Scientific Programming 19 (2011) 91–105
DOI 10.3233/SPR-2011-0320
IOS Press

# Scalability of parallel scientific applications on the cloud

Satish Narayana Srirama [*], Oleg Batrashev, Pelle Jakovits and Eero Vainikko
*Distributed Systems Group, Institute of Computer Science, University of Tartu, J. Liivi 2, Tartu, Estonia*
*E-mails: {srirama, olegus, jakovits, eero}@ut.ee*

**Abstract.** Cloud computing, with its promise of virtually infinite resources, seems to suit well in solving resource greedy scientific computing problems. To study the effects of moving parallel scientific applications onto the cloud, we deployed several benchmark applications like matrix–vector operations and NAS parallel benchmarks, and DOUG (Domain decomposition On Unstructured Grids) on the cloud. DOUG is an open source software package for parallel iterative solution of very large sparse systems of linear equations. The detailed analysis of DOUG on the cloud showed that parallel applications benefit a lot and scale reasonable on the cloud. We could also observe the limitations of the cloud and its comparison with cluster in terms of performance. However, for efficiently running the scientific applications on the cloud infrastructure, the applications must be reduced to frameworks that can successfully exploit the cloud resources, like the MapReduce framework. Several iterative and embarrassingly parallel algorithms are reduced to the MapReduce model and their performance is measured and analyzed. The analysis showed that Hadoop MapReduce has significant problems with iterative methods, while it suits well for embarrassingly parallel algorithms. Scientific computing often uses iterative methods to solve large problems. Thus, for scientific computing on the cloud, this paper raises the necessity for better frameworks or optimizations for MapReduce.

Keywords: Scientific computing, cloud computing, MapReduce, benchmarking, iterative solvers, parallel programming

## 1. Introduction

Scientific computing is a field of study that applies computer science to solve typical scientific problems. Scientific computing is usually associated with large scale computer modeling and simulation and often requires large amount of computer resources. Cloud computing [3] suits well in solving these scientific computing problems, with its promise of provisioning virtually infinite resources. Cloud computing is a style of computing in which, typically, resources scalable on demand are provided "as a service (aaS)" over the Internet to users who need not have knowledge of, expertise in, or control over the cloud infrastructure that supports them. The provisioning of the cloud services can be at the Infrastructural level (IaaS), Platform level (PaaS) or at the Software level (SaaS). A cloud computing platform dynamically provisions, configures, reconfigures and de-provisions servers as requested. This ensures elasticity of the systems deployed in the cloud. Elasticity of any framework can be defined as its ability to adjust according to the varying loads of requests or requirements it has to support. Cloud computing mainly forwards utility computing model, where consumers pay based on their usage. Servers in the cloud can be physical or virtual machines.

To analyze the cost of science on the clouds, Scientific Computing Cloud [36] (SciCloud) project has been initiated at the University of Tartu. The goal of the project is to study the scope of establishing private clouds at universities, so that the initial experimental costs can be reduced and one can efficiently use the already existing university cluster resources. While there are several public clouds on the market, Google Apps like Google Mail, Docs, Sites, etc., Google App Engine and Amazon EC2 (Amazon Elastic Compute Cloud) are probably most known and widely used. However, to establish a private cloud, several free implementations of the cloud infrastructure like Eucalyptus [26], OpenNebula [27], etc. have been studied, that allow creating private clouds compatible with Amazon EC2. Using the Eucalyptus technology, scientific computing cloud (SciCloud) has been established on our high-performance computing (HPC) clusters.

Initially several customized cloud images are created, that supported some of our research applications

[*]Corresponding author.

in mobile web services and parallel computing domains. This setup helped us in achieving elasticity and load balancing for these applications. Later experiments targeted at deploying DOUG (Domain decomposition On Unstructured Grids) application to the cloud. DOUG is an open source software package for parallel iterative solution of very large sparse systems of linear equations with up to several millions of unknowns. While running these parallel applications on the SciCloud, it was realized that the transmission delays in the cloud environment to be the major problem for adapting HPC problems on the cloud. To prove this and to observe how scientific computing applications scale on the clouds, we experimentally measured performance latencies for several scientific computing tasks running on clusters and cloud nodes.

During this analysis it was also realized that, to be able to run the scientific computing applications on the cloud infrastructure, it is better that the applications be reduced to frameworks that can successfully exploit the cloud resources. For clarity: private clouds and public clouds are most often established on commodity hardware, which are prone to fail at regular intervals. So the applications deployed on them should be aware of the caveat or built with frameworks that can adapt to these failures. In the SciCloud project we are mainly studying at adapting some of the scientific computing problems to the MapReduce [11] framework. MapReduce implementations achieve this fault tolerance by replicating both data and computation. The study observed that MapReduce and some of its well-known implementations like Hadoop suit well for embarrassingly parallel algorithms while posing several problems for the iterative algorithms. This paper explains the study in detail and is organized as follows.

Section 2 briefly introduces the SciCloud project and the applications benefiting from it. Section 3 discusses the concepts of deploying scientific computing applications to the cloud. Section 4 discusses the DOUG system and the details of exporting DOUG to the SciCloud along with detailed analysis. Section 5 explains the study of adapting scientific computing applications to the MapReduce framework along with several algorithms and their analysis. Section 6 describes the related work and Section 7 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 [36] 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. The current project tries to achieve this with more interactive and service oriented models of the cloud computing that fits a larger class of applications. It mainly targets the development of a framework, including models and methods for establishment, proper selection, state management (managing running state and data), 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 at finding new distributed computing algorithms and tries to reduce some of the scientific computing problems to MapReduce algorithm. The SciCloud project thus shall ultimately benefit the cluster, cloud and grid community.

While there are several public clouds on the market, Google Apps (examples include Google Mail, Docs, Sites, Calendar, etc.), Google App Engine [18] (provides elastic platform for Java and Python applications with some limitations) and Amazon EC2 [2] are probably most known and widely used. Elastic Java Virtual Machine on Google App Engine allows developers to concentrate on creating functionality rather than bother about maintenance and system setup. Such sandboxing, however, places some restrictions on the allowed functionality. Amazon EC2 on the other hand allows full control over 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 (VM), which differ in CPU power, memory and disk space. This functionality allows to freely select suitable technologies for any particular task. In case of EC2, price for the service depends on machine size, its uptime, and used bandwidth in and out of the cloud.

There are also free implementations of the cloud infrastructure, e.g., Eucalyptus [15]. Eucalyptus allows creating private clouds compatible with Amazon EC2. Thus, the cloud computing applications can initially be developed at the private clouds and can later be scaled to the public clouds. The setup is of great help for the research and academic communities, as

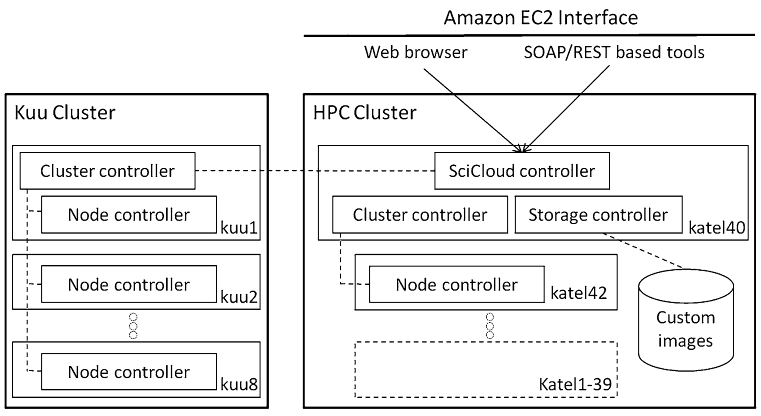


Fig. 1. Architecture of the SciCloud.

Table 1
Cluster and cloud setup used for the experiments

| | Hardware | OS | Software |
|---|---|---|---|
| Cluster (kuu) | Cluster node: AMD Opteron 2.2 GHz with 2 cores, 4 GB memory, 1 MB cache | CentOS 5.4 kernel 2.6.18 | OpenMPI 1.3.2 |
| Cloud (kuu) | On the cluster node: up to 2 cloud instances, each with 1 core and 1 GB memory | Ubuntu 9.04 kernel 2.6.27 | OpenMPI 1.3 |
| Cluster (katel) | Cluster node: Intel Xeon 2.5 GHz with 8 cores, 32 GB memory, 6 MB cache | CentOS 5.4 kernel 2.6.18 | OpenMPI 1.3.2 BLAS (netlib 3.1.1) |
| Cloud (katel) | On the cluster node: up to 8 cloud instances, each with 1 core and 1 GB memory | Ubuntu 9.04 kernel 2.6.27 | OpenMPI 1.3 BLAS (libblas3gf 1.2) |
| Cloud[+atlas] (katel) | *Same as above* | Ubuntu 9.04 kernel 2.6.27 | OpenMPI 1.3 BLAS (libatlas3gf 3.6.0) |
| Hadoop cloud (katel) | On the cluster node: up to 8 cloud instances, each with 1 core and 1.5 GB memory | Ubuntu 9.04 kernel 2.6.27 | hadoop 0.20.2+320, Java 1.6.0 |

the initial expenses of experiments can be reduced by great extent. With this primary goal we initially have set up a SciCloud on a cluster consisting of 8 nodes of SUN FireServer Blade system (Kuu) with 2-core AMD Opteron Processors, using Eucalyptus technology [15]. The architecture of the SciCloud is shown in Fig. 1 and the details of hardware/software are in Table 1. The SciCloud controller exposes and manages the underlying virtualized resources via Amazon EC2 compatible APIs. The Cluster controller manages the execution of VMs running on the nodes. The Node controller takes care of the VM activities, including the execution, inspection and termination. The Storage controller provides mechanism for persistent storage and access control of virtual machine images and user data. We later extended it to 2 more nodes from our high-performance computing cluster (Katel) of double quadcore processor with 32 GB memory per node, and moved the SciCloud controller there. The cluster has 42 of such nodes in total and all the cores are available for the SciCloud usage when the need arises. So SciCloud has 352 cores in total at its disposal. The Kuu and Katel clusters are physically located at different places, and are connected by a 100 Mbps Ethernet connection.

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 [36]. In the mobile web services domain, we scaled our Mobile Enterprise [37] 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 [19], also on the radio link and via resource constrained smart phones. Several applications were developed and demonstrated with the Mobile Host in healthcare systems, collaborative m-learning, social networks and multimedia services domains [37]. We shifted some of the components and load balancers of Mobile Enterprise to the SciCloud and proved that Mobile Web Services Mediation Framework [38] and components are horizontally scalable. More details of the analysis are available at [39]. SciCloud also has several machine images supporting in data mining and bio-informatics domains. Moreover, SciCloud provides several supplementary benefits, which are discussed in the following sections, at appropriate locations.

## 3. Scientific computing on the cloud

Once the SciCloud is established, customized machine images are prepared having support for several scientific computing tools and simulations like Python with NumPy and SciPy, and Scilab tool [35]. Machine images with OpenMPI [16] support are also developed. Moreover, scripts are developed, which prepare the complete setup for MPI (Message Passing Interface) applications, starting from instantiating the specified number of instances of given machine image type, preparing the master node and making the configuration ready to running the specified MPI program till storing the performance metrics of the execution. The scripts are also intuitive enough to consider several known problems like some instances not getting started and not getting IP. The scripts start creating new nodes, if the earlier instances cannot start or encounter any problems, considering timeouts. The failure of instance creation can also be due to some nodes in the cloud failing and thus not being able to join the cloud. Detailed analysis with several benchmark applications like matrix–vector multiplications and NASA Advanced Supercomputing parallel benchmarks (NAS PB) [25] are performed using the setup.

We expected that the transmission delays are the major problem in running scientific computing applications on the SciCloud. To come up with true transmission delays, the experiments were performed on a single cluster. This was achieved by restricting all the cloud instances to be initiated on a single cluster with the help of our scripts. We have run CG (Conjugate Gradient) and EP (Embarrassingly Parallel) problems from NAS PB. CG is an iterative algorithm for solving systems of linear equations. The general idea of CG is to perform an initial inaccurate guess of the solution and then improve the accuracy of the guess at each following iteration using different matrix and vector operations. In EP benchmark, two-dimensional statistics are accumulated from a large number of Gaussian pseudo-random numbers, which are generated according to a particular scheme that is well suited for parallel computation. This problem is typical of many Monte Carlo applications [25]. The former is used to evaluate transmission delays: latency and bandwidth, while the latter is used for testing processor performance. The CG times rise significantly with the number of processors on the cloud (see Fig. 2) starting at 8 instances. The EP times did not show any slowdown on the cloud (see Table 2).

All the experiments were conducted several times (5–10 times) and the mean values were considered for the fully loaded cluster cases (8, 16 nodes). For some of the other cases, we had to eliminate outliers as we had no control over the scheduling mechanism. Rows 1–2 of Table 2 show the execution times for the CG problem of Class B from NAS PB on the cluster and SciCloud, under varying number of instances.

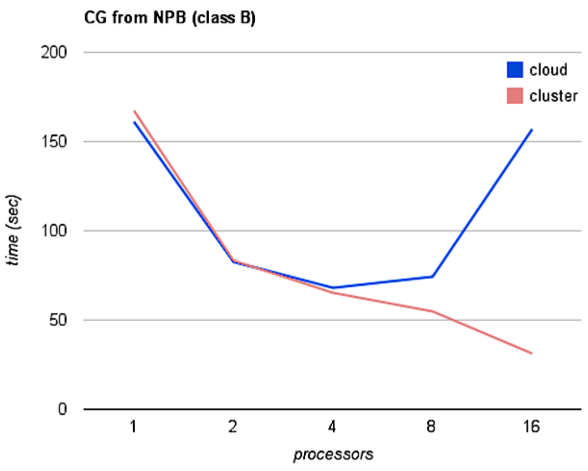


Fig. 2. Comparison of NAS PB CG run times in the cluster and cloud cases. (Colors are visible in the online version of the article; http://dx. doi.org/10.3233/SPR-2011-0320.)

Table 2

Execution times (s) of CG and EP from NAS PB

| Processors | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| CG on cluster (katel) | 167.5 | 83 | 65 | 55 | 31 |
| CG on cloud (katel) | 161.1 | 82.4 | 68.1 | 74 | 156.7 |
| EP on cluster (kuu) | 143.4 | 86.1 | 42.2 | 20.3 | |
| EP on cloud (kuu) | 130.5 | 65.1 | 33.6 | 16.9 | |

The configurations of the instances are described in Table 1. Rows 3–4 show the execution times for the EP problem from NAS PB. The numbers show that EP scales well but CG does not scale on the cloud after 8 instances. To uncover the reasons for such high run times the experiments were run with MPE Profiler [7] that shows MPI function calls and hence allows to distinguish calculations from communication.

The NAS CG problem of Class B runs CG algorithm multiple times and each algorithm makes 75 iterations. Each iteration consists of calculation and communication part, so it is Bulk Synchronous Parallel model. The respective times of one typical CG iteration for 8 MPI processes are given in Table 3. The times of calculation part in one CG iteration are 26 and 24 ms for cluster and cloud, respectively. The times of communication are very different: 2.54 and 13 ms. This makes CG time for 8 processes on the cloud about 50% slower, what we have observed in Table 2.

With MPE profiler we can observe the communication pattern of a CG iteration on the cluster and cloud. The times spent in `MPI_Send` (darker) and `MPI_Wait` (lighter) are shown on Fig. 3. Each process sends a fraction of its vector values to 3 other processes during sparse matrix–vector multiplication operation in each CG iteration. Distributed dot product is performed twice, what can be clearly seen from the iteration for cluster on Fig. 3(a). The communication delays for the cloud are almost one order longer and less predictable, which leaves some processes waiting in `MPI_Wait` call (see Fig. 3(b)).

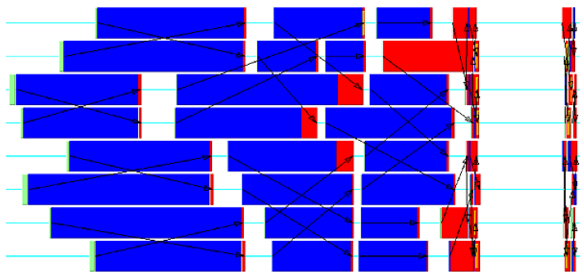
Table 3
Calculation and communication times (ms) of one CG iteration

| | Calculation | Communication |
|---|---|---|
| Cluster | 26 | 2.54 |
| Cloud | 24 | 13 |

Similar timestamps for the case of 16 cloud instances are much higher and thus the abnormal behavior of the CG in Fig. 2. We predict the reason for such high transmission latency in the cloud case is due to the virtualization technology. Virtualization techniques are the basis of the cloud computing. Virtualization technologies partition hardware and thus provide flexible and scalable computing platforms. SciCloud is built on top of XEN virtualization technology [4]. Moreover, we also have observed that MPI applications are very sensitive to the number of VMs per bare-hardware node. Also the latencies and processing capabilities are much worse with the cloud core assignment strategies, such as multiple VMs per core. This leaves a lot of research scope at the virtualization technology to address these problems.

The analysis had given us a chance to have a clear look at the comparison of running the scientific and mathematical problems in a cluster and on the cloud. However, one should not forget here the actual benefit of the cloud with its promise of infinite resources. For example, if the problem defined here needs resources beyond one possesses in their clusters, cloud becomes the only solution. SciCloud being compatible with Amazon EC2, the load can be seamlessly shifted to the public cloud nodes when the need arises. Moreover, Amazon EC2 recently has initiated support for HPC with their Cluster Compute instances [1], which promise much lesser latency between instances.

Apart from helping in handling dynamic loads of applications, SciCloud also provides several supplementary benefits. Generally, clouds, especially IaaS providers, leverage complete access to the instances that are being initiated. This feature comes in handy in several situations. For example, from the earlier analysis, one can realize that to improve the efficiency of cloud application, several optimizations can


(a)

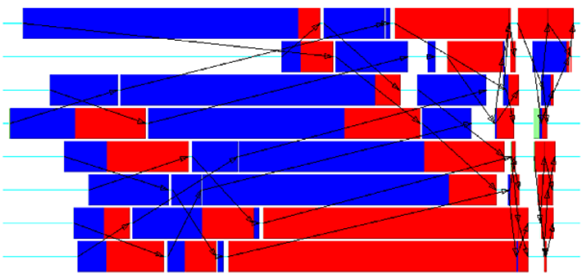

(b)

Fig. 3. Communication pattern of a CG iteration with 8 nodes on the cluster and cloud. The time intervals of the iterations are shown under the respective profiling diagrams. (a) Cluster: 2.54 ms; (b) cloud: 13 ms. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0320.)

Table 4
Level 1 (vvm), 2 (mvm), 3 (mmm) problem sizes

| | Runs (times) | Data size | Operation count |
|---|---|---|---|
| vv multiplication | 1000 | 100,000 size vectors | $10^8$ multiply |
| mv multiplication | 1000 | $1000 \times 1000$ matrix, 1000 size vector | $10^9$ multiply-add |
| mm multiplication | 2 | $1000 \times 1000$ shape matrices | $2 \times 10^9$ multiply-add |

Table 5
Testing Level 1 (vvm), 2 (mvm), 3 (mmm) BLAS routines

| | vvm (s) | mvm (s) | mmm (s) |
|---|---|---|---|
| Cluster (katel) | 0.159 | 2.603 | 14.319 |
| Cloud (katel) | 0.157 | 2.589 | 5.007 |
| Cloud[+atlas] (katel) | 0.27 | 2.044 | 0.995 |

be adopted locally for libraries like BLAS (Basic Linear Algebra Subprograms), UMFPACK (Unsymmetric Multifrontal sparse LU factorization Package), etc. Table 4 shows problem sizes and Table 5 shows the execution times of NumPy program with an optimization on the cloud instances. Cloud instances had newer Linux distribution with more recent packages, which gave the win of approximately 3 times for matrix–matrix multiplication (mmm), and the package that links NumPy and a BLAS implementation, ATLAS (Automatically Tuned Linear Algebra Software), which increased the speed by further 5 times. The other results of the Table 5 are for matrix–vector multiplication (mvm) and vector–vector multiplication (vvm).

Having optimized BLAS implementation on a cluster is compulsory, but optimizations of other libraries on the cluster is a cumbersome process, as they must be installed on the cluster by the administrator. Whereas in the cloud case, the procedure is comparatively simple, and one just has to prepare the proper SciCloud machine image with the software and optimizations. During the experiments, one can run as many instances of his choice or as the application demands, from these machine images.

## 4. Parallel scientific applications on the SciCloud

Once the preliminary analysis with applications on the SciCloud was performed, we tried to use the setup in more serious scientific computing problems, in particular DOUG. The experiments outlined in the previous chapter show that applications with small calculation–communication ratio, like CG behave worse on the cloud than on cluster. In the Fig. 2 it can be observed that CG problem does not scale to 16 cloud instances. Fortunately, pure CG is never used in real life applications, because it requires too many iterations to converge. There are algorithms built on top of CG, called Preconditioned CG, which have much better calculation–communication ratio. DOUG is one such system, and this section presents the analysis of moving DOUG to the SciCloud.

### *4.1. DOUG (Domain decomposition On Unstructured Grids)*

DOUG is a software package for parallel solution of large sparse systems of linear equations typically arising from discretizations of partial differential equations to be solved in 2D or 3D regions which may consist of materials with high variations in physical properties. DOUG is developed at the University of Tartu, Estonia and the University of Bath, England since 1997. It was first implemented in FORTRAN 77 but has been completely rewritten in Fortran 95. To achieve good parallel performance, DOUG uses automatic load-balancing and parallelization being implemented through MPI and overlapping communication and calculations through non-blocking communication whenever it is applicable.

Basically, DOUG uses an iterative Krylov subspace method to solve linear systems in a form

$$Ax = b,$$

where the matrix $A$ is sparse and the dimension of $A$ is very large. Due to large dimensions of $A$ the convergence rate is by far too slow. For that reason, a preconditioner is used. A good preconditioner transforms the original linear system into one with the same solution, but with the transformed linear system the iterative solver needs much smaller number of iterations [32]. DOUG implements the Preconditioned Conjugate Gradient (PCG), the Stabilized Bi-Conjugate Gradient (BiCGstab), the minimal residual (MINRES) and the 2-layered Flexible Preconditioned Generalized Minimum Residual method (FPGMRES) with left or right preconditioning.

The general algorithm used for creating the subproblems that can be assigned to separate CPUs is called domain decomposition. The idea of domain decomposition is to decompose the problem into $n$ sub-problems, usually with some overlap, each of which will be solved on one CPU. In a way, it is similar to a divide and conquer scheme but with domain decomposition there is communication on the borders of the sub-problems (overlaps) involved. Usually, communication is measured to be more costly than CPU time and therefore the decomposition algorithm tries to minimize the cross-sections between neighboring sub-domains.

In domain decomposition, instead of solving the global system of equations, smaller problems are solved on each sub-domain, solutions of which are combined together to form an approximation to the original problem. The common practice is to use domain decomposition as a preconditioning step for Krylov subspace methods such as the conjugate gradient method or the method of generalized minimum residual [8]. DOUG employs 2-level preconditioning in which a coarse matrix is used which approximates the global matrix on a suitable chosen coarser scale. This reduces the total work of a preconditioned Krylov method (like PCG) to almost a constant number of iterations independent of the matrix $A$ size.

Recently, the development and research has been focused around aggregation based domain decomposition methods. Major progress has been made in determining an upper bound for the condition number of the preconditioner in case of highly variable coefficients. This allows a better estimate of the error and thus enables the solver to finish in less iterations. This approach has been implemented in DOUG and it has been shown experimentally that it is of superior speed to comparable methods [33,34].

## 4.2. DOUG on the SciCloud

To run DOUG on the SciCloud, machine images with DOUG software have been prepared and the scripts that helped us in preparing the MPI setup are extended for the DOUG. Test problem has been generated from the Poisson's equation $\nabla^2\phi = f$ on unit square with Dirichlet boundary conditions. This equation arises in many situations like the calculation of heat distribution or the calculation of strains and stresses in structural mechanics

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = f(x,y),$$
$$\phi(0,y) = \phi(x,0) = \phi(x,1) = \phi(1,y) = 0.$$

The approximate solution for $\phi(x,y)$ is found by Finite Difference method on 2D grid, which approximates partial derivatives in each point $(x', y')$ of the grid by their differences

$$\begin{aligned}\left.\frac{\partial^2\phi}{\partial x^2}\right|_{x',y'} &\approx \big(\phi(x'-\Delta_x, y') - 2\phi(x',y') \\ &\quad + \phi(x'+\Delta_x, y')\big)/\Delta_x^2, \\ \left.\frac{\partial^2\phi}{\partial y^2}\right|_{x',y'} &\approx \big(\phi(x', y'-\Delta_y) - 2\phi(x',y') \\ &\quad + \phi(x', y'+\Delta_y)\big)/\Delta_y^2.\end{aligned}$$

Applying Finite Differences to a grid point gives one linear equation, which reflects the Poisson's equation for the point. Unknowns of the equation stand for the approximated values of the original function $z_i \approx \phi(x', y')$ in the corresponding points. All equations comprise a system of linear equations with the set of unknowns $z_i, 1 \leqslant i \leqslant N$, where $N$ is the size of the grid except grid boundaries.

The size of the grid considered for the analysis is $1536 \times 1536$, which gives rise to the system of linear equations with about 2.5 millions of unknowns. The matrix of the system contains more than 10 million non-zero elements and its data file is 180 MB in size. DOUG master node first reads data from disk and distributes it to slave nodes, which together with master node use Preconditioned CG algorithm to solve the system. Running DOUG program with the given data takes more than 1 GB of memory when running on less than 3 nodes, so it was not possible to run it on the cloud with 1 GB of available memory per VM. The cloud has been tested with 4–16 instances running on two 8-core nodes in the SciCloud.

As has been mentioned DOUG runs one iterative solver as its core and a preconditioner is applied at each iteration. The preconditioner may also require some interprocess communication but usually has much better computation–communication ratio than pure CG. Thus, we expected transmission delays in the cloud to be less critical for DOUG than it was for CG. The results are shown on Fig. 4.

The graph shows that the cloud does not introduce extra communication overhead with 16 instances and overall performance is better than on cluster, probably because of the newer software. However, the overall scaling is not very good for both cluster and cloud cases. This may be due to the optimal DOUG requirement to have 2 cores per processor. Our future research

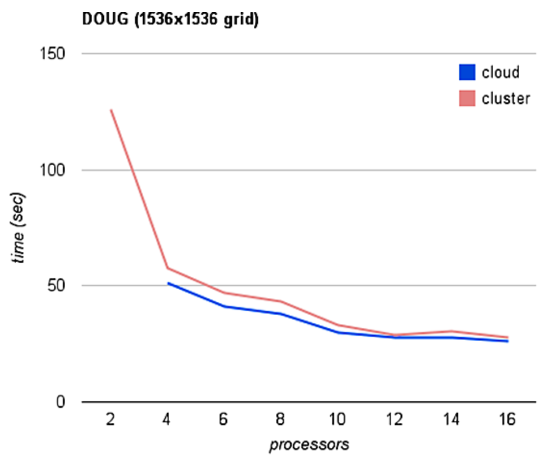


Fig. 4. DOUG run times in the cluster and cloud cases. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0320.)

in this domain will address this issue in detail and we are interested in repeating the tests with 2 cores per instance.

From this analysis it can be observed that parallel scientific applications scale reasonable on the cloud. However, our experience of moving DOUG to the SciCloud reveals some problems with the cloud. First, instances in the cloud are run on random nodes in the network. This may cause problems of establishing connection from one instance to another. We circumvented the problem with our scripts, which made them initiate on a single cluster or on a single node of a cluster. Public clouds, especially Amazon EC2, also provide such support for HPC, with their Cluster Compute instances, which promise much lesser latency between instances [1]. Second problem is that MPI configuration needs to be created dynamically during run time, which is done by our scripts. However, for others to move their MPI applications to the cloud, additional tools are helpful.

Apart from being able to scale, with the deploying of DOUG onto the SciCloud, we also get several intangible benefits. For example, our experience with GRID and ordinary private clusters revealed that installing DOUG is not a trivial task, as it needs a number of specific libraries and proper configuration of MPI development environment. The administrator of a cluster has usually no knowledge about these details, while it is straightforward for the application developers and the advanced users of the application. The software configuration and administrative tasks have been one of the major problems with GRID establishment, which we observed to lessen by using customized images and flexible models of the cloud computing. Our solution is to let administrator prepare general machine image with compilers, MPI environment and common scientific libraries installed and allow others to extend this image to their needs. The resulting image can again be used by everyone.

## 5. Adapting scientific computing problems to the clouds using MapReduce

From the earlier subsections it can be deduced that Parallel Scientific Applications benefit a lot from the cloud computing. However, cloud computing comes with several typical problems. Most often cloud infrastructure is built on commodity computers, which are prone to fail regularly. So the applications that are run on cloud computing platforms should be aware of this fact and the applications should be foolproof. Thus, to be able to gain maximum efficiency from the cloud infrastructure, the scientific computing applications must be reduced to frameworks that can successfully exploit the cloud resources. In the SciCloud project we are mainly studying at adapting some of the scientific computing problems to the MapReduce [11] framework.

### *5.1. SciCloud Hadoop 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 [20] MapReduce. Hadoop is a Java software framework inspired by Google's MapReduce and Google File System [17] (GFS). 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. 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.

MapReduce applications get a list of key–value pairs as an input and consist of two main methods, Map and Reduce. Map method processes each key–value pair in the input list separately, and outputs one or more key–value pairs as a result

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

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

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

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 Hadoop cluster.

### 5.2. *Scientific computing problems reduced to MapReduce framework*

The structure of a MapReduce application is very strict. It is not trivial to reduce complex algorithms to MapReduce model and there is no guarantee that the resulting algorithms are effective. Previous work has shown that MapReduce is well suited for simple, often embarrassingly parallel problems. Google shows in their paper [11] that they use MapReduce for wide variety of problems like large-scale indexing, graph computations, machine learning and extracting specific data from a huge set of indexed web pages. Cohen [10] 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 [6]. It performed well for simple problems like Marsaglia polar method for generating random variables and integer sort. However, MapReduce had significant problems with more complex algorithms, like CG, fast Fourier transform, and block tridiagonal linear system solver. Moreover, most of these problems use iterative methods to solve them.

To be able to get a more accurate overview of the problems MapReduce has with iterative algorithms we decided to reduce several iterative and also, as a comparison, several embarrassingly parallel scientific algorithms to the Hadoop MapReduce framework and compare the results. The algorithms we chose are following:

- Conjugate Gradient;
- $k$-medoid clustering (PAM);
- Integer factorization;
- Monte Carlo integration.

As already mentioned, CG is an iterative algorithm for solving systems of linear equations. The general idea of CG is to perform an initial inaccurate guess of the solution and then improve the accuracy of the guess at each following iteration using different matrix and vector operations. 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 iteration are reduced to the MapReduce model. These operations are matrix–vector multiplication, dot product, two vector addition and vector and scalar multiplication. As a result, multiple MapReduce jobs are executed at every iteration.

Partitioning Around Medoids [22] (PAM) is a iterative $k$-medoid clustering algorithm. The general idea of a $k$-medoid clustering is that each cluster is represented by it's 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. 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 the cluster.
  - Input: (cluster id, (list of all objects in the cluster)).
  - Output: (cluster id, new medoid).

Map method recalculates to which cluster each object belongs to, and 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.

For the integer factorization we chose the most basic, the trial division method. This method is not used in practise because it is relatively slow and there exist much faster methods like general number field sieve [28]. But we chose this method purely to illustrate adapting an embarrassingly parallel problem to MapReduce, as a comparison to the iterative algorithms.

To factor a number using trial division, all possible factors of the number are checked to see if they divide the number evenly and thus are its factors. This can be adopted to the MapReduce model easily, by dividing all possible factors into multiple subgroups and checking each of the subgroups 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 $\sqrt{number}$ into $n$ smaller ranges, and outputs each of them. Dividing the work between $n$ nodes.
  - Input: (key, (function, experiments)).
  - Output: (id, (start, end, number)) [one output for each range, $n$ total].
- 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).

Monte Carlo integration uses the Monte Carlo method [31] to calculate the integral of a function in a given area. Monte Carlo methods are a class of algorithms that utilize repeated random sampling for calculating the approximation of certain computational problems. The error of such approximation is reduced by increasing the size of the random sampling.

To find the integral of a function in a specific area we can find the average value of the function in this area and multiply it with the volume of the area. The result is the approximation of the integral. Monte Carlo method is used to find the average function value in the give area, by generating a number of random inputs for the function and calculating the average of the resulting function outputs. To improve the accuracy of the result we are using a quasi-random sequence [29] as an input instead of randomly generated numbers, because the quasi-random sequence provides more uniformly distributed samples. Similarly to the integer factorization example, this algorithm can be expressed as a single MapReduce job, which is outlined as follows:

- Map:
  - Gets a function to be integrated and the number of experiments as an input. Divides the number of experiments into $n$ (number of nodes) smaller experiments and outputs each of them. Dividing the work between $n$ nodes.
  - Input: (key, (function, experiments)).
  - Output: (key_i, (function, experiments/$n$)) [one output for each node, $n$ total].
- Reduce:
  - Gets a function to be integrated and the number of experiments as an input. It then finds the function value with random argument for each experiment and sums the values. Reducer outputs the calculated sum.
  - Input: (key, (function, experiments)).
  - Output: (key, sum).

This MapReduce job outputs the sum total of all function values. The algorithm divides this value with the number of experiments to get the average value of the function and multiplies it with the selected area volume to find the value of the integral.

For the testing setup we created a small 12 node Hadoop cluster, using the Hadoop cloud image, and recorded the run times for each of the algorithms in 1, 2, 4, 8 and 12 node Hadoop MapReduce configurations. For each of these configurations we calculated

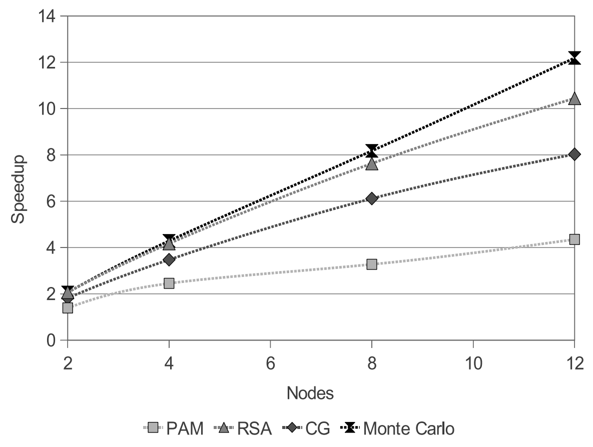


Fig. 5. Speedup comparison of the algorithms on the SciCloud.

the parallel speedup for each of the algorithms to measure how well they scale. Parallel speedup measures how many times the parallel execution is faster than running the algorithm on one machine. 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. The calculated speedup for each of the algorithms is shown on the Fig. 5.

From the calculated speedup numbers, we can see that iterative algorithms CG and $k$-medoid clustering are not able to achieve the ideal parallel speedup, equal to the number of nodes, while non-iterative algorithms integer factorization and Monte Carlo integration are able to achieve ideal speedup in a 8 node set up. Integer factorization algorithm was not able to attain the ideal speedup in the 12 node setup, indicating that the problem size was not large enough and the time spent on background tasks still played a large role in the measured run times when executing the algorithm on 12 nodes.

These experiments show us that Monte Carlo and other embarrassingly parallel algorithms, which do not require iterating over a number of MapReduce jobs, can be very efficient and scalable. But complex iterative algorithms, that require the execution of one or more MapReduce jobs at every iteration, have serious problems, as it means that many smaller MapReduce jobs must be executed in sequence. For each of the MapReduce job executions, it takes time for the framework to schedule, run and clean them up, adding up to a significant overhead if the number of iterations is large. Also, each time a new job is started, the input data must be read again from the file system. For algorithms like CG and $k$-medoid clustering, where large part of the input data does not change between the iterations, this adds a significant extra work at every iteration. This is different form the job execution lag as this overhead is relative to the size of the input, meaning increasing the problem size also increases the overhead. These two are the main problems that limit the efficiency and scalability of complex iterative algorithms adapted to the MapReduce framework.

As a result we have to conclude that current Hadoop MapReduce framework lacks the support for scientific and complex iterative algorithms in general. This is a regrettable result for us as scientific computing very often uses iterative methods to solve scientific problems. Thus, for scientific problems Hadoop MapReduce must be optimised or extended to support iterative algorithms, or other cloud computing frameworks, which do support iterative algorithms, must be used instead.

The main aspects that we feel currently lack from the Hadoop MapReduce framework are support for longer running MapReduce jobs and allowing to cache input data between MapReduce jobs. Longer running MapReduce jobs, which are not automatically terminated and can be reused at every iteration, would allow to save time by minimizing the job execution latency. Being able to cache input data, which does not change between different MapReduce job executions, would allow to save time from reading the same data again from the file system at every iteration.

## 6. Related work

While this paper addressed deploying scientific computing application to the cloud, several other studies are also focused at similar goals. Deelman et al. studied the cost of doing science on the cloud with one parallel computing application, the Montage example. They mainly focused at knowing the costs for executing certain types of scientific computing applications on the Amazon EC2 [12]. Trying to establish private cloud infrastructures for using in scientific applications is getting popular with the emergence of open source cloud software like Eucalyptus, Nimbus etc. Keahey et al. first established a science cloud testbed based on the Nimbus CloudKit [23]. Similarly, Wang et al. in CUMULUS project, focused at building a scientific cloud. The project merged existing grid infrastructures with the cloud technologies by building a frontend service that unifies OpenNebula and GLOBUS [40]. As far as public cloud providers, Amazon EC2 has initiated support recently for HPC and scientific computing problems with their Cluster Compute instances.

Reducing scientific computing applications to MapReduce was studied by Bunch et al. [6]. However, they have not analyzed the reasons for the poor performance in detail. Moreover, for iterative problems like CG no results were published. We focused more at iterative problems and have studied them extensively. After studying the utilization of MapReduce for Scientific Computing and encountering problems with iterative algorithms, we were interested in finding related works which have encountered similar problems or propose solutions for them. One such study is Twister [13] MapReduce framework. Twister is advertising itself as an iterative MapReduce framework which allows creating MapReduce jobs consisting of several iterations of Map and Reduce tasks. It also distinguishes between static data that does not change in the course of the iterations, and normal data which may change during each iteration. Ekanayake et al. [14] compared Hadoop MapReduce, Twister and MPI for different data and computing intensive applications. Their results show that enchanted MapReduce runtime Twister greatly reduces the overhead of iterative MapReduce applications. We are interested in the Twister developments and our future research addresses implementing the algorithms outlined in this paper also in the Twister MapReduce framework to compare the results.

Bu et al. presented HaLoop [5] as a solution for the lack of built-in support in MapReduce for iterative algorithms. HaLoop is a modified Hadoop MapReduce framework designed for iterative algorithms. It extends 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 long running Twister MapReduce tasks and memory cache make it less suitable for commodity hardware and Twister is more prone to failures.

Zaharia et al. [41] also found that MapReduce is not suitable for many applications that need to reuse a working set of input data across parallel operations. They 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. They also claim Spark is more suited for iterative algorithms than Twister because it does not currently implement fault tolerance. At the same time, 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 [41].

Google solution for the MapReduce model problems with iterative graph problems is Pregel [24]. 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 Valiant's Bulk Synchronous Parallel model [9]. In Pregel the computations consist of super-steps, 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.

Phoenix [30] 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. Hoefler et al. [21] have studied using MPI to implement the MapReduce parallel computational model. While their results show that MPI would need additional features to fully support the MapReduce model, we find their work very appealing.

## 7. 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. 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. SciCloud is being used in solving computationally intensive scientific, mathematical, and academic problems. To study the effects of moving the parallel scientific applications onto the cloud, we deployed several benchmark applications like matrix–vector multiplications and NAS parallel benchmarks, and DOUG on SciCloud. The detailed analysis of DOUG on the cloud showed that parallel applications scale reasonable on the cloud. From this analysis, it was also observed that the transmission delays to be the major problem in running scientific computing applications on the cloud. The study blames the transmission delays to be from the virtualization technology. Moreover, virtualization technology splits the physical CPU cores to multiple virtual cores that made the performance comparison of the cluster and cloud a tricky process. However, cloud computing offers elasticity and several other intangible benefits, which themselves make deploying parallel scientific computing applications to the cloud beneficial.

Still, for efficiently running the several types of scientific applications on the cloud infrastructure, the applications must be reduced to frameworks that can successfully exploit the cloud resources, like the MapReduce framework. This work studied adapting several embarrassingly parallel and iterative scientific computing problems to Hadoop MapReduce framework. The study observed that Hadoop MapReduce framework has 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 of the 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 and lowers the efficiency of such algorithms. Moreover, the input to a Hadoop MapReduce job is stored on the HDFS, where the data is distributed to the local hard drives of the machines in the Hadoop cluster. 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.

For these reasons, the paper concludes that Hadoop MapReduce framework is more suited for embarrassingly parallel algorithms, where the algorithm can be divided into independent concurrent tasks and little or no communication is required between them. Such algorithms can often be reduced into a single MapReduce job, like the factoring integers. While there is still job lag when executing a single MapReduce job, the effect is minimal. Compared to algorithms with multiple MapReduce job iterations, less time is spent on background tasks and more time is spent on the actual computations, resulting in a greater overall efficiency. In the context of SciCloud project this is not a very attractive result. Scientific computing often uses iterative methods and our results show that Hadoop MapReduce has significant problems with them.

Thus, for scientific computing on the cloud, this paper raises the necessity for better frameworks or optimizations for MapReduce and provides a lot of scope for future research. As such, the future work will include research into optimizing Hadoop MapReduce framework for iterative algorithms, study of other MapReduce frameworks which have better support for iterative MapReduce applications, like Twister, Spark and HaLoop, and adapting the algorithms used in this paper to each of these frameworks to compare the results. Our future work will also include implementing other embarrassingly parallel scientific computing methods on Hadoop MapReduce framework, either directly or by reducing them to the Monte Carlo method. Apart from this, we are also interested in deploying several other scientific computing applications from different domains like chemistry and bioinformatics. Research groups from these domains are showing lot of interest in our SciCloud developments and are eager in using the SciCloud resources for their research. Our final goal is to come up with a cloud computing framework that fully supports iterative scientific applications and to use the framework to build a solution base for relevant scientific computing problems.

## Acknowledgements

The research is supported by the European Social Fund through Mobilitas program, the European Regional Development Fund through the Estonian Centre of Excellence in Computer Science and Estonian Science Foundation, grant 7566.

## References

[1] Amazon Inc., High performance computing using Amazon EC2, available at: http://aws.amazon.com/hpc-applications/.

[2] Amazon Inc., Amazon Elastic Compute Cloud (Amazon EC2), available at: http://aws.amazon.com/ec2/.

[3] M. Armbrust et al., Above the clouds, a Berkeley view of cloud computing, Technical report, University of California, 2009.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield, Xen and the art of virtualization, in: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, USA, ACM Press, 2003, pp. 164–177.

[5] 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, VLDB Endowment, Berkeley, CA, USA, 2010.

[6] C. Bunch, B. Drawert and M. Norman, MapScale: a cloud environment for scientific computing, Technical report, Computer Science Department, University of California, 2009.

[7] A. Chan, Performance visualization for parallel programs, available at: http://www.mcs.anl.gov/research/projects/perfvis/.

[8] T.F. Chan and T.P. Mathew, Domain decomposition algorithms, *Acta Numerica* **3**(-1) (1994), 61–143.

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

[10] J. Cohen, Graph twiddling in a MapReduce world, *Computing in Science and Engineering* **11**(4) (2009), 29–41.

[11] J. Dean and S. Ghemawat, MapReduce: Simplified data processing on large clusters, in: *Proc. of the 6th OSDI*, San Francisco, CA, USA, December 2004, USENIX Association, Berkeley, CA, USA.

[12] E. Deelman, G. Singh, M. Livny, B. Berriman and J. Good, The cost of doing science on the cloud: the montage example, in: *International Conference for High Performance Computing, Networking, Storage and Analysis, 2008, SC 2008*, Austin, TX, USA, IEEE CS Press, 2009, pp. 1–12.

[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 (HPDC'10)*, Chicago, IL, USA, ACM Press, 2010, pp. 810–818.

[14] J. Ekanayake, X. Qiu, T. Gunarathne, S. Beason and G. Fox, High performance parallel computing with cloud and cloud technologies, 2009, available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.148.6044.

[15] Eucalyptus Systems Inc., Eucalyptus, available at: http://www.eucalyptus.com.

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

[17] S. Ghemawat, H. Gobioff and S.-T. Leung, The Google file system, *SIGOPS Oper. Syst. Rev.* **37**(5) (2003), 29–43.

[18] Google Inc., App engine java overview, available at: http://code.google.com/appengine/docs/java/overview.html.

[19] K. Gottschalk, S. Graham, H. Kreger and J. Snell, Introduction to web services architecture, *IBM Systems Journal: New Developments in Web Services and E-commerce* **41**(2) (2002), 178–198.

[20] Hadoop, available at: http://wiki.apache.org/hadoop/.

[21] T. Hoefler, A. Lumsdaine and J. Dongarra, Towards efficient MapReduce using MPI, in: *PVM/MPI*, M. Ropo, J. Westerholm and J. Dongarra, eds, Lecture Notes in Computer Science, Vol. 5759, Springer-Verlag, 2009, pp. 240–249.

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

[23] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman and M. Tsugawa, Science clouds: Early experiences in cloud computing for scientific applications, in: *Cloud Computing and Applications 2008*, Chicago, IL, USA, 2008.

[24] 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, SIGMOD'10*, New York, NY, USA, 2010, ACM Press, pp. 135–146.

[25] NASA, NAS Parallel Benchmarks, available at: http://www.nas.nasa.gov/resources/software/npb.html.

[26] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff and D. Zagorodnov, The eucalyptus open-source cloud-computing system, in: *9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'09)*, Washington, DC, USA, 2009, IEEE CS Press, pp. 124–131.

[27] OpenNebula.org, OpenNebula: the open source toolkit for cloud computing, available at: http://www.opennebula.org/.

[28] C. Pomerance, A tale of two sieves, *Notices Amer. Math. Soc.* **43** (1996), 1473–1485.

[29] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, *Quasi- (that is, Sub-) Random Sequences*, 2nd edn, Cambridge Univ. Press, New York, NY, USA, 1993.

[30] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski and C. Kozyrakis, Evaluating MapReduce for multi-core and multiprocessor systems, in: *13th International Symposium on High Performance Computer Architecture (HPCA 07)*, Phoenix, AZ, USA, IEEE CS Press, 2007, pp. 13–24.

[31] C. Robert and G. Casella, *Monte Carlo Statistical Methods*, Springer-Verlag, New York, NY, USA, 2004.

[32] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 1st edn, PWS, Boston, MA, USA, 1996.

[33] R. Scheichl and E. Vainikko, Additive Schwarz and aggregation-based coarsening for elliptic problems with highly variable coefficients, Preprint 9/06, Bath Institute For Complex Systems, 2006.

[34] R. Scheichl and E. Vainikko, Robust aggregation-based coarsening for additive Schwarz in the case of highly variable coefficients, in: *Proceedings of the European Conference on Computational Fluid Dynamics, ECCOMAS CFD 2006*, P. Wesseling, E. O'Nate and J. Periaux, eds, TU Delft, 2006.

[35] Scilab.org, Scilab: the free platform for numerical computation, available at: http://www.scilab.org/.

[36] S.N. Srirama, O. Batrashev and E. Vainikko, SciCloud: Scientific Computing on the Cloud, in: *The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010)*, Melbourne, Australia, IEEE CS Press, 2010, p. 579.

[37] S.N. Srirama and M. Jarke, Mobile hosts in enterprise service integration, *International Journal of Web Engineering and Technology* **5**(2) (2009), 187–213.

[38] S.N. Srirama, M. Jarke and W. Prinz, Mobile web services mediation framework, in: *Middleware for Service Oriented Computing (MW4SOC) Workshop @ 8th Int. Middleware Conf. 2007*, Newport Beach, CA, USA, ACM Press, 2007, pp. 6–11.

[39] S.N. Srirama, V. Shor, E. Vainikko and M. Jarke, Scalable mobile web services mediation framework, in: *Fifth International Conference on Internet and Web Applications and Services (ICIW 2010)*, Barcelona, Spain, IEEE CS Press, 2010, pp. 315–320.

[40] L. Wang, J. Tao, M. Kunze, D. Rattu and A. Castellanos, The Cumulus project: Build a scientific cloud for a data center, in: *Cloud Computing and Its Applications*, Chicago, IL, USA, 2008.

[41] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker and I. Stoica, Spark: cluster computing with working sets, in: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10*, Berkeley, CA, USA, USENIX Association, 2010, p. 10.