ee (P. Jakovits), eero@ut.ee (E. Vainikko).

0167-739X/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2011.05.025

MapReduce algorithms are effective. Previous work has shown that MapReduce is well suited for simple, often embarrassingly parallel problems. Google show in their paper [3] that they use MapReduce for a wide variety of problems like large-scale indexing, graph computations, machine learning and extracting specific data from a huge set of indexed web pages. Other related work [6] shows that MapReduce can be successfully used for graph problems, like finding graph components, barycentric clustering, enumerating rectangles and enumerating triangles. MapReduce has also been tested for scientific problems [7]. It performed well for simple problems like the Marsaglia polar method for generating random variables and integer sort.

However, MapReduce has also been shown to have significant problems [7] with more complex algorithms, like conjugate gradient, fast Fourier transform and block tridiagonal linear system solver. Moreover, most of these problems use iterative methods to solve them, indicating that MapReduce may not be well suited for algorithms that have an iterative nature. However, there is more than one type of iterative algorithm. To study if MapReduce model is unsuitable for all iterative algorithms or only a certain subset of them, we devised a set of classes for scientific algorithms. Algorithms are divided between these classes by how difficult it is to adapt them to the MapReduce model and their resulting structure. To be able to compare the classes to each other, we selected and adapted algorithms from each class to the MapReduce model and studied their efficiency and scalability. Such a classification allows us to precisely judge which algorithms are more easily adaptable to the MapReduce model and what kind of effect belonging to a specific class has on the parallel efficiency and scalability of the adapted algorithms.

The rest of the paper is structured as follows. Section 2 briefly introduces the SciCloud project. Section 3 describes the Hadoop MapReduce model on our SciCloud infrastructure and Section 4 describes the different classes for iterative algorithms. Section 5 outlines the algorithms that were implemented and analyzed. Section 6 describes an alternative MapReduce framework called Twister and produces the analysis of the algorithms on the framework. Section 7 mentions the related work and Section 8 concludes the paper and describes the future research directions in the context of the SciCloud project.

2. SciCloud

The main goal of the scientific computing cloud (SciCloud) project [2] is to study the scope of establishing private clouds at universities. With these clouds, students and researchers can efficiently use the already existing resources of university computer networks, in solving computationally intensive scientific, mathematical, and academic problems. Traditionally, such computationally intensive problems were targeted by batch-oriented models of the GRID computing domain. SciCloud tries to achieve this with more interactive and service oriented models of cloud computing that fit a larger class of applications. It targets the development of a framework, including models and methods for establishment, proper selection, state and data management, auto scaling and interoperability of the private clouds. Once such clouds are feasible, they can be used to provide better platforms for collaboration among interested groups of universities and in testing internal pilots, innovations and social networks. SciCloud also focuses on finding new distributed computing algorithms and tries to reduce some of the scientific computing problems to MapReduce algorithm.

While there are several public clouds on the market, Google Apps (examples include Google Mail, Docs, Sites, Calendar etc.), Google App Engine [8] (which provides an elastic platform for Java and Python applications with some limitations) and Amazon EC2 [9] are probably the most known and widely used. Amazon EC2 allows full control over the virtual machine, starting from the operating system. It is possible to select a suitable operating system and platform (32 and 64 bit) from many available Amazon Machine Images (AMI) and several possible virtual machines, which differ in CPU power, memory and disk space. This functionality allows us to freely select suitable technologies for any particular task. In the case of EC2, the price for the service depends on the machine size, its uptime, and the used bandwidth in and out of the cloud.

There are also free implementations of cloud infrastructure e.g. Eucalyptus [10]. Eucalyptus allows the creation of private clouds compatible with Amazon EC2. Thus the cloud computing applications can initially be developed in private clouds and can later be scaled to the public clouds. This is of great help for the research and academic communities, as the initial expenses of experiments can be reduced by a great extent. With this primary goal we have set up SciCloud on a cluster consisting of 8 nodes of SUN FireServer Blade system with 2-core AMD Opteron Processors, using Eucalyptus technology. The cluster was later extended by 2 nodes with double quad-core processors and 32 GB memory per node, plus 4 more nodes with a single quad-core processor and 8 GB of memory each.

While several applications are obvious from such a private cloud setup, we have used it in solving some of our research problems in distributed computing and mobile web services domains [2]. In the mobile web services domain, we scaled our Mobile Enterprise [11] to the loads possible in cellular networks. A Mobile Enterprise can be established in a cellular network by participating Mobile Hosts, which act as web service providers from smart phones, and their clients. Mobile Hosts enable seamless integration of user-specific services to the enterprise, by following web service standards [12], also on the radio link and via resource constrained smart phones. Several applications were developed and demonstrated with the Mobile Host in health care systems, collaborative m-learning, social networks and multimedia services domains [11]. We shifted some of the components and load balancers of Mobile Enterprise to the SciCloud and proved that the Mobile Web Services Mediation Framework [13] and components are horizontally scalable. More details of the analysis are available at [14]. Apart from helping us in our research, SciCloud also has several images supporting in data mining and bio-informatics domains.

3. SciCloud Hadoop framework

With the intent of having a setup for experimenting with MapReduce based applications, we have set up a dynamically configurable SciCloud Hadoop framework. We used the Hadoop cluster to reduce some of the scientific computing problems like CG to MapReduce algorithms. The details are addressed in this section.

MapReduce is a programming model and a distributed computing framework. It was first developed by Google to process very large amounts of raw data that it has to deal with on a daily basis, like indexed Internet documents and web requests logs, which grows every day. Google uses MapReduce to process data across hundreds or thousands of commodity computers. MapReduce applications get a list of key-value pairs as an input and consist of two main methods, Map and Reduce. The Map method processes each key-value pair in the input list separately, and outputs one or more key-value pairs as a result.

$$\text{map}(\text{key}, \text{value}) \Rightarrow [(\text{key}, \text{value})].$$

The Reduce method aggregates the output of the Map method. It gets a key and a list of all values assigned to this key as an input, performs user defined aggregation on it and outputs one or more key-value pairs.

$$\text{reduce}(\text{key}, [\text{value}]) \Rightarrow [(\text{key}, \text{value})].$$

Users only have to produce these two methods to define a MapReduce application; the framework takes care of everything else, including data distribution, communication, synchronization and fault tolerance. This makes writing distributed applications with MapReduce much easier, as the framework allows the user to concentrate on the algorithm and is able to handle almost everything else. Parallelization in the MapReduce framework is achieved by executing multiple Map and Reduce tasks concurrently on different machines in the cluster.

However, Google implementation of MapReduce is proprietary. Apache Hadoop [4] is an open source implementation of MapReduce written in Java. Apart from MapReduce, Hadoop also provides the Hadoop Distributed File System [15] (HDFS) to reliably store data across hundreds of computers. HDFS is based on, and thus conceptually similar, to the Google File System (GFS) [5]. The Hadoop MapReduce framework uses HDFS to store both the input and output of MapReduce applications in a distributed fashion. A simple Hadoop cluster consists of $n \geq 1$ machines running the Hadoop software. The cluster is a single master cluster with a varying number of slave nodes. Slave nodes can act as both the computing nodes for the MapReduce and as data nodes for the HDFS. Apache Hadoop is in active development and is used both commercially and for research.

To analyze the performance of MapReduce for scientific computing, we set up a small Hadoop cluster in the SciCloud. The cluster is composed of one master and sixteen slave nodes. Only the slaves act as MapReduce task nodes, resulting in 16 parallel workers where the MapReduce tasks can be executed. Each node is a virtual machine with 2.2 GHz CPU, 500 MB RAM and 10 GB disk space allocated for the HDFS, making the total size of the HDFS 160 GB. More nodes can be added to the cluster dynamically, when needed. We have scripts that can add more slaves to the framework and can configure to the Master. We also have support for Auto scaling, which starts more slave nodes based on the observed loads of individual instances. The details will be addressed by our future publications.

4. Algorithm classes

We have devised a set of classes for scientific algorithms based on how difficult it is to adapt them to the MapReduce model and what steps are required. The algorithms are divided into different classes as follows:

- Algorithms that can be adapted as a single execution of a MapReduce model.
- Algorithms that can be adapted as a sequential execution of a constant number of MapReduce models.
- Algorithms where the content of one iteration is represented as an execution of a single MapReduce model.
- Algorithms where the content of one iteration is represented as an execution of multiple MapReduce models.

First class can be considered to represent embarrassingly parallel algorithms and the second class easily parallelizable algorithms. Third and fourth represent iterative algorithms, where some type of synchronization must be performed between each iteration; for example to check the ending condition or to aggregate and broadcast the result of the previous iteration. Algorithms belonging to the fourth class are considered to be more complex iterative algorithms where only some operations in each iteration can be parallelized completely. Algorithms belonging to class 4 are generally difficult to parallelize efficiently, which is even more difficult to achieve when adapting them to the MapReduce model. To study how belonging to a specific class affects the efficiency and scalability of an algorithm, we adapted algorithms from each class to MapReduce and analyzed the results. The algorithms we chose are described in the following chapter.

Apart from belonging to the specific class, the parallel efficiency and scalability is also affected by the individual characteristics of the algorithm. For example, it depends on how large part of the computation stays outside of the MapReduce model. Checking the ending condition in an iterative algorithm or aggregating and processing the final result, when it cannot be done in the reducer method in MapReduce, means that most often some part of the iterative algorithm must be executed outside parallelism, which decreases the parallel efficiency of the whole algorithm. Also, algorithms that belong to the second class can become less efficient if the number of sequential MapReduce model executions is large. Switching between different MapReduce models acts as a synchronization step and the input data for each different MapReduce execution must be processed again, meaning there might be no practical difference between the second and third classes when the number of steps in the second is comparable to the number of iterations in the third.

Furthermore, the parallel efficiency does not only depend on how the algorithm was adapted to the MapReduce model or on the inherit characteristics of the algorithm itself, but also on the environment it is executed in. Executing MapReduce applications on different MapReduce frameworks can have a significant impact on the running time of the application, also depending on which class the algorithms used in this application belong to.

5. Reducing iterative algorithms to MapReduce

We chose one algorithm from each of the algorithm classes outlined in Section 3 to illustrate the different design choices and problems that can arise when adapting scientific computing problems to the Hadoop MapReduce framework. These algorithms are:

- Conjugate Gradient (CG).
- Two different *k*-medoid clustering algorithms:
  - Partitioning Around Medoids (PAM).
  - Clustering Large Application (CLARA).
- Factoring integers.

CG belongs to class 4, PAM to class 3, CLARA to class 2, and factoring integers is an example of class 1 and embarrassingly parallel algorithms. For each of these algorithms we provide a short description, reasoning why they belong to the given class, the steps taken to adapt them to the MapReduce model and the results of our experiments.

5.1. Conjugate Gradient

Conjugate Gradient [16] (CG) is an iterative algorithm for solving systems of linear equations in matrix form:

$$Ax = b$$

where the A is a known matrix, b is a known vector and x is the solution vector of this linear system. The general idea of CG is to perform an initial inaccurate guess of the solution x and then improve its accuracy at every following iteration.

CG is a relatively complex algorithm, it is not possible to directly adapt the whole algorithm to the MapReduce model. Instead, the matrix and vector operations used by CG at each iterations are reduced to the MapReduce model. Because of this it directly belongs to the fourth algorithm class. Matrix and vector operations that need to be adapted to the MapReduce model separately are:

- Matrix–vector multiplication.
- Dot product.
- Two vector addition.
- Vector and scalar multiplication.

Table 1
Run times for the CG implementation in MapReduce under varying cluster size.

Unknowns	24	500	1000	2000	4000	6000	8000
1 node	259	261	327	687	1938	3810	7619
2 nodes	255	259	298	507	1268	2495	4185
4 nodes	255	236	281	360	721	1374	2193
8 nodes	251	251	291	397	563	824	1246
16 nodes	236	240	278	297	338	511	809

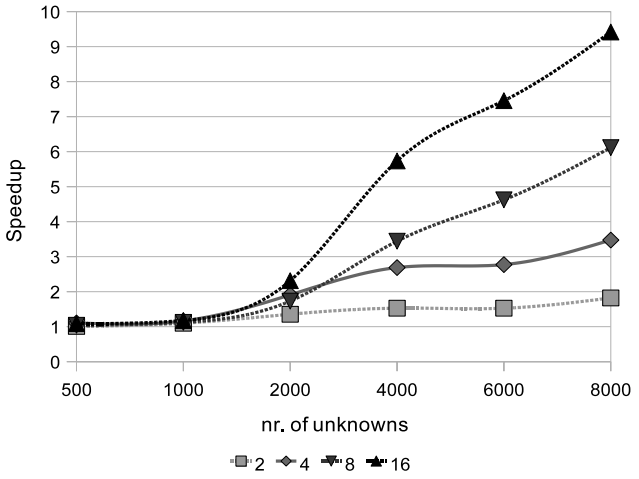

Fig. 1. Speedup for Conjugate Gradient algorithm with different number of nodes.

Every time one of these operation is used in the CG, a new MapReduce job is executed. As a result, multiple MapReduce jobs are executed at every iteration. This is not the most efficient way as it takes time for the Hadoop framework to schedule, start up and finish MapReduce jobs. It can be viewed as MapReduce job latency and executing multiple jobs at each iteration adds up to a significant overhead.

Additionally, in Hadoop, the matrix A is stored on the HDFS and is used as an input for the matrix–vector multiplication operation at every iteration. In the Hadoop MapReduce framework it is not possible to cache the input between different executions of MapReduce jobs, so every time this operation is executed, the input must be read again from the file system. As the matrix A values never change between iterations, the exact same work is repeated at every iteration. This adds up to a significant additional overhead.

Experiments were run with different numbers of parallel nodes to be able to calculate parallel speedup. Parallel speedup measures how many times the parallel execution is faster than running the same MapReduce algorithm on a single node. If it is larger than 1, it means there is at least some gain from doing the work in parallel. Speedup which is equal to the number of nodes is considered ideal and means that the algorithm has a perfect scalability. Run times for the CG algorithm are shown on Table 1 and calculated speedup is shown on Fig. 1.

It took 220 s to solve a system with only 24 unknowns in a 16 node cluster, which is definitely very slow for solving a linear system with such a small number of calculations needed. Unfortunately, the tests solving larger systems also showed that the CG MapReduce algorithm does not improve as the size of the data increases. For example, a linear system with 8000 unknowns took almost 2 h to solve using the MapReduce algorithm. These results indicate that most of the time in the MapReduce CG is spent on the background tasks and not on the actual calculations.

5.2. Partitioning Around Medoids

Partitioning Around Medoids [17] (PAM) is an iterative k-medoid clustering algorithm, that has significant value in the datamining domain. The general idea of a k-medoid clustering is that each cluster is represented by its most central element, the medoid, and all comparisons between objects and clusters are reduced into comparisons between objects and the medoids of the clusters.

To cluster a set of objects into k different clusters, the PAM algorithm first chooses k random objects as the initial medoids. As a second step, for each object in the dataset, the distances from each of the k medoids is calculated and the object is assigned to the cluster with the closest medoid. As a result, the dataset is divided into k different clusters. At the next step the PAM algorithm recalculates the medoid positions for each of the clusters, choosing the most central object as the new medoid. This process of dividing the objects into clusters and recalculating the cluster medoid positions is repeated, until there is no change from the previous iteration, meaning the clusters have become stable.

Similar to CG, PAM makes an initial guess of the solution, in this case the clustering, and at each following iteration it improves the accuracy of the solution. Also, as with CG, it is not possible to reduce the whole algorithm to the MapReduce model. However, the content of a whole iteration can be reduced to the MapReduce model, showing that PAM belongs to the third algorithm class. The resulting MapReduce job can be expressed as:

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

The Map method recalculates to which cluster each object belongs to, and the Reduce method finds a new center for each of the resulting clusters. This MapReduce job is repeated until medoid positions of the clusters no longer change.

Similar to CG, PAM also has issues with job lag and rereading the input from the file system at every iteration, because a new MapReduce job is executed at each time.

5.3. Clustering Large Applications

Clustering Large Applications [17] (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. Differently from the PAM, the results of the iterations are independent of each other and do not have to be executed in a sequence.

As a result, it is possible to execute the content of the iterations concurrently in separate tasks and remove the iterative structure of the algorithm. Everything can be reduced into two different MapReduce jobs, both executing different tasks. The 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. As a result of having only two MapReduce jobs, the job latency stays minimal and the input data set is only read twice. These two MapReduce jobs are outlined as follows:
First CLARA MapReduce job:

Table 2
Run times for the PAM algorithm.

Objects	10 000	25 000	50 000	75 000	100 000
1 node	1389	1347	2014	3620	6959
2 nodes	1133	1697	1826	2011	6130
4 nodes	803	782	1156	2562	2563
8 nodes	635	627	1513	1084	1851
16 nodes	297	497	432	761	1029

Table 3
Run times for the CLARA algorithm.

Objects (thousands)	25	50	100	500	1000	5000	10 000
1 node	117	118	125	183	261	819	1517
2 nodes	79	84	89	150	215	476	832
4 nodes	61	66	72	120	127	316	486
8 nodes	52	56	61	114	124	218	320
16 nodes	44	50	58	99	98	104	156

- Map:
  - Assign a random key to each object.
  - Input: (key, object).
  - Output: (random key, object).
- Reduce:
  - Read first n objects, which are sorted in the ascending order of the keys. Because the keys were assigned randomly, the order of the objects is random after sorting. 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, find the closest medoid and calculate the distance from it. For each object, this is done as many times as there were candidate sets of medoids, 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, each representing the total sum of distances from all objects and their closest medoids, one for each candidate set. The candidate set of medoids with the smallest sum of distances between objects and their closest medoids is chosen as the best clustering. As the number of different MapReduce jobs executed is always two, this algorithm belongs to the second class.

From the experiment results (Tables 2, 3 and Figs. 2, 3) it is possible to see that the CLARA MapReduce algorithm works much faster than PAM, especially when the number of objects in the dataset increases. PAM was not able to handle datasets larger than 100 000 objects while CLARA could cluster datasets consisting of millions or even tens of millions of objects. It should also be noted that the time to cluster the smallest dataset is quite large for both CLARA and PAM. This is because the background tasks of the MapReduce framework are relatively slow to start, so each separate MapReduce job that is started slows down the algorithm. This affects PAM more greatly than CLARA because PAM consists of many MapReduce job iterations while CLARA only uses two MapReduce jobs.

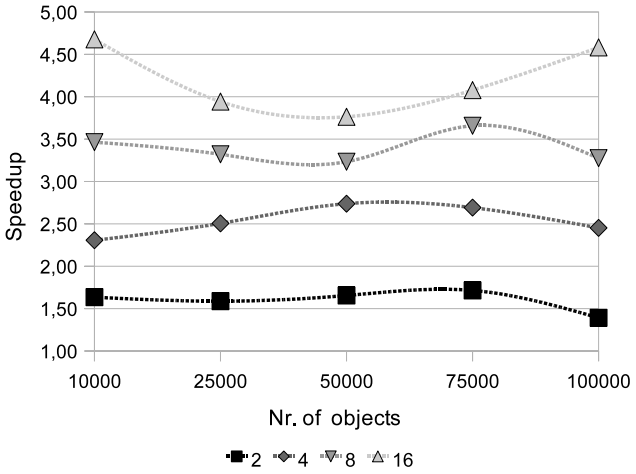

Fig. 2. Parallel speedup for PAM with different numbers of nodes.

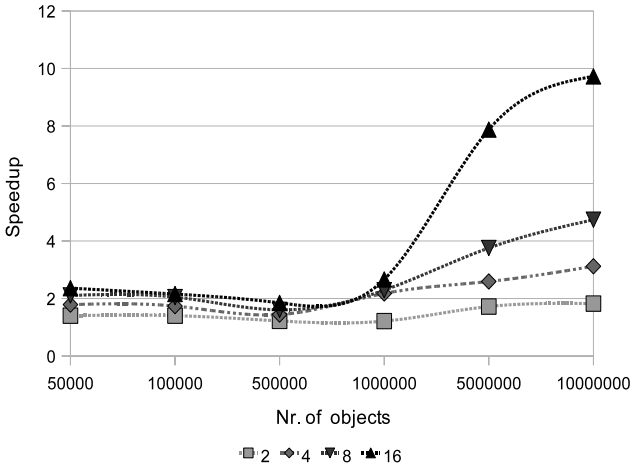

Fig. 3. Parallel speedup for the CLARA algorithm with different number of nodes.

5.4. Factoring integers

Factoring integers is a method for dividing an integer into a set of prime numbers that make up the original number by multiplying them all. For example the factors of a number 21 are 3 and 7. Factoring integers is used for example to break RSA cryptosystem.

In this case we chose the most basic method of factoring integers, the trial division. This method is not used in practice, as it is relatively slow and there exist much faster methods like the general number field sieve [18]. But we chose this method purely to illustrate adapting an embarrassingly parallel problem, belonging to the first class, to the MapReduce model as comparison to the other three algorithms.

To factor a number using trial division, all possible factors of the number are checked to see if they divide the number evenly. If one of them does, then it is a factor. This can be adopted to the MapReduce model, by dividing all possible factors into multiple subgroups and checking each of them in a separate Map or Reduce task concurrently:

- Map:
  - Gets a number to be factored as an input, finds the square root of the number and divides the range from 2 to $\sqrt{\text{number}}$ into n smaller ranges, and outputs each of them.
  - Input: (key, number).
  - Output: (id, (start, end, number)) [one output for each range, n total].

Table 4
Run times for the integer factorization with different numbers of nodes.

Digits	17	18	19	20	21
1 node	51	142	361	2058	6767
2 nodes	37	67	188	1117	3271
4 nodes	30	50	120	512	1622
8 nodes	27	36	70	299	887
16 nodes	27	38	59	215	566

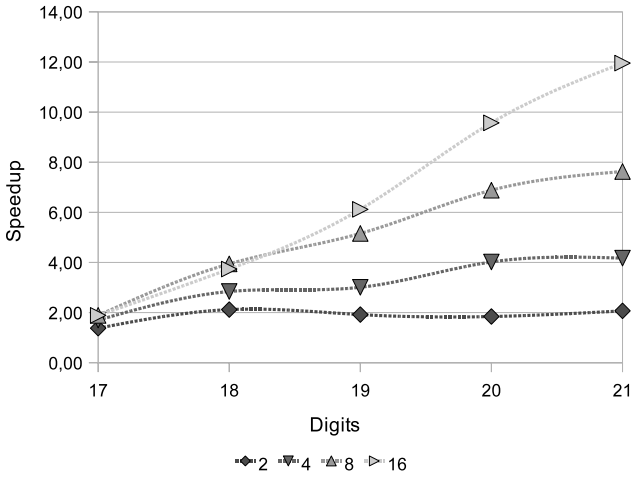

Fig. 4. Parallel speedup for the integer factorization with different numbers of nodes.

- Reduce:
  - Gets a number and a range, in where to check for factors, as an input and finds if any of the numbers in this range divide the number evenly.
  - Input: (id, (start, end, number)).
  - Output: (id, factor).

As a result, in contrast to the previous algorithms, this algorithm is reduced to a single MapReduce job, meaning there is no overhead from executing multiple jobs in sequence and why this algorithm belongs to the first algorithm class.

The run times for the integer factorization are given on the Table 4 and speedup is shown on Fig. 4. From the Fig. 4 it is possible to see that when the factored number is small, there is only a small advantage in using multiple workers in parallel. The speedup is slightly above 1 for 2 node cluster and only reaches 2.22 in 16 node cluster. This is because the number of calculations done was relatively small compared to the background tasks of the framework. However, with the increase of the size of the input number, the speedup started to grow significantly. With the input size of 21 digits, the speedup for two and four node executions was 2.07 and 4.17, showing that there is an ideal gain from using multiple nodes to find the factors when the size of the input is large enough. With a larger number of nodes the speedup does not reach the number of nodes, indicating that calculations were not long enough to get the full benefit from using 16 nodes. The calculated speedup numbers suggest that this algorithm has a good scalability and that algorithms belonging to the first class can be very suitable for the Hadoop MapReduce framework.

5.5. Summary of the analysis

From the results of our experiments we observed that Hadoop MapReduce has several problems with iterative algorithms. Complex iterative algorithms of the class 4 may require one or more MapReduce jobs to be executed at every iteration. However, if the number of iterations is large, then executing that many

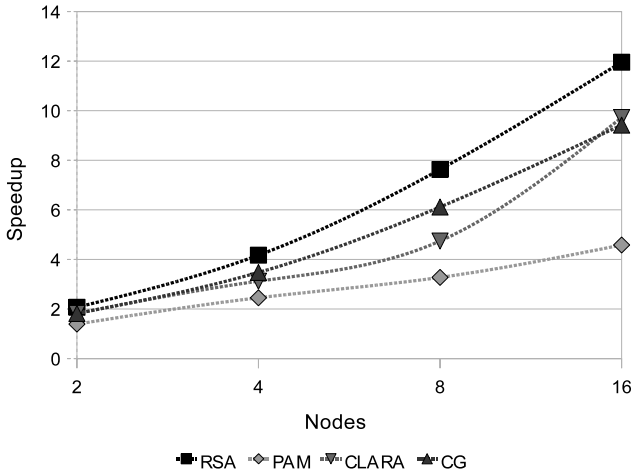

Fig. 5. Achieved speedup comparison of the four algorithms.

MapReduce jobs in a sequence lowers the efficiency of the result because of the job latency. Job latency is the time it takes for the MapReduce framework to schedule, start and finish a MapReduce job, excluding the time spent on doing the actual calculations.

Additionally, input to Hadoop MapReduce jobs is stored on the HDFS, where the data is saved in a distributed fashion. This input needs to be read again every time a MapReduce job is executed, even if a large portion of the input does not change between the job executions, as it is not possible to cache it in the Hadoop MapReduce framework. For algorithms like CG and PAM, where the bulk of the input data stays the same, it means reading the same input data from the file system many times, doing duplicate work at every iteration and lowering the efficiency of the result.

For algorithm classes 3 and 4, where one or more MapReduce jobs are executed at every iteration, job latency and also the inability to cache MapReduce application input can add up to a significant overhead. This means that a significant amount of time is spent on background tasks managed by the MapReduce framework, and less time is spent on performing the actual calculations. This greatly affects the iterative algorithms which have a large number of iterations.

Regardless of the problems encountered, all implemented algorithms were able to achieve speedup from using multiple nodes, as shown in the illustration 5, with RSA having the best and PAM the worst speedup in our tests. However, it is hard to adequately compare the speedup figures between different algorithms, as they strongly depend on the algorithm characteristics, input size, time spent on calculations and the background tasks etc. For iterative MapReduce algorithms, which require significantly more background tasks, achieving an ideal speedup is more difficult. While increasing the problem size, and thus the time spent on the actual calculation, often improves the result, it also increases the time spent on the background tasks, as increasing the input size means that more time must be spent on reading the input data from the file system at every iteration.

6. Twister MapReduce framework

After identifying the problems Hadoop has with each of the algorithm classes, we were also very interested in comparing the results to other MapReduce frameworks to be able to judge which of the framework's problems are inherent to the Hadoop framework itself and which are inherent to the MapReduce model in general and additionally, to see what effect the choice of the MapReduce framework implementation has on the efficiency and scalability of the algorithms in each algorithm class.

Table 5
Run times for the CG implementation in Twister.

Unknowns	500	1000	2000	4000	6000	8000	10 000	20 000
1 node	3.19	3.40	3.00	4.96	7.69	11.27	16.22	56.01
2 nodes	3.33	3.40	2.82	3.99	5.72	6.98	9.51	28.15
4 nodes	3.27	3.29	2.76	3.54	4.03	5.29	6.54	16.33
8 nodes	3.38	3.30	2.81	3.56	3.79	4.76	5.44	14.75
16 nodes	3.40	3.42	2.75	3.50	3.56	4.11	4.86	10.05

Table 6
Run times for the PAM algorithm in Twister.

Objects	10 000	25 000	50 000	75 000	100 000	200 000	300 000
1 node	5.45	20.55	25.00	96.61	204.55	638.56	1888.71
2 nodes	2.93	10.06	22.85	51.19	93.06	359.88	808.96
4 nodes	3.91	7.99	14.63	15.51	91.78	197.15	343.64
8 nodes	4.04	4.93	15.11	31.84	38.13	131.41	355.77
16 nodes	4.25	6.63	11.55	22.26	24.87	85.76	237.43

We chose Twister [19] as the alternative MapReduce framework because it is advertised as an iterative MapReduce framework and thus should provide a very good comparison to Hadoop for algorithm classes 3 and 4, which Hadoop has been shown to have troubles with. The Twister MapReduce framework distinguishes between static data that does not change in the course of the iterations and normal data which may change at every iteration. It also provides long running Map and Reduce tasks which do not have to be terminated between iterations of MapReduce executions as is required in Hadoop. These two are the main characteristics which separate Twister from Hadoop and provide better support for iterative algorithms. Jaliya Ekanayake et al. [20] compared Hadoop MapReduce, Twister and MPI for different data and computing intensive applications. Their results show that Twister can greatly reduce the overhead of iterative MapReduce applications.

We decided to test the Twister MapReduce framework for classes 3 and 4 to see if and how much faster Twister can manage iterative algorithms. We set up a small Twister cluster in the SciCloud. The cluster is kept very similar to the Hadoop cluster for more accurate comparison. It is composed of one master and fifteen slave nodes. Both the master and the slaves act as MapReduce task nodes, resulting in 16 parallel workers where the MapReduce tasks can be executed. Each node is a virtual machine with 2.2 GHz CPU, 500 MB RAM. Twister has no distributed file system, and all input files are simply distributed to the local drives of the nodes.

The algorithms we implemented in Twister are CG and PAM, representing classes 3 and 4. Tables 5 and 6 show the run times for these experiments and Figs. 6 and 7 show the calculated speedup.

Comparing the Twister and Hadoop (Tables 1 and 2) run times for these algorithms clearly shows that Twister is much more efficient for classes 3 and 4. Twister can solve larger problems in less time and for the same size problems it is 50–100 times faster than Hadoop, when running on 16 nodes. The way the algorithm structure is adapted to the MapReduce model stays exactly the same in both Twister and Hadoop, yet Twister is much more efficient in handling background tasks for iterative MapReduce applications. In Hadoop the job lag was around 19–20 s per iteration, while in Twister it is below 3 s regardless of the number of iterations. Twister also stores the bulk of the input to the memory and does not need to read it again from the file system at every iteration. In CG, it means being able to store the whole matrix into the collective memory of the cluster and in PAM it means being able to store all the clustered objects.

However, Twister also has certain limitations for distributed applications. The significant advantage in using Twister comes from its ability to keep static input data in memory across iterations. But it means that this static data must fit into the collective memory of the machines Twister is configured on. For data intensive tasks this may be quite an unreasonable demand. For example, processing 1 TB of data with Twister would require more than 128 machines with 8 GB of memory each just to store the data into the memory, not to mention the memory needed for the rest of the application, the framework itself and the operating system it runs in. Twister also does not have a proper fault tolerance when

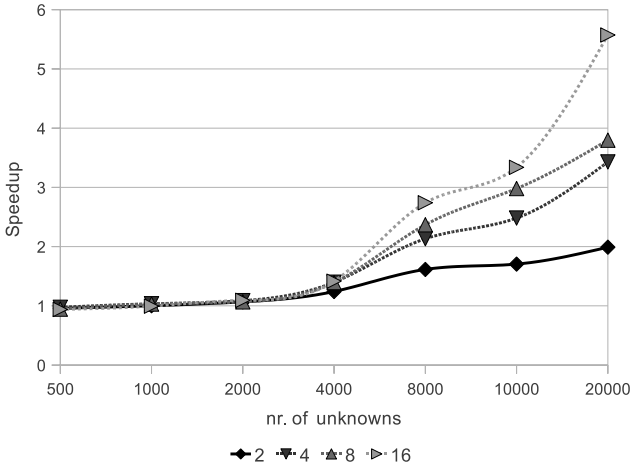

Fig. 6. Parallel speedup for CG in Twister.

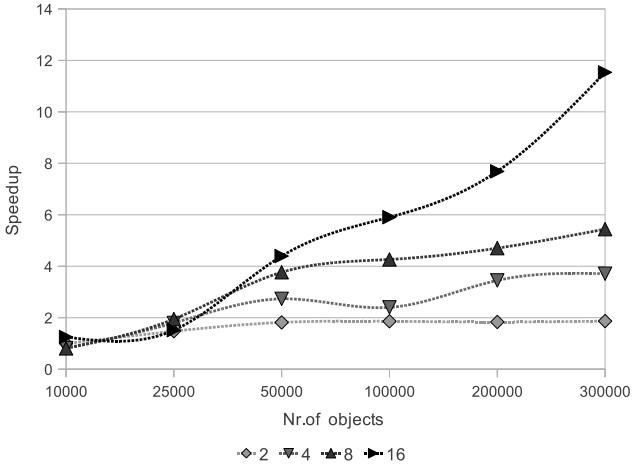

Fig. 7. Parallel speedup for PAM in Twister.

compared to the fault tolerance provided by Hadoop, which can be a very serious problem when running twister on a public cloud where machines are prone to relatively frequent failures.

Comparing Twister and Hadoop for algorithms belonging to the third and fourth class has shown that Twister is much more suitable for these classes. At the same time, Hadoop is more suitable for the first class of algorithms, thanks to the fault tolerance it provides, and also for data intensive algorithms in general, as Twister has problems fitting the data into the memory. For less data intensive algorithms belonging to the second class the results are not so clear. When the number of different MapReduce executions is not large, Hadoop can perform well and given the fault tolerance, it should be considered to be more suitable. But because of the short running tasks in Hadoop, which are terminated each time one MapReduce cycle is over, Hadoop loses its efficiency as the number of MapReduce executions increase and Twister should be preferred instead. Thus, the choice of the framework for the second class of algorithms strongly depends on the number of MapReduce steps needed and on how data intensive the task is.

7. Related work

Apart from Hadoop and Twister there are multiple implementations of distributed computing frameworks based on the MapReduce model. Yingyi Bu et al. presented HaLoop [21], which extends the Hadoop MapReduce framework by supporting iterative MapReduce applications, adding various data caching mechanisms and making the task scheduler loop-aware. They separate themselves from Twister by claiming that HaLoop is more suited for iterative algorithms because using the memory cache and long running MapReduce tasks makes Twister less suitable for commodity hardware and more prone to failures.

Matei Zaharia et al. [22] propose Spark, a framework that supports iterative applications, yet retains the scalability and fault tolerance of MapReduce. Spark focuses on caching the data between different MapReduce-like task executions by introducing resilient distributed datasets (RDDs) that can be explicitly kept in memory across the machines in the cluster. However, Spark does not support group reduction operation and only uses one task to collect the results, which can seriously affect the scalability of algorithms that would benefit from concurrent Reduce tasks, each task processing a different subgroup of the data [22].

The Google solution for the MapReduce problems with iterative graph algorithms is Pregel [23]. Grzegorz Malewicz et al. introduce Pregel as a scalable and fault-tolerant platform for iterative graph algorithms. Compared to previous related work, Pregel is not based on the MapReduce model but rather on the Bulk Synchronous Parallel model [24]. In Pregel the computations consist of supersteps, where user defined methods are invoked on each graph vertex concurrently. Each vertex has a state and is able to receive messages sent to it from the other vertexes in the previous super-step. While the vertex central approach is similar to the MapReduce map operation which is performed on each item locally, the ability to preserve the state of each vertex between the super-steps provides the support for iterative algorithms.

Similarly, Phoenix [25] implements MapReduce for shared-memory systems. Its goal is to support efficient execution on multiple cores without burdening the programmer with concurrency management. Because it is used on shared-memory systems it is less prone to the problems we encountered with iterative algorithms as long as the data can fit into the memory. The idea is interesting, but a shared memory model cannot be considered a solution for the SciCloud project, as we are more interested in using existing university resources and commodity hardware.

Saurabh Sehgal et al. [26] implement the MapReduce model using SAGA (Simple API for Grid Applications) – an API that supports platform independent distributed programming – with the aim to provide interoperability for distributed scientific applications. They argue that the MapReduce model suits well for implementing application interoperability and demonstrate three different levels of interoperability for distributed applications. First, where the application can use different distributed platforms without changes to the application. Second, where the application can seamlessly switch between different programming models. Third, where multiple programming models can be used concurrently to solve a single task. They conclude that the impact of application level interoperability is an important step towards understanding general-purpose programming models.

8. Conclusions and future research directions

Cloud computing, with its promise of virtually infinite resources, seems to suit well in solving resource greedy scientific computing problems. The work presented in the paper studies adapting scientific computing problems to the MapReduce model. The study has formulated 4 different classes for algorithms and has divided scientific algorithms among these classes as follows: embarrassingly parallel algorithms, easily parallelizable algorithms, iterative algorithms and complex iterative algorithms. The paper investigates what affects the parallel efficiency and scalability of algorithms in each of these classes by adapting them to the Hadoop MapReduce framework and analyzing the results.

From this analysis, it can be observed that the Hadoop MapReduce framework has several problems with iterative algorithms, where one or more MapReduce jobs need to be executed at each iteration. For each MapReduce job that is executed, some time is spent on background tasks, regardless of the input size, which can be viewed as MapReduce job latency. If the number of iterations is large, then this latency adds up to a significant overhead in Hadoop and lowers the efficiency of such algorithms. Moreover, the input to a Hadoop MapReduce job is stored on the HDFS. If a Hadoop MapReduce job is executed more than once, it means that the input has to be read again from the HDFS every time, regardless of how much of the input has changed from the previous iterations. For algorithms like CG and PAM, where most of the input does not change between the iterations and the number of iterations is large, this is a serious problem.

The study later tried to implement algorithms from the last two iterative classes to an alternative MapReduce framework called Twister, to be able to judge which problems we encountered are specific to the Hadoop framework and which are inherent from the MapReduce model itself. The results of the experiments show that Twister performs much better for iterative algorithms belonging to classes 3 and 4, showing run times incomparable to Hadoop. However, Twister is not without faults. Because the main advantage of Twister comes from its ability to store input data into memory between iterations, it also requires this data to fit into the collective memory of the cluster in order to be effective. This is unfeasible for tasks that are required to process hundreds of Terabytes of data. Another disadvantage of Twister is its weak fault tolerance compared to Hadoop. Fault tolerance is of significant importance for long iterative tasks running on cloud computing platforms which are generally prone to hardware and network failures.

For these reasons we conclude that the Hadoop MapReduce framework is more suited for data intensive algorithms belonging to the first and the second class, which consist of embarrassingly parallel and easily parallelizable algorithms. However, when a large number of MapReduce jobs must be executed in an algorithm belonging to the second class, then Hadoop will lose

its efficiency and Twister should be considered instead. For third and fourth classes, which consist of iterative and complex iterative algorithms, Twister has proven to be much more efficient than Hadoop. The results show us that MapReduce can be used successfully for solving scientific computing problems as long as the algorithm characteristics are properly considered and a suitable MapReduce framework is chosen based on those characteristics.

Apart from Hadoop and Twister, we are also considering other frameworks for utilizing cloud computing resources to solve scientific computing problems. As such, future work will include study into other iterative MapReduce frameworks like Spark or HaLoop and alternative distributed computing models like Bulk Synchronous Parallel. Apart from evaluating existing cloud computing solutions, we are also interested in designing an original distributed cloud computing framework for scientific computing which would cater for our needs, providing automatic parallelism, fault tolerance and would be suitable for all of the algorithm classes described in this article. Because it is clear that the Hadoop MapReduce framework is very well suited for embarrassingly parallel algorithms, our future work will also include implementing other embarrassingly parallel scientific computing algorithms on the Hadoop MapReduce framework, for example algorithms based on the Monte Carlo method [27,28].

Acknowledgments

The research is supported by the European Social Fund through the Mobilitas program and the European Regional Development Fund through the Estonian Centre of Excellence in Computer Science.

References

[1] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging it platforms: vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems 25 (2009) 599–616.
[2] S.N. Srirama, O. Batrashev, E. Vainikko, Scicloud: scientific computing on the cloud, in: The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2010, p. 579.
[3] S.G. Jeffrey Dean, MapReduce: simplified data processing on large clusters, in: Proc. of the 6th OSDI.
[4] Apache Software Foundation, Hadoop, 2011. http://wiki.apache.org/hadoop/.
[5] S. Ghemawat, H. Gobioff, S.-T. Leung, The google file system, SIGOPS Operating Systems Review 37 (2003) 29–43.
[6] J. Cohen, Graph twiddling in a MapReduce world, Computing in Science and Engineering 11 (2009) 29–41.
[7] C. Bunch, B. Drawert, M. Norman, Mapscale: a cloud environment for scientific computing, Technical Report, University of California, Computer Science Department, 2009.
[8] Google Inc., App engine java overview, 2011. http://code.google.com/appengine/docs/java/overview.html.
[9] Amazon Inc., Amazon elastic compute cloud (Amazon ec2), 2011. http://aws.amazon.com/ec2/.
[10] Eucalyptus Systems Inc., Eucalyptus, 2011. http://www.eucalyptus.com.
[11] S. Srirama, M. Jarke, Mobile hosts in enterprise service integration, International Journal of Web Engineering and Technology (IJWET) 5 (2009) 187–213.
[12] K. Gottschalk, S. Graham, H. Kreger, J. Snell, Introduction to web services architecture, IBM Systems Journal: New Developments in Web Services and E-commerce 41 (2) (2002) 178–198.
[13] S.N. Srirama, M. Jarke, W. Prinz, Mobile web services mediation framework, in: Middleware for Service Oriented Computing (MW4SOC) Workshop @ 8th Int. Middleware Conf. 2007, ACM Press, 2007.
[14] S.N. Srirama, V. Shor, E. Vainikko, M. Jarke, Scalable mobile web services mediation framework, in: Fifth Int. Conf. on Internet and Web Applications and Services, ICIW 2010, IEEE CS, 2010, pp. 315–320.
[15] Apache Software Foundation, Hdfs, 2011. http://hadoop.apache.org/common/docs/current/hdfs_design.html.
[16] J.R. Shewchuk, An introduction to the conjugate gradient method without the agonizing pain, Technical Report, Pittsburgh, PA, USA, 1994.
[17] L. Kaufman, P. Rousseeuw, Finding Groups in Data An Introduction to Cluster Analysis, Wiley Interscience, New York, 1990.
[18] C. Pomerance, A tale of two sieves, Notices of the American Mathematical Society 43 (1996) 1473–1485.
[19] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, G. Fox, Twister: a runtime for iterative MapReduce, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC'10, ACM, New York, NY, USA, 2010, pp. 810–818.
[20] J. Ekanayake, X. Qiu, T. Gunarathne, S. Beason, G. Fox, High performance parallel computing with cloud and cloud technologies, Technical Report, Indiana University, 2009.
[21] Y. Bu, B. Howe, M. Balazinska, M.D. Ernst, HaLoop: efficient iterative data processing on large clusters, in: 36th International Conference on Very Large Data Bases, Singapore.
[22] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: 2nd USENIX Conf. on Hot Topics in Cloud Computing, HotCloud'10, p. 10.
[23] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, 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.
[24] W.F. McColl, Scalability, portability and predictability: the bsp approach to parallel programming, Future Generation Computer Systems 12 (1996) 265–272. Parallel Computing Applications.
[25] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, Evaluating MapReduce for multi-core and multiprocessor systems, in: 13th International Symposium on High Performance Computer Architecture, HPCA 07, IEEE CS, 2007, pp. 13–24.
[26] S. Sehgal, M. Erdelyi, A. Merzky, S. Jha, Understanding application-level interoperability: Scaling-out MapReduce over high-performancee Grids and clouds, Future Generation Computer Systems 27 (2011) 590–599.
[27] C. Robert, G. Casella, Monte Carlo Statistical Methods, Springer Verlag, 2004.
[28] S. Branford, C. Sahin, A. Thandavan, V. Alexandrov, I. Dimov, Monte Carlo methods for matrix computations on the Grid, Future Generation Computer Systems 24 (2008) 605–612.

Satish Narayana Srirama is a senior researcher at Distributed Systems Group, Institute of Computer Science, University of Tartu. He received his Ph.D. in computer science and Masters in Software Systems Engineering from RWTH Aachen University, Germany and Bachelors degree in Computer Science and Systems Engineering from Andhra University, India. His current research focuses at mobile web services, cloud computing, scientific computing and mobile community support.

Pelle Jakovits is a Ph.D. student in the Institute of Computer Science, the University of Tartu. He received his M.S. in Info-technology in the University of Tartu in 2010 on the topic "Reducing scientific computing problems to MapReduce". His current interests in informatics research are parallel computing, Scientific Computing on the Cloud, distributed programming model MapReduce and its alternatives.

Eero Vainikko studied applied mathematics at the University of Tartu and Moscow Institute of Control Problems (1981–1990), he got his Ph.D. in informatics from the University of Bergen, Norway (1997). He is professor of distributed systems at the Institute of Computer Science, University of Tartu. His main research interests include numerical analysis and domain decomposition methods, parallel and distributed computing techniques and environments for solving large engineering and scientific computing problems.