SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **00**:1–27
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Implementation and Performance Evaluation of a Distributed Conjugate Gradient Method in a Cloud Computing Environment

Leila Ismail*, Rajeev Barua

Faculty of IT, Al-Ain, UAE & Electrical and Computer Engineering Department, University of Maryland, College Park, USA

SUMMARY

Cloud computing is an emerging technology where IT resources are provisioned to users in a set of a unified computing resources on a pay per use basis. The resources are dynamically chosen to satisfy a user Service Level Agreement and a required level of performance. A Cloud is seen as a computing platform for heavy load applications. Conjugate Gradient (CG) method is an iterative linear solver which is used by many scientific and engineering applications to solve a linear system of algebraic equations. CG generates a heavy load of computation and therefore it slows the performance of the applications using it. Distributing CG is considered as a way to increase its performance. However, running a distributed CG, based on a standard API, such as MPI, in a Cloud face many challenges, such as the Cloud processing and networking capabilities. In this work, we present an in-depth analysis of the CG algorithm and its complexity in order to develop adequate distributed algorithms. The implementation of these algorithms and their evaluation in our Cloud environment reveals the gains and losses achieved by distributing the CG. The performance results show that despite the complexity of the CG processing and communication, a speedup gain of at least 1,157.7 is obtained using 128 cores compared to NAS sequential execution. Given the emergence of Clouds, the results in this paper analyzes performance issues when a generic public Cloud, along with a standard development library, such as MPI, is used for High Performance applications, without the need of some specialized hardware and software.

Received . . .

KEY WORDS: Distributed Computing, Conjugate Gradient (CG) Method, Performance

1. INTRODUCTION

Conjugate Gradient (CG) [17] method is one of the most popular linear solvers to solve a system of linear equations in many scientific and engineering applications, such as oil reservoir simulation, aerospace vehicle guidance and control, circuit analysis, physics, etc. The CG is an effective method for symmetric positive definite systems. However, CG is computationally intensive, mainly due to the number of arithmetic operations involved in its equations. Therefore, enhancing the performance of the CG will enhance the performance of the applications using it. Distributed computation is meant to increase the efficiency of computing-intensive applications, as computational load is distributed among a number of workers. However, this efficiency decreases when communications have to take place between the distributed computational parts to reach the final solution.

While many researchers have been seeking the best way to distribute data and computing load of the CG method for specific parallel computing platforms, very few tackle the

*Correspondence to: Leila Ismail, Faculty of IT, UAE University, 17551 Al-Ain, UAE, Email: leila@uaeu.ac.ae

inter-communication issues between the computational load, and proposes and evaluates performance on a generic computing platform.

The Cloud Computing [1] [2] [3] is an emerging technology in which computation philosophy has shifted from the use of a personal computer or an individual, specialized server to a cloud of distributed resources. The main objective is to draw benefits from the underlying infrastructure services to satisfy a Service Level Agreement [7] for a user. As Cloud Computing is targeting a wide of public use, many of the standard and known libraries and tools should be available in the Cloud to Cloud developers as Software services to develop their own applications that can run in the Cloud. Message Passing Interface (MPI) is probably one of the most known and widely used tool to distribute computations in a distributed system environment. In this work, our cloud consists of computing resources, such as nodes, network and cores. A core here is an abstraction of the smallest processing unit that is allocated to an application. A core can be a context, a core or a processor. Our cloud uses provisioning of resources as defined by the National Institute of Standards and Technology [5], where nodes, network and cores are provisioned to run an application as requested by the application user. We use the Rocks technology [6] for provisioning. Our cloud does not use virtualization technique and images are deployed to run the CG application. There is a sense of location independence where our distributed CG application can run in any of the provisioned resources which fit the application. This is achieved by using the Sun N1 Grid Engine scheduler which schedules the application components to the underlying provisioned computing resources.

In this work, we study the impact of load distribution and communication strategies on the performance of the CG method using a Cloud of computing resources and MPI. Our parallel CG scheme has the following advantages:

- Portability. Our parallel CG scheme and implementation are not dependent on any specialized platform. Many of the works in CG evaluate the performance of CG on a specialized platform ([18], [19], [20] [21], [11], [12], [22]).
- Considering unstructured sparse matrix. Our parallel CG scheme considers a sparse matrix of unstructured nature which makes load balancing of matrix-vector multiplication more challenging. [22] introduces a parallel algorithm for matrix-vector multiplication, but considers a particular matrix with regular sparsity. On the other hand, our scheme is generic and can be applied to structured regular matrix shapes as well.
- Load balancing. Our scheme introduces a load-balancing model. Computational load is equally distributed to available computing resources in the system.
- Reducing communication overhead. Our scheme introduces an overlapping strategy to hide communication overhead. The communication is needed between computational resources to exchange data needed for a further step in the computation.

There is an interest in porting and implementing scientific applications on top of a Cloud computing environment [4]. However, to our knowledge, our technique is the first attempt to distribute the CG method in a Cloud computing environment using standard libraries, available in the Cloud. In particular, the CG method is implemented using MPI and it is distributed to the Cloud using the Sun N1 Grid Engine as an infrastructure for scheduling the computations. Significant performance can be obtained when considering load distribution and communication strategies in a Cloud of processors of the CG method. We implement these strategies and analyze their performance. Significant performance is obtained, encouraging implementation by other applications. The performance of our parallelization technique is evaluated by measuring the speedup gain of our parallel CG method against the scalar sequential; the former being demanding in terms of designing efforts, in particular when considering communication cost. We compare our results to the National Aeronautics and Space Administration (NASA) benchmark [16], which is formally an application of the inverse iteration algorithm for finding an estimate of the largest eigenvalue of a symmetric positive definite sparse unstructured matrix with a random pattern of non zeros. However, the essential

DOI: 10.1002/spe

part of the benchmark is the solution to an unstructured sparse linear system using CG. The NAS benchmark distributes the matrix row-wise and does not consider a load balanced approach. It uses a broadcast strategy to broadcast the value of the vector p after every iteration.

The rest of the paper is organized as follows: In Section 2, we provide the background for this work. The complexity analysis of the CG method is presented in Section 3. Section 4 describes the system model. In section 5, we present a dependency graph for the CG method computational components and discuss the communication issues in a distributed CG. Section 6 discusses a scheduling algorithm for the CG distribution. In Sections 7 and 8, we describe our parallel algorithm approach. Section 9 describes the inter-communication problem between the different CG computational parts and our approach to solve the problem. Section 10 overviews related works. The experiments and performance evaluation of our approach are presented in Section 11. Section 12 concludes our work.

2. BACKGROUND

CG method is one of the most popular iterative method for solving large systems of linear equations. These systems should be symmetric and positive definite. Many scientific and engineering applications generate such type of systems, such as structural analysis, and circuit analysis. CG method is also used in oil reservoir simulation techniques, as for instance to compute the pressure, the porosity, etc. of a reservoir. The CG can be applied to large linear systems due to its considerably smaller memory requirement as compared to other direct linear solvers.

The CG method solves a system of linear equations of the following form:

$$Ax = b$$

Where x is an unknown vector, b is a known vector and A is a known, square, symmetric, positive-definite or positive-indefinite matrix.

The CG algorithm finds the solution to a system of linear equations which represent an engineering application. For example, in oil reservoir simulation, the number of linear equations corresponds to the number of grids of a reservoir; the unknown vector x being the oil pressure of the reservoir. Each element of the vector x is the oil pressure of a specific grid of the reservoir.

However, as most of the linear solvers, the CG may take a long time to solve a system due to its complexity; i.e., the number of operations needed in order to solve a system of linear equations is relatively high. This complexity increases with the number of equations involved in the linear system. In particular, this complexity comes from the matrix-vector and vector-vector multiplications which are heavily involved when using CG.

As shown in Figure 1, the CG method starts with a random initial guess of the solution x_0 (step 1). Then, it proceeds by generating vector sequences of iterates (i.e., successive approximations to the solution (step 10)), residuals corresponding to the iterates (step 11), and search directions used in updating the iterates and residuals (step 14). Although the length of these sequences can become large, only a small number of vectors needs to be kept in memory. In every iteration of the method, two inner products (in steps 9 and 13) are performed in order to compute update scalars (steps 9 and 13) that are defined to make the sequences satisfy certain orthogonality conditions. On a symmetric positive definite linear system these conditions imply that the distance to the true solution is minimized in some norm (step 12).

3. COMPLEXITY ANALYSIS OF THE CONJUGATE GRADIENT METHOD

Figure 1 shows the algorithm of the CG method which consists of two main steps: initialization where the residual vector r_0 associated with initial guess of the solution x_0 is computed, and

DOI: 10.1002/spe

1. $x_0 = 0$
2. $r_0 := b - Ax_0$
3. $p_0 := r_0$
4. $k := 0$
5. $Kmax :=$ maximum number of iterations to be done
6. if $k < kmax$ then perform 8 to 16
7. if $k = kmax$ then exit
8. calculate $v = Ap_k$
9. $\alpha_k := \frac{r_k^T r_k}{p_k^T v}$
10. $x_{k+1} := x_k + \alpha_k p_k$
11. $r_{k+1} := r_k - \alpha_k v$
12. if r_{k+1} is sufficiently small then go to 16
    end if
13. $\beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
14. $p_{k+1} := r_{k+1} + \beta_k p_k$
15. $k := k + 1$
16. $result = x_{k+1}$

Figure 1. Conjugate Gradient Algorithm.

iterating the search to reach the solution. The initialization computes an initial search direction p_0=r_0, which is used in the first iteration of the CG computation loop. The subsequent values p_k are computed in subsequent iterations of the loop. In the first iteration of the loop, the value α_0 is computed. Then, the values of p_0 and α_0 are used to find the next value of x, namely x_1. This result (x_1) gives an *improved* approximate solution for the system. Now, the next residual vector r_1 is computed using the formula $r_1 = r_0 - \alpha_0 Ap_0$. If the value of resulting r is very small, then the loop terminates and the value of x is found to be x_1. Otherwise, the scalar β_0 is computed that will be used to determine the next search direction p_1 by using the relationship: $p_1 = r_0 - \beta_0 p_0$. The iteration number k is incremented by one and iteration is repeated starting from step 6. The search stops when either the residual value r becomes very small or a given maximum number of iterations has been reached.

For a matrix of $A(n, n)$ and vectors (x, r, and p) of length n each, the CG complexity consists mostly of matrix-vector multiplications and vector-vector multiplications. As CG is applied for sparse matrices, let us assume that there are m_i non-zeros elements in each row i for the matrix A.

In the following, we screen the total number of arithmetic operations (TNAO), computed from the number of multiplications (NM), summations (NS) and subtractions (NB), involved in the computation of the CG. In this analysis, we assume that every type of operation takes the same CPU computation time.

- For the initialization step, which is out of the CG *for* loop statement (step 2)
  NM(r_0)=$\sum_{i=1}^{n} m_i$,
  NS(r_0)=$\sum_{i=1}^{n}(m_i) - n$ and NB(r_0)=n
  where $i = 1, ..., n$ and n is the number of rows of the matrix A,
  $\Rightarrow$ TNAO(r_0)=$2\sum_{i=1}^{n} m_i$

For a single iteration, the CG involves the following complexity:

- For the matrix-vector multiplication $v = Ap_k$ (step 8)
  NM(v)=$\sum_{i=1}^{n} m_i$,

DOI: 10.1002/spe

NS(r_0)=$\sum_{i=1}^{n}(m_i) - n$
$\Rightarrow$ TNAO(r_0)=$2\sum_{i=1}^{n}(m_i) - n$

- For computing α_k (step 9)
  NM(α_k)=$2n$ and NS(α_k)=$2n-2$
  $\Rightarrow$ adding 1 division, then TNAO(α_k)=$4n-1$

- For computing x_{k+1} (step 10)
  NM(x_{k+1})=n and NS(x_{k+1})=n
  $\Rightarrow$ TNAO(x_{k+1})=$2n$

- For computing r_{k+1} (step 11)
  NM(r_{k+1})=n, and NB(r_{k+1})=n
  $\Rightarrow$ TNAO(r_{k+1})=$2n$

- For computing β_k (step 13)
  NM(β_k)=$2n$ and NS(β_k)=$2n-2$
  $\Rightarrow$ adding 1 division, then TNAO(β_k)=$4n-1$

- For computing p_{k+1} (step 14)
  NM(p_{k+1})=n and NS(p_{k+1})=n
  $\Rightarrow$ TNAO(p_{k+1})=$2n$

The total number of arithmetic operations (TNAO) for the whole for computing a CG is then equal to:

$$TNAO = 2\sum_{i=1}^{n} m_i + k_{max}(2\sum_{i=1}^{n} m_i + 13n - 2) \quad (1)$$

where k_{max} is the number of iterations involved in the computation of the CG.
In the section 7, we exploit the complexity analysis results of the CG method to determine a relationship between the CG method workload and the matrix row indices and the vector indices p, r, and x, to achieve an automatic load-balanced distribution of the CG.

4. SYSTEM MODEL

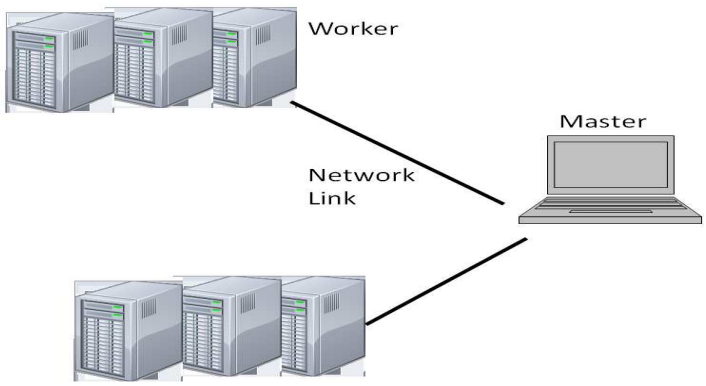

Figure 2. Computing Platform Model

The computing platforms that we are targeting are Clouds of computing resources. As shown in Figure 2, our platform model consists of the following components:

DOI: 10.1002/spe

- Nodes. A platform consists of a collection of nodes which are connected via a network link whose speed dictates the speed of the communication processing when data is transmitted among the nodes. Those nodes could be located at remote distance, and they are operating in an independent heterogeneous mode.
- Cores. We define a core here as an abstraction the smallest computing processing unit within a node. Our core could be a context, a core or a processor. Each individual node may consist of one or multiple cores, which depend on the built-in capacity in terms of computing power, memory limitation and communication protocols.

When an application is scheduled to be divided and run, one of the cores of the available computing nodes is selected at random as the master and other computing cores are selected as associate independent computing workers to perform partitions (tasks), which are assigned by the master core. As a result, a communication Star topology is formed between the master worker and the computing workers. Each worker (master or computing) can receive data from the network and perform computation simultaneously. The computing workers have computing capacity, noted μ_i, where $i = 1, ..., N$, and N is the number of computing workers in the system. The computing workers communicate with each others to exchange partial results. By the end of the algorithm, each computing worker would have participated in computing part of the final solution.

In particular, the computing platform model we consider consists of the following elements and assumptions:

- A Cloud is considered to have N computing workers. Each computing worker i, $i \in \{1, \ldots, N\}$, in the Cloud has a computing processing capacity of μ_i, where μ_i is the computation speed of the worker i in float operations per second. It also has a data receiving capacity C_i to receive data from the master.
- An application consists of a total of W_{total} computing float operations per second that can be divided into tasks that are assigned to available computing workers.
- Consider a portion of the total load, $task_i \leq W_{total}$, which is to be processed on worker i. We represent the time required for a computing worker i to perform the computation of $task_i$, $TComp_i$, as:

$$TComp_i = \frac{task_i}{\mu_i} \tag{2}$$

We represent the time for sending $chunk_i$ units of data to computing worker i, $TComm_i$, as:

$$TComm_i = \frac{chunk_i}{C_i} \tag{3}$$

where C_i is the data transfer rate in terms of units of data per second that can be provided by the communication link from the master to the worker i.

5. COMMUNICATION ISSUES IN CONJUGATE GRADIENT

In this section, we present the communications issues in a parallel CG. In particular, we devise a dependency graph among the computational components. This dependency graph gives directions of data flow within one iteration in the same worker and among the workers of the system for computing the distributed CG. We then discuss data distribution and communications cost.

DOI: 10.1002/spe

5.1. Data Dependency Graph

The complexity analysis in previous section shows the different computational components involved in the CG method. In each iteration of the CG method, each computational component can be parallelized to compute part of the output values: α_k, x_{k+1}, r_{k+1}, β_k, and p_{k+1}.

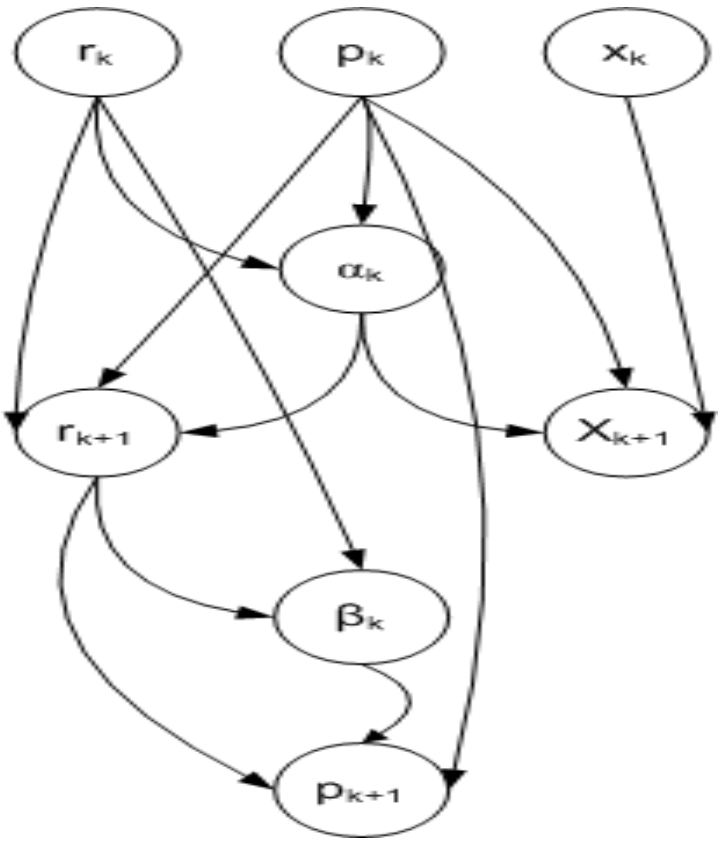

Figure 3. Dependency Graph in CG

Figure 3 shows data dependency graph, where values are dependent on other values which are connected to and which are higher than them in the graph representation. For example, α_k is dependent on r_k and p_k. Looking at the graph, one can analyze that the computation of the r_{k+1}, and p_{k+1} are dependent on global scalars α_k and βk respectively. It is obvious that a direct data level parallelization would not be possible without global communication between the workers, as the data inputs to every iteration r_{k+1} and p_{k+1} are globally inter-dependent because of the scalar values α_k and βk. In addition, the computation of α_k depends on the matrix-vector multiplication results, which in turn depends on the global vector p_k. So, in a next iteration, every worker will wait for the computational part p_{k+1} from all the other computing workers in order to get the elements of p_{k+1} that are needed to compute its own part of the matrix-vector multiplication.

Another global dependency is the checking of the global value r_{k+1} whether sufficiently small. Every computing worker must compute its part of r_{k+1} and participate in the computation of a norm ($Norm(r_k)$) which is sent to the master worker for checking. In summary, the CG method involves the global distribution of the scalars α_k, βk, and $Norm(r_k)$and the vectors p_k, and x_k. The latter is needed from every computing worker in order to construct the global solution x.

DOI: 10.1002/spe

5.2. CG Load Distribution

One of the main goals of a CG distribution algorithm is to reduce the CG complexity, by dividing the number of the CG operations by the number of available computing workers in the system. The flow chart presented in Figure 1 presents 2 types of divisible loads: 1) the matrix-vector multiplication presented in step 8 of the flowchart, and 2) the scalar-vector and/or vector-vector operations presented in steps 9, 10, 11, 13, and 14 of the flowchart. Our load distribution strategy follow a master-worker model, widely used in distributed applications.

The master worker divides the matrix-vector multiplication load equally among the computing workers in the system. That means, based on the sparseness shape of the matrix, the computing workers may get different number of rows of the matrix. Every computing worker also participates in computing part of the scalar-vector and vector-vector manipulations which are needed to compute α_k and β_k. Consequently, in every iteration, every computing worker computes an equal part of the α_k numerator and denominator. The distribution of α_k and β_k is based on the distribution of the vector r_k (part of the numerators and denumerators). The vector r_k follows the decomposition model of the matrix A, which means the number of elements of r_k updated by every computing workers is equal to the number of rows distributed to every computing worker. In every iteration, every computing worker communicates these values to the other computing workers in the system so that the global α_k is known by every computing worker to continue its execution. Each computing worker then uses the α_k to compute its part of x_{k+1} and r_{k+1}. Similarly to r_k, x_{k+1} and r_{k+1} is decomposed in a way that every computing workers computes number of elements that is equal to the number of rows of its local matrix; part of the matrix A. Each computing worker computes a norm of r_{k+1} ($Norm(r_{k+1})$) and sends the value to the master worker who will verify whether the solution is reached. If the solution is reached, every computing worker sends to the master worker its part of the solution of x. However, if the solution is not reached, then it is necessary to compute a new search direction p_{k+1}, which is needed to re-iterate the CG. Each processor then needs to compute an equal part of p_{k+1}. Therefore, every computing worker computes part of the load of β_k numerator and denominator. The numerators and denominators are then sent to all other computing workers in the system to compute the global β_k. Based on β_k, each computing worker computes an equal part of p_{k+1}, which is, as mentioned previously, necessary to start the next iteration. Every computing worker updates a number of elements of p_{k+1} which is equal to the number of rows of its local matrix, that is part of the matrix A.

6. SCHEDULING PARALLEL CG IN A CLOUD COMPUTING INFRASTRUCTURE

Initially, the master distributes the matrix elements to all the computing workers of the system. The matrix elements are distributed in a way to have an equal number of operations of the CG matrix-vector multiplication by each computing worker i. Only the non-zero elements of the CG matrix are distributed. Section 7 explains a load balanced distribution algorithm of the matrix to the computing workers. Each set of non-zero elements, denoted by nnz_i, which is sent to computing worker i, generates a value $task_i$. In this context, $task_i$ represents the computational load (total number of operations) generated in a single iteration in each computing worker. It constitutes the load to compute part of v, α, x_{k+1}, r_{k+1}, β, and p_{k+1}, denoted by $Part(v, \alpha, x_{k+1}, r_{k+1}, \beta, p_{k+1})$. The transmission of nnz_i chunk to the computing workers is done in a sequential fashion.

The time required for the worker i to perform the $task_i$, in a single iteration, is defined as:

$$TComp_i = \frac{task_i}{\mu_i} \tag{4}$$

where,

DOI: 10.1002/spe

$$task_i = \sum Part(v, \alpha, x_{k+1}, r_{k+1}, \beta, p_{k+1}) \quad (5)$$

In CG, there are two types of communication delays: master-workers communications delay and inter-workers communications delay.

The master-workers communications delay, denoted by $TComm_{mw}$, is obtained by the following formula:

$$\begin{aligned} TComm_{mw} &= TComm_{nnz} + k_{max} * TComm_{Norm(r_k)} \\ &\quad + TComm_{x_k} \end{aligned} \quad (6)$$

where,

k_{max} is the number of maximum iterations performed to reach the final solution when running the CG.

$TComm_{nnz}$ is the time spent by the master sending nnz_i to every computing worker i. It is obtained by the following formula:

$$TComm_{nnz} = N * TComm_{nnz_i} \quad (7)$$

$$= N(\frac{nnz_i}{C_i}) \quad (8)$$

$TComm_Norm(rk)$ is the communication time spent by the computing workers communicating the value $Norm(r_k)$ to the master within the iteration k. All the computing workers send their their data simultaneously on the network. However, as there is no synchronized clock and the start of sending data on each computing worker cannot be comparable, then the time of communicating data over the network is determined by the finishing time of the last computing worker i $(i = 1, ..., N)$ sending its value of r_k to the master. Consequently, we obtain the following formula:

$$\begin{aligned} TComm_Norm(rk) &= Maximum(TCom_{1fr}, \\ &\quad TComm_{2fr}, ..., \\ &\quad TComm_{ifr}, ..., \\ &\quad TComm_{Nfr}) \end{aligned} \quad (9)$$

where,

$TCom_{ifr}$ is the finishing time of sending the scalar $Norm(r)$ from the computing worker i to the master.

$TComm_{xk}$ is the spent to communicate the vector x_k from the computing workers to the master in the last iteration.

$$\begin{aligned} TComm_{xk} &= Maximum(TCom_{1fx}, \\ &\quad TComm_{2fx}, ..., TComm_{ifx}, ..., \\ &\quad TComm_{Nfx}) \end{aligned} \quad (10)$$

where,

$TComm_{ifr}$ is the finishing time of sending the elements of the vectors x from the computing worker i to the master.

The inter-workers communications delay, denoted by $TComm_{ww}$, is obtained by the following formula:

DOI: 10.1002/spe

$$TComm_{ww} = k_{max}(TComm_{Part(\alpha)} + TComm_{Part(\beta)} + TComm_{Part(p_k)}) \quad (11)$$

where, $TComm_{Part(\alpha)}$ is the time to distribute part of the numerator and denumerator of α from each computing worker to all other computing workers. $TComm_{Part(\beta)}$ is the time to distribute part of the numerator and denumerator of β from each computing worker to all other computing workers. And, $TComm_{Part(p_k)}$ is the time to distribute the part of the vector p_k from each computing worker to all other computing workers in a single iteration. The numerator and the denumerators are of scalars type. The number of elements of p_k to exchange among the workers is dependent on the matrix distribution policy of the CG. In addition, this communication cost depends on whether there is an overlap between communication and computation on each computing worker and the percentage of this overlap. In the next sections, we explore different algorithms and their impact on this communication cost.

7. RELATIONSHIP BETWEEN WORKLOAD AND MATRIX DOMAIN DECOMPOSITION

A blind decomposition of the matrix equally over the number of available computing workers, whether horizontally or vertically, may not provide a load balanced distribution of the workload assigned to each computing worker in each iteration of the CG method. Depending on the degree of sparseness of the matrix, some rows may have more or less non-zero elements than other rows. The total number of arithmetic operations required to multiply a matrix $A(n,n)$ by a vector p, $TNAO(Ap)$, is equal to:

$$TNAO(Ap) = 2\sum_{i=1}^{n}(m_i) - n \qquad i = 1, ..., n \quad (12)$$

where,

m_i is the number of non-zero elements in each row i of the matrix A.

To identify equal chunk of computation, we identify a relationship between a workload and the indices of the rows that should be assigned to each computing worker. A row index k is a row number. Each row of a matrix $A(n,n)$ has a number k, $1 \leq k \leq n$, where n is the number of rows for the matrix A. For example, $k = 1$ defines the row number 1 of the matrix A. The main goal is to identify equal parts of computation to be distributed to all the workers of the system. We define the cumulative workload $W[1:k]$ as the total number of arithmetic operations needed to multiply the sub matrix $A(k,n)$ by the vector p for all i, $1 \leq i \leq k$. Based on eq. 12, we obtain the following equation, which defines a relationship between the cumulative workload the k indices:

$$W[1:k] = 2\sum_{i=1}^{n}(m_i) - k \qquad k = 1, ..., n \quad (13)$$

The number of non-zero elements corresponding to a cumulative workload $W[1,k]$ can be obtained by the following equation:

$$nnz = \frac{W[1:k]}{2} + 1 \quad (14)$$

Based on eq. 13 and eq. 14, we obtain the following equation:

$$k = 2nnz - W[1:k] \quad (15)$$

The above equation gives the row-index of a matrix, based on cumulative value of workloads.

DOI: 10.1002/spe

8. LOAD BALANCED DISTRIBUTION OF THE CG METHOD

Our objective is to balance the CG method workload over all the processors of the system. An algorithm is needed to equally distribute the calculations to be performed without much overhead in communication. The complexity analysis shows that the most consuming part of the CG algorithm lies in the matrix-vector multiplications. An algorithm is needed to equally distribute the calculations to be performed. Our load balancing model calculates the total work required to be conducted (Equation 12) and distributes it to the number of available computing workers in the system according to a certain ratio. This ratio could be denoted as Work Per Unit (WPU). Each worker calculates a part of the resultant vector so that the work done by each worker is the same. If the computing resources are homogeneous, then WPU is the ratio of the total workload to the number of available workers in the system: $\frac{TNAO}{N}$. In case of heteregeneous resources, the computing performance of every computing worker involved in the computation of the distributed CG should be taken into account in the load-balanced approach. Let us denote by $nnz_i = WPU[i]$, the number of non-zero elements of the matrix that should be distributed to each computing resource i. To have a load balanced distributed CG on heteregeneous resources, the following formula should hold when running the operations on computing workers:

$$\frac{nnz_1}{\mu_1} = \frac{nnz_2}{\mu 2} = ... = \frac{nnz_i}{\mu_i} = ... = \frac{nnz_n}{\mu_n} \qquad i = 1, ..., n \tag{16}$$

where, nnz_i $(1 \leq i \leq n)$ is the number of non-zero elements that should be distributed to each computing resource.

Also, we have:

$$nnz = \sum_{i=1}^{n} nnz_i \qquad i = 1, ..., n \tag{17}$$

Based on Equations 16 and 17, the number of non-zero elements to be distributed to the first computing woker of computing power μ_1 is obtained by the followng formula:

$$nnz_1 = \frac{nnz}{1 + \frac{\sum_{j=1}^{n} \mu j}{\mu_1}} \tag{18}$$

Based on Equations 16 and 18, the number of non-zero elements to be distributed to the remaining computing workers of computing powr μ_i $(i = 1, ..., n)$ is given by the following formula:

$$nnz_i = \frac{\mu i}{\mu_1} nnz_1 \qquad i = 2, ..., n \tag{19}$$

The formula for calculating the end index k of the matrix block assigned to each worker accepts the value $W[1:k]$, which is the cumulative value of the workloads, Ws, of the previous assignments to other workers, and returns the value of k based on Equation 15. As shown in the pseudo code below, WPU is the unit of load that should be distributed to each computing worker, while $totalwork$ is that total load of the matrix-vector multiplication.

DOI: 10.1002/spe

```
LoadBalance(int * workers, int * wpu, int nnz, int * m)
 1  ▷ Compute the total work to be done
 2  int totalwork = 0, temp = 0, work = 0,
 3  i = 1, k = 0, size = 0;
 4
 5  for i = 1 to i = n
 6        do totalwork = totalwork + m[i]; i = i+1;

 7  totalwork = totalwork -n;
 8  i = 0;
 9  while work <= totalwork − 1
10        do
11           ▷ The cumulative workload
12           ▷ of previous assignments
13           Work = work + wpu[i];
14           k = 2nnz-Work;
15           ▷ Each worker holds the end index
16           ▷ of the matrix chunk that is to be
17           ▷ distributed to worker[i]
18           end-row-orker[i]=k;
19           i = i+1;
```

The decomposition of the vectors x, r, and p follows the decomposition of the matrix $A(1, n)$. Consequently, the index k, calculated by the equation eq. 15 is used to calculate the number of elements of the vector x, r, and p, which are computed by each computing worker during an iteration. consequently, every computing worker computes a chunk of x, r, and p in step 10, step 11, and step 14 respectively.

9. INTER-WORKERS COMMUNICATIONS DELAY

As described previously, one of the main challenges of a distributed CG is the inter-workers communications delay. This delay adds up to the total execution time of a distributed CG. As mentioned in section 6, the inter-workers communications delay consists of exchanging over the network the scalar values $(Part_{(\alpha)}, Part_{(\beta)})$, and the vector p in each iteration of a CG distributed algorithm. Step 14 shows that a new p is calculated in each iteration and it is used in calculating the local matrix-vector-multiplication v (step 8) in the next iteration. Consequently, a new p has to be made available to all the computing workers for a next iteration, which requires inter-workers communication. Table I shows the results of experiments we conducted on our Cloud of 16 Intel Xeon machines connected by Infiniband switch, making a total of 128 cores. In those experiments, we send a simultaneous broadcast from a computing core to all other computing cores and we measure the communication cost from one of the computing workers. We repeated the experiments 100 times and we have taken the average. The experiments show that for large problem sizes, the communication cost of a scalar is minimal compared to the communication cost of a vector as the vector size to communicate becomes large. Consequently, in this study, we then focus on the communication cost induced by communicating the vector p among the workers. The rest of this section describes 2 implementation strategies and their impact on the communication cost. In both strategies, the matrix is decomposed based on the load-balanced approach described in section 8.

DOI: 10.1002/spe

Table I. Experimental Simultaneous Broadcast of a Scalar vis-a-vis a Vector

Experiment	Time in Seconds
Broadcast of a scalar	0.0.0002671
Broadcast of a vector of size 10	0.0002716
Broadcast of a vector of size 100	0.0003606
Broadcast of a vector of size 1000	0.0004491
Broadcast of a vector of size 10000	0.0011064
Broadcast of a vector of size 100000	0.005383
Broadcast of a vector of size 1000000	0.0500894

9.1. Communication and Computation Non-Overlap-Based Strategy

Figure 4 shows an example of a matrix which is decomposed over 4 computing workers. Our first approach for exchanging p is for every computing worker to gather a vector p of size l_i $(1 \leq i \leq N)$ from $(N-1)$ computing workers participating in the CG distributed computation; i.e., a broadcast approach. l_i is equal to the number of rows associated to each computing worker based on the load-balanced approach. The total communication cost is as follows:

$$TComm_p = TBcast(l_i) * k_{max} \qquad i = 1, ..., N \tag{20}$$

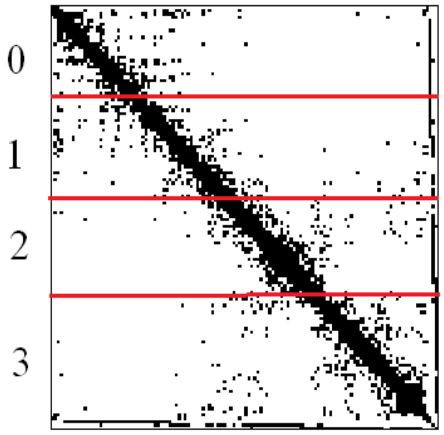

Figure 4. Data Distribution of a Matrix on 4 Computing Workers

DOI: 10.1002/spe

9.2. Communication and Computation Overlap-Based Strategy

In this algorithm, a computing worker uses asynchronous communication, in which a vector p is sent in a non-blocking way, allowing a computing worker to complete the rest of the iteration. In this approach, in the worst case, the cost of communication is:

$$\begin{aligned} TComm_p &= T(IReceive(l_i) + Wait) \\ &\quad *(N-1) * k_{max} \;\; i = 1, ..., N \end{aligned} \tag{21}$$

where $IReceive$ represents asynchronous reception of the data by a computing worker, and $Wait$ represents the waiting time by a worker to receive the data if the data has not been received in the receiver buffer yet.

However, by using simply asynchronous communication, a computing worker still cannot start its next iteration unless all the elements of the vector p have been received from $(N-1)$ computing workers. Therefore, we introduce a new overlap approach, in which to make use of the idle communication time while p is communicated to perform part of the vector-matrix multiplication in a next iteration. In this approach, we first divide the matrix based on our load-balanced approach. Then locally, within every computing worker, the local matrix is divided into vertical blocks to accommodate the communication-computation overlap. The vertical division uses the row indices k found in horizontal distribution. Figure 5 shows an example of the decomposition, where every computing worker splits its local matrix into $N = 4$ vertical blocks.

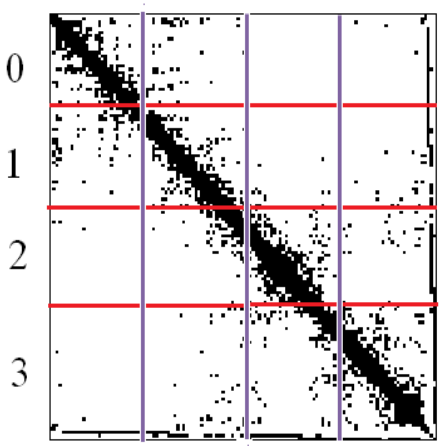

Figure 5. Data Decomposition to Accommodate Communication-Computation Overlap

The algorithm works as follows. For the entire local matrix-vector-multiplication, every computing worker needs the whole p vector. Every computing worker divides its local matrix-vector-multiplication into N steps. Initially, every computing worker has its own part of the vector p. In each step, before starting the local matrix-vector-multiplication, a computing worker sends its own part of the vector p, in a non-blocking communication, to the left neighbor and simultaneously receive part of the vector p from right neighbor forming a ring of communication. The communication takes place in the form of a ring as presented in Figure 6 for a set of 4 communicating computing workers. Initially, computing worker $Rank_0$ has chunk p_o. Consequently, computing worker $Rank_0$ sends p_0 to computing worker $Rank_3$. Likewise, computing worker $Rank_1$ has chunk p_1, then $Rank_1$ sends chunk p_1 to computing worker $Rank_0$. Similarity, computing worker $Rank_2$ has chunk p_2, so $Rank_2$ sends p_2 to computing

DOI: 10.1002/spe

worker $Rank_1$. Finally, computing worker $Rank_3$ has chunk p_3, which it sends to computing worker $Rank_2$

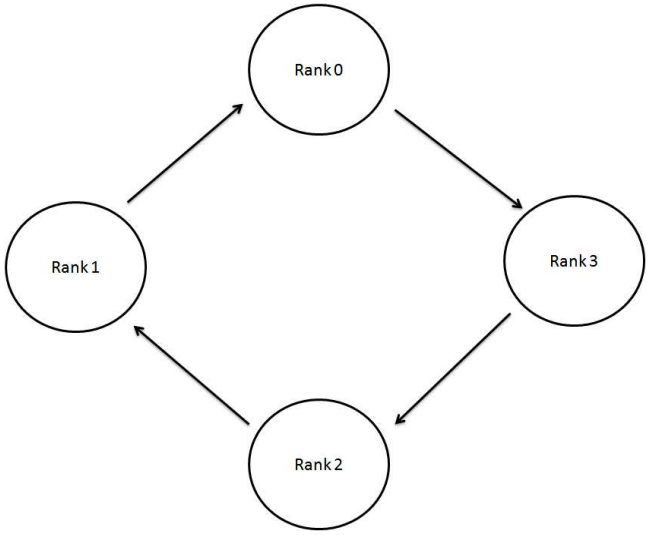

Figure 6. Ring-Based Communication for Communication-Computation Overlap Strategy

After initiating the communication, all the computing workers start the local matrix-vector multiplication to find the vector v. The local matrix-vector multiplication starts on the block number for which the computing worker has its own chunk of p. Figure 7 illustrates the starting computational part in each computing worker. The local matrix-vector multiplication is performed using the non-zero elements of the respective blocks, as illustrated in Figure 8.

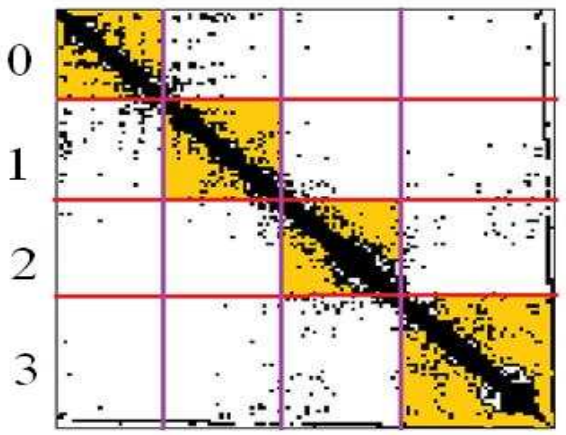

Figure 7. Initialization of Computing in Communication-Computation Overlap Strategy

Before starting the next step of computation of its local matrix-vector multiplication, as shown in Figure 9, every computing worker sends the chunk p received from the left neighbor and receives the chunk p from the right neighbor which will be used for computing the matrix-vector multiplication on the next vertical block, and the partial values of v computed in the current step will be added to the values of v computed in the previous step. Figures 10 and 11 illustrate the remaining steps.

DOI: 10.1002/spe

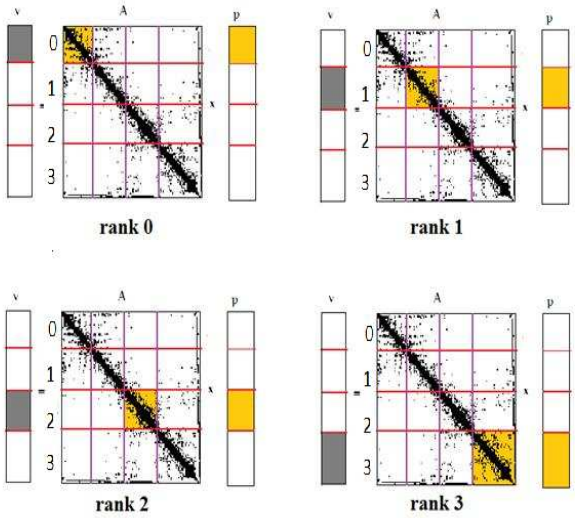

Figure 8. Step 1 of Computation in Communication-Computation Overlap Strategy

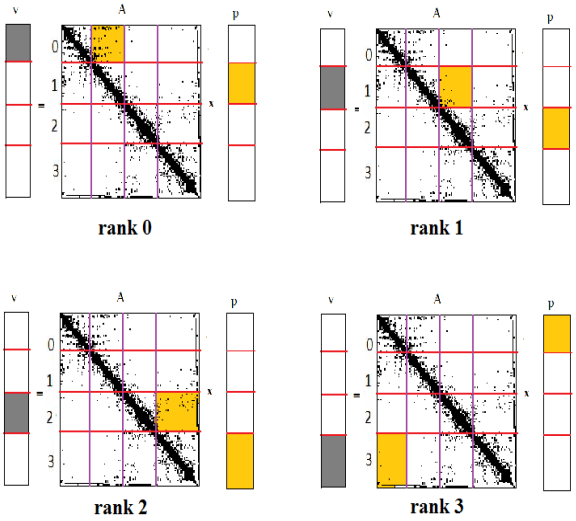

Figure 9. Step 2 of Computation in Communication-Computation Overlap Strategy

10. RELATED WORKS

Several algorithms have been published to parallelize CG [18], [19], [20] [21], [11], [12]. In [18], [19], [20] and [21], algorithms have been implemented on top of a specialized event-driven multi-threaded platform. In [11] and [12], algorithms have been implemented on top of a distributed shared memory cluster. [22] introduces a parallel algorithm for matrix-vector multiplication, but considers a particular matrix with regular sparsity and studies the impact of mesh partitioning on the performance. [25] and [27] introduce data decomposition strategies for matrix-vector multiplications to increase CG efficiency on hypercubes and mesh networks for unstructured sparse matrix. Blocks of matrix are assigned to processors to perform partial result of the matrix-vector multiplication. Non-zeros are transferred using storage techniques of non-zeros in a vector. But the algorithm does not consider a load-balanced distribution of the operations of the matrix-vector multiplication, the communication costs involved in the

DOI: 10.1002/spe

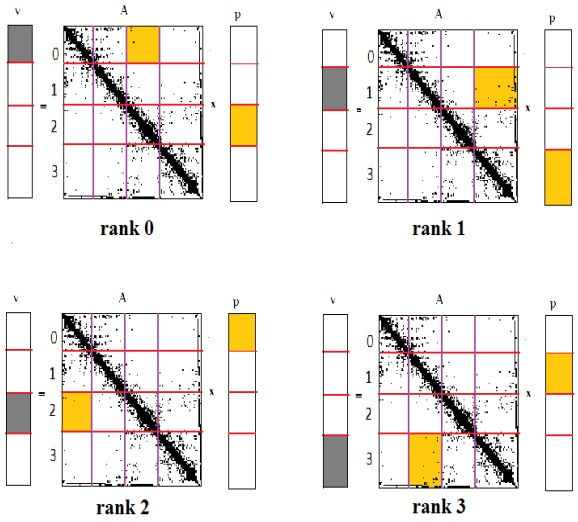

Figure 10. Step 3 of Computation in Communication-Computation Overlap Strategy

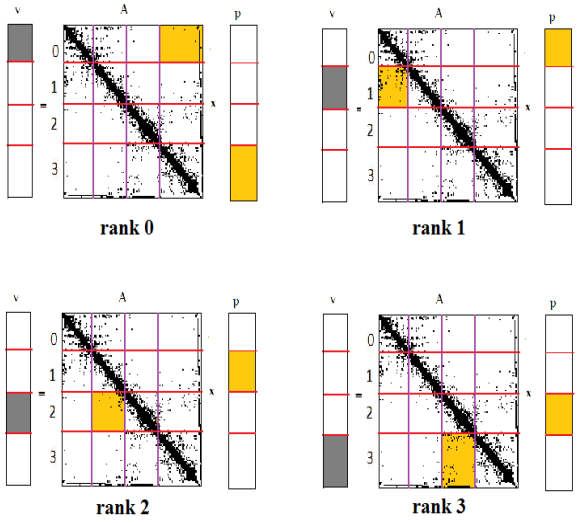

Figure 11. Step 4 of Computation in Communication-Computation Overlap Strategy

global CG method. Our experiments with the broadcast-based approach give the same results as in [25]; i.e., the approach does not scale with the increasing number of computing workers. In [25], a ring-based overlap mechanism is used for global summation within the CG method, a speedup of of 2.5 was obtained on 128 cores compared to the original NAS. [26] presents communication-avoiding algorithms to decrease communication costs of applications. We used a ring-based overlap mechanism for collecting a vector within the CG method, giving a speedup of 1,157.7 on 128 cores compared to NAS.

[28] and [29] divide the overall CG algorithm into blocks of algorithms to reduce communication, but did not address the cost of communication among the blocks. Jordan et al. [23] reported experiments where matrix-vector multiplication loads are distributed based on processors speed in a heterogeneous cluster, using a dense matrix. Other works concentrate on the ways to store the non-zero values of a sparse matrix in the CG method. [30] produces a speedup of 3.7876 when using 16 processors compared to the CG sequential execution.

DOI: 10.1002/spe

[24] is based on [23], but uses a special method for storing sparse coefficient matrices where only non-zeros are stored taken into account during communication. The parallel algorithm suggested by Kim et.al [10] explored the parallel conjugate gradient solver with both Jacobi diagonal preconditioning and incomplete Cholesky factorization preconditioning on a number of different meshfree analysis applications. They investigated the parallel performance of the solver using a homogeneous cluster and obtained a speedup of 12 compared to sequential execution in both cases. The cluster used consists of 12 processors.

11. PERFORMANCE ANALYSIS

In this section, we analyze the performance of our CG method parallel algorithms (broadcast and overlap). We compare the performance of our overlap approach compared to the NAS CG parallel benchmark [16]. We also report on the gains and losses obtained while implementing and evaluating our parallel CG method algorithms on top of our Cloud computing infrastructure. In terms of performance, though it could be expected that the overlap approach will perform better than the broadcast approach, our experiments reveal the speedup factor provided by our overlap technique compared to the broadcast technique.

11.1. Experimental Environment

The experiments are conducted on a Cloud of 16 Xeon Intel Quad Core 5355 machines with 2.66 GHz CPUs. Each machine has a dual CPU. Each core has 4MB of cache, 1GB of memory, 2.66 x 4GFLOPS of peak performance. The machines are connected using an InfiniBand (IB) standard network. The operating system used on the machines is Red Hat Enterprise Linux Server release 5.2. The experiments are done using the parallel capabilities of a single multi-core machine, and using a Cloud of machines. Message Passing Interface (Open MPI version 1.3.2) library is used for implementing the parallel CG. For the sake of comparison with MPI-based NAS CG method parallel benchmark, the NPB3.2.1 version of the NAS code was used to run both sequential execution of the CG method and the NAS parallel benchmark.

To analyze the performance of the parallel CG algorithms on a heteregeneous environment, we run experiments using a heteregeneous Cloud made of 2 AMD Opteron processors and 7 Intel Xeon processors. The AMD Opteron Processor model is 252 with 2.59 GHz dual CPU single core. Each core has 1MB of cache, and 2 GB of memory. 5 out of the 7 Intel machines have Intel Xeon CPU of 3.06 GHz, dual CPU, dual core. Each core has 512KB of cache and 4GB of memory. Let us denote those Intel machines by Intel Type1. 1 out of 7 Intel machines has Intel Xeon CPU of 3.06 GHz with dual CPU, single core. Each core has 512 KB of cache and 4 GB of memory. Let us denote this Intel machine by Intel Type2. The AMD Opteron machines along with those 6 Intel machines described earlier are connected to each others via one hop 1Gb/second switch, those are connected to another Intel machine located in the shared LAN network via two-hop connectivity of 1Gbits/second. That Intel machine has Xeon CPU of 3.0 GHz and it has a single CPU, dual core. Let us denote that Intel machine by Intel Type3. Each core has 4MB of cache and 2GB of memory. The Linux Suse 3.1.0. was used. 16 computing workers (cores) from the heteregeneous platform were used to run the experiments. The workers are mapped to their corresponding machines as shown in Table II. Figure 12 shows the performance of the machines in terms of MFlops, measured by running the class C of the NAS benchmark on the platform.

11.2. Experiments

The speedup of the parallel CG versus its sequential execution is measured. In our experiments, one core acts as a master which distributes the tasks to the other cores that we call workers. The overall execution time for CG in a parallel environment is the elapsed time when master starts distributing the matrix till the final results are received by the master from the workers. The *gettimeofday* function is used to compute the elapsed time on the master. In the sequential

DOI: 10.1002/spe

Table II. Heteregeneous Computing Workers Forming the Experimental Heteregeneous Testbed

Computing Worker	Machine		
	CPU Type	Memory Size	Cache Size
CW0 and CW1	AMD Opteron Processor	2GB	1MB
CW2 and CW3	AMD Opteron Processor	2GB	1MB
CW4 and CW5	Intel Type 1	4GB	512KB
CW6 and CW7	Intel Type 1	4GB	512KB
CW8 and CW9	Intel Type 1	4GB	512KB
CW10	Intel Type 3	4GB	512KB
CW11 and CW12	Intel Type 1	4GB	512KB
CW13 and CW14	Intel Type 1	4GB	512KB
CW15	Intel Type 2	4GB	512KB

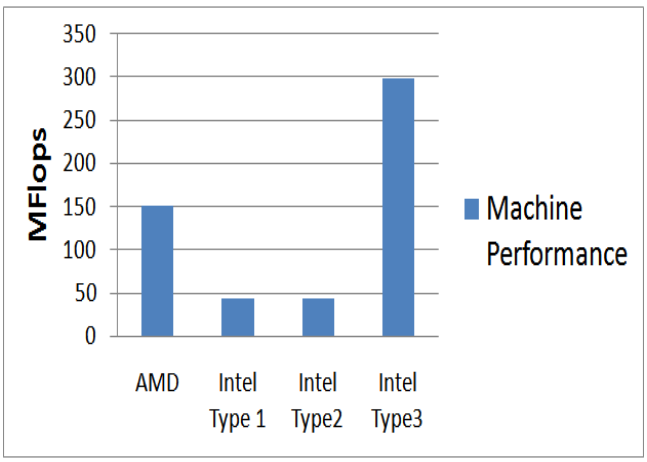

Figure 12. Computing performance of our experimental heteregeneous platform.

execution case, the *gettimeofday* function is used as well to compute the overall run time. In all our experiments, each experiment was run 100 times and the average was computed.

The experiments use different matrix sizes of A (total of 5 experiments or runs as shown in Table III to assess the impact of the matrix size on the performance. The matrix sizes are the same used by the NAS parallel CG benchmark. We also use the same sparsity patterns as implemented by NAS. In each run on the homogeneous platform, the parallel CG is measured by horizontally scaling the number of cores up to 128 cores. On the heteregeneous platform, each run has executed using the load-balanced approach. In the load-balanced approach, we

Prepared using speauth.cls
DOI: 10.1002/spe

Table III. Experimental Runs

Run	Workload		
	Benchmark Name	Matrix Size(A)	Number of Non-Zero Elements
1	S	1400	78,148
2	W	7000	508,402
3	A	14000	1853104
4	B	75000	13,708,072
5	C	150000	36,121,058

used our formulas to calculate the number of non-zero elements that should be distributed to each worker based on its available computing power.

To measure the computation workload distribution among the computing workers, the average execution time is measured on each computing worker. The average communication time; i.e., the data transfer time between the master and the computing worker, as well as among the computing workers (inter-communication), is also measured on each computing worker. In the nonparallel scenario, the *gettimeofday* function is used to compute the time elapsed. In the parallel scenarios, the *gettimeofday* function is used as well to compute the CG method time elapsed from the master side, as well as the time elapsed on each computing worker.

11.3. Experimental Results Analysis

In devising a parallel algorithm for the CG method to be distributed among the processing units of the system, the following requirements are considered. These requirement considerations have an effect on the experimental results obtained.

a) Load Balancing: One of our objectives is to balance the CG method workload over all the computing workers of the system. An algorithm is needed to distribute the calculations to be performed without much overhead in communication.

b) Parallelism: The Communication induced by the distribution strategy between the computing workers should be minimal or absent. Our parallel algorithm do not include any communication overhead. We allocate to every computing worker a part of the total workload and the result is transferred from a computing worker to the master.

c) Features of the CG Method: The CG method presents the following features that have to be considered when developing the parallel algorithm:

- The CG method presents global dependencies between its steps within a single iteration. These dependencies introduce inter-computing-workers communication in a parallel CG method, and consequently increase its total execution time. In our implementation, we use an overlap strategy of communication and computation.
- The CG method reaches a final solution when the global vector r reaches a minimum value. In our implementation, the master coordinates that process, which impose an extra overhead in communication in every iteration.

d) Portability: Portability is required in 2-fold:

- From an algorithm perspective, we tried to design a distribution strategy and an overlap scheme which are quite general and applicable to different applications presenting the same nature of problems as the CG method.

DOI: 10.1002/spe

- From a software perspective, we use MPI library to implement our approaches, which is a very well known and a standard library used for distributed and parallel implementations. We do not want to use any specialized hardware or software. We believe this is consistent with Cloud perspectives, as Cloud software developers and users can develop and run their applications on a generic public Cloud, using standard libraries and interfaces.

At the functional level, while it looks quite intuitive that the overlap approach for inter-computing workers communication will achieve better performance than a broadcast approach, yet our first attempt was to use the broadcast approach, as the corresponding programming model is simple and straightforward. However, the broadcast cost increases with the data size to broadcast; i.e., the size of the vector p, and the increasing number of computing workers, and makes the broadcast approach inefficient. On the homogeneous platform,an inspection of the overall speedup of the broadcast-based approach vis-a-vis the parallel version of the NAS benchmark (Figure 13) shows that with the increasing number of computing workers, the broadcast-based approach performs better than NAS for small matrix sizes (66 times better for benchmark S and 30 times better for benchmark W). However, the speedup of the broadcast-based approach declines considerably for large matrix sizes and for a large number of computing workers, e.g., the broadcast-based approach speedup is only only 3.8 times better than NAS for benchmark B and 4.6 times better than NAS for benchmark C using 128 computing workers. Furthermore, in our experiments the communication cost is load balanced over all the computing workers in case of a homogeneous platform, as shown in Figures 14 for runs which use 16 computing workers.

The increase in communication cost of the broadcast approach has led us to devise a communication-computation overlap strategy in which the communication of the vector p from one computing worker to all other computing workers is done in parallel to the computation of the local matrix-vector multiplication that has to be performed by every computing worker. An inspection of the overall speedup, obtained by the overlap method vis-a-vis broadcast-based approach (Figure 15, shows that the overlap-based approach performs 24.2 times better than the broadcast-based approach for big matrix sizes and large number of computing workers involved in the computation. Figure 16 shows a comparison of performance of the overall execution times of the overlap-based approach, the broadcast-based approach and the NAS benchmark. As expected, the overlap-based approach has in general better performance than the cited algorithms, as the cost of communication is hidden. As Figure 17 shows, the overlap-based approach performs 169.6 times better and 144.4 times better than NAS benchmark for classes B and C respectively. However, for small matrix sizes (sizes S and W), the broadcast approach performs better than the overlap approach. The performance of the overlap-based approach decreases till 0.14, for class S on 128 computing workers, compared to the broadcast approach, as the computation performed by every computing working in the overlap approach, as the computation load on a computing worker does not overweight the communication cost induced by sending a vector over the network from that computing worker. In other words, the computing workers spend much time waiting for the communication to take place than computing. In summary, the overlap-based approach is not much interesting for small matrix sizes and a big number number of computing worker compared to the NAS distributed benchmark. However, the overlap-based approach 1,157.7 times faster than the NAS sequential execution for matrix of size C using 128 computing workers. The broadcast-based distributed algorithm is 50.8 times faster than the NAS sequential execution for matrix of size C using 128 computing workers. However, in addition to the advantages of the overlap-based approach for large matrix sizes, this super-linear speedup is explained by the fact that for measuring the sequential execution of the NAS benchmark, we used its parallel version with NPROCS=1 which introduces some overhead compared to a *true* serial version.

Regarding load balancing over the computing workers, Figure 18 shows that the CG load is equally distributed over the computing workers. In this load-balanced-based approach, during experiments, we notice small discrepancies among the workloads of the computing workers.

DOI: 10.1002/spe

Table IV. Total Execution Time in Seconds of the Distributed CG Method on our Heteregeneous Testbed

Experiment	Matrix Size				
	Class S	Class W	Class A	Class B	Class C
Broadcast-Based CG Method	0.058722533	0.151632667	0.265799667	7.59671	21.21196667
Overlap-Based CG Method	0.041212767	0.104833333	0.191004667	3.959763333	7.825793333
NAS Benchmark	0.88	3.653333333	7.093333333	214.3366667	406.8833333

This is due to the rounding we do to each resultant chunk allocated to each computing worker to ensure that a matrix row would not be divided among the computing workers. Dividing a row among computing workers will impose exchange of messages for calculating a local matrix-vector multiplication, and thus increase the CG method communication cost.

On a heteregeneous platform, Table IV shows that the overlap approach is performing 2.7 times better than the broadcast-based approach, and 51.99 times better than the NAS benchmark for the class C matrix size.

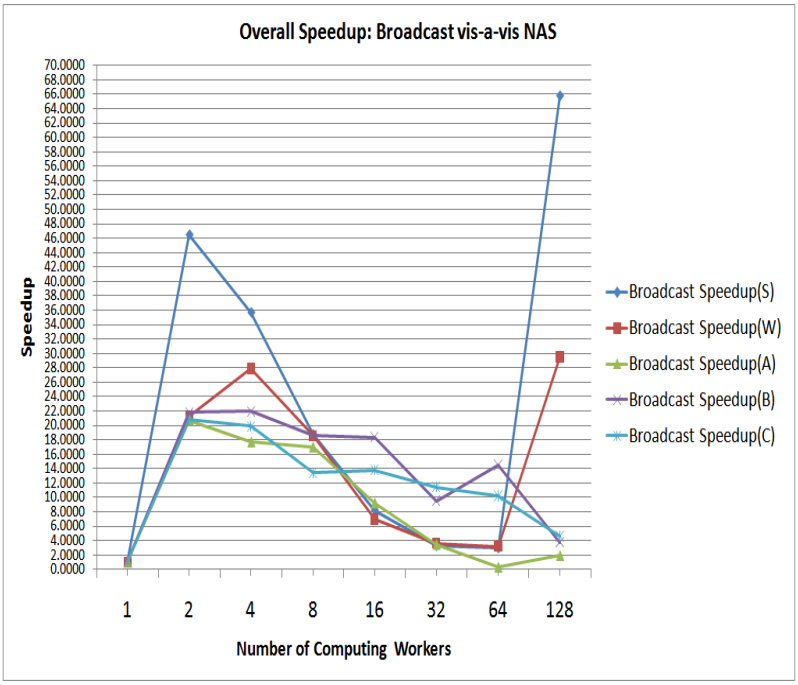

Figure 13. Overall speedup of broadcast-based approach vis-a-vis NAS distributed benchmark.

12. CONCLUSION

CG method is a mathematical tool used in a wide range of engineering and sciences applications. In particular, it is used because of its rapid convergence to solution. However, CG method is computationally challenging, in particular when the matrix becomes more dense. In this work, we studied the complexity of the CG method in terms of number and introduced a dependency graph among its different computing components. This study helped us to identify exactly the number of operations involved in a CG method and devise distributed

Prepared using speauth.cls
DOI: 10.1002/spe

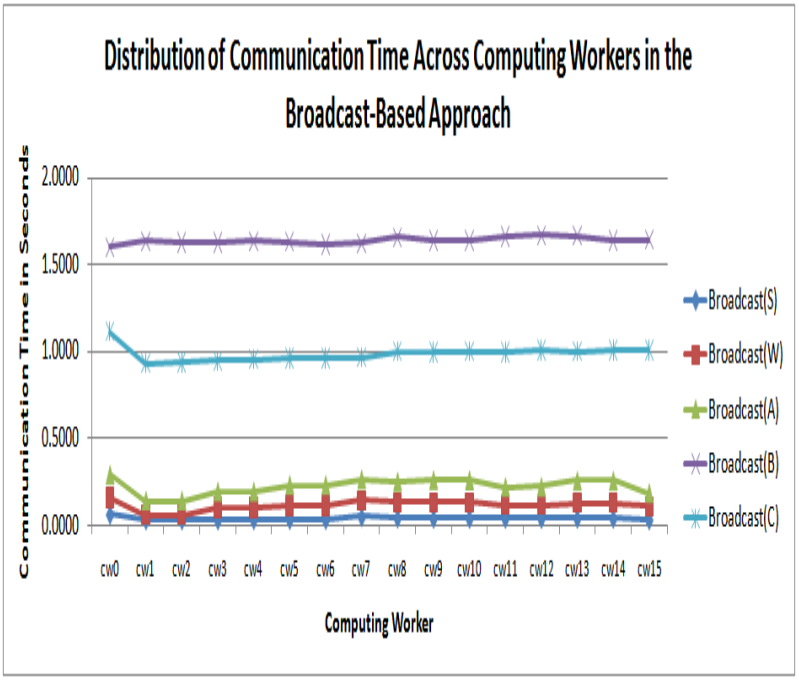

Figure 14. Distribution of communication time across computing workers for broadcast-based approach.

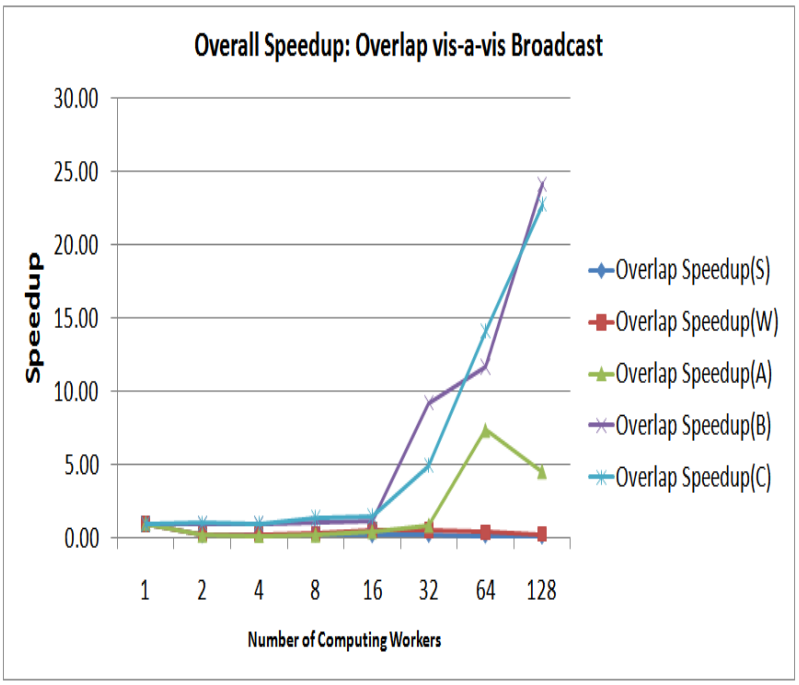

Figure 15. Overall speedup of overlap-based approach vis-a-vis broadcast-based approach.

algorithms with a communication approach to increase the speedup of CG distributed algorithms. We then implemented two distributed algorithms for the CG method on a Cloud Computing infrastructure, one which uses the broadcast-based approach; and one that uses

Prepared using speauth.cls
DOI: 10.1002/spe

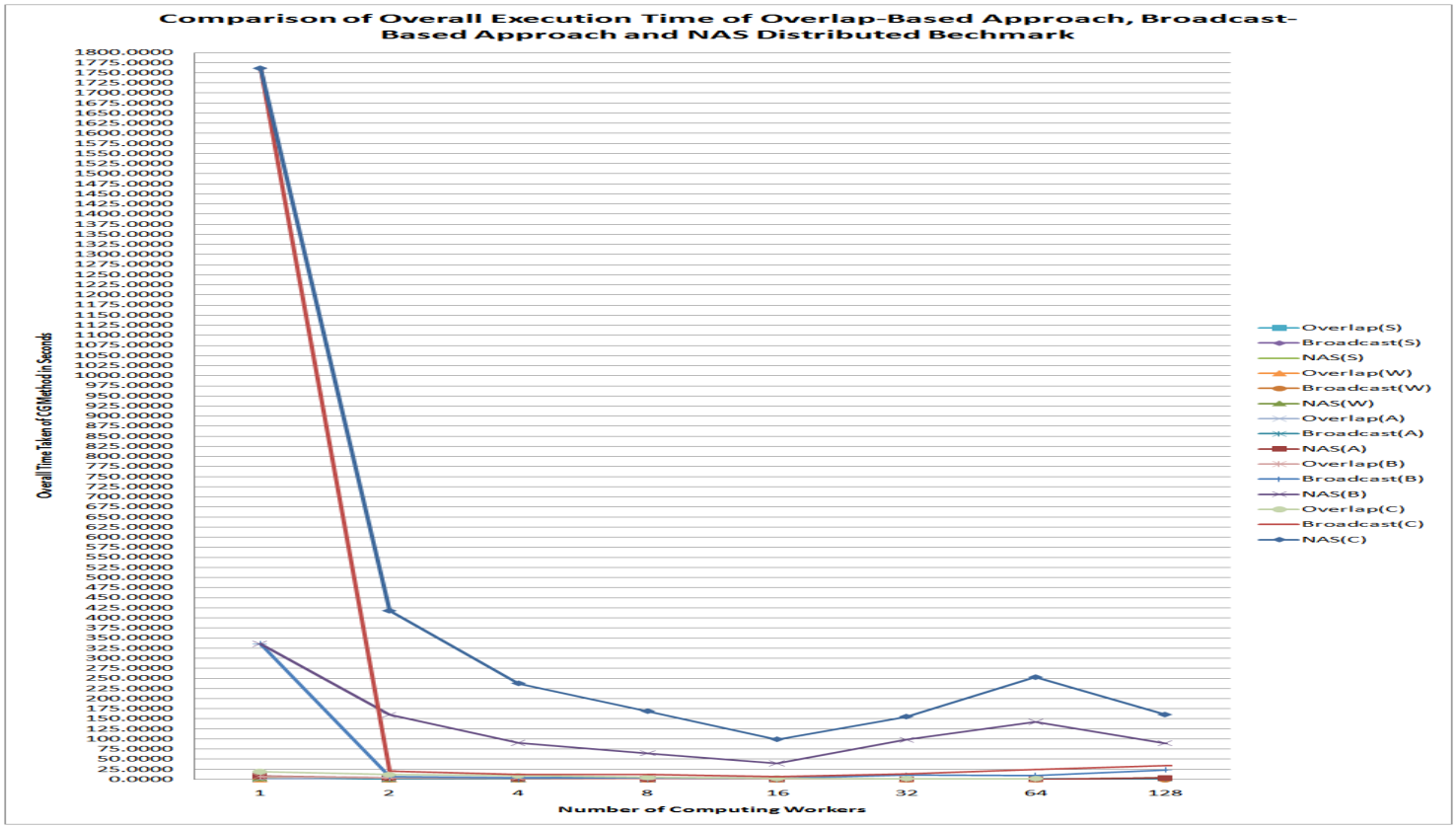

Figure 16. Comparison of performance of overlap-based approach, broadcast-based approach, and NAS distributed benchmark for the CG method.

an overlap-based approach where communication overlaps computation. In the first algorithm, the broadcast-based approach, each computing worker computes its own part of the global computation and broadcast the results to all other computing workers. In the second algorithm, the overlap-based approach algorithm, each computing worker computes part of the resultant vector and sends the vector needed by other computing workers to the neighbor computing worker, in a ring fashion, before continuing the next step of its current computation. The broadcast-based approach gives significant results compared to the sequential approach. By the end of the computation, the vector would have reached every other computing worker. Both algorithm follows a load-balanced distribution approach for the computation. Both distributed algorithms give significant results compared to the non-distributed algorithm in our Cloud environment. The broadcast-based algorithm is 245.4 times faster than the sequential algorithm using 64 computing workers for matrix of size C. This speedup decreases to reach 50.8 using 128 computing workers for the same matrix size, due to the broadcast cost which increases when the number of computing workers involved in the computation of CG method increases. The overlap-based algorithm is 1,157.7 times faster than the sequential algorithm for a matrix of size C and using 128 computing workers.

Our experimental results from distributed algorithms show a fairly load-balanced distribution of workload across available computing workers. This is thanks to the distribution algorithm which distributes to its best the workload equally between the different computing workers. However, the load balancing does not achieve exact distribution. This is due to the fact that discrete work units are supposed to be allocated; hence the workload can be lopsided if the work units are not divisible by the number of available computing workers. Moreover, when the terminal indices of the work units are calculated by our formulas, those terminal indices

DOI: 10.1002/spe

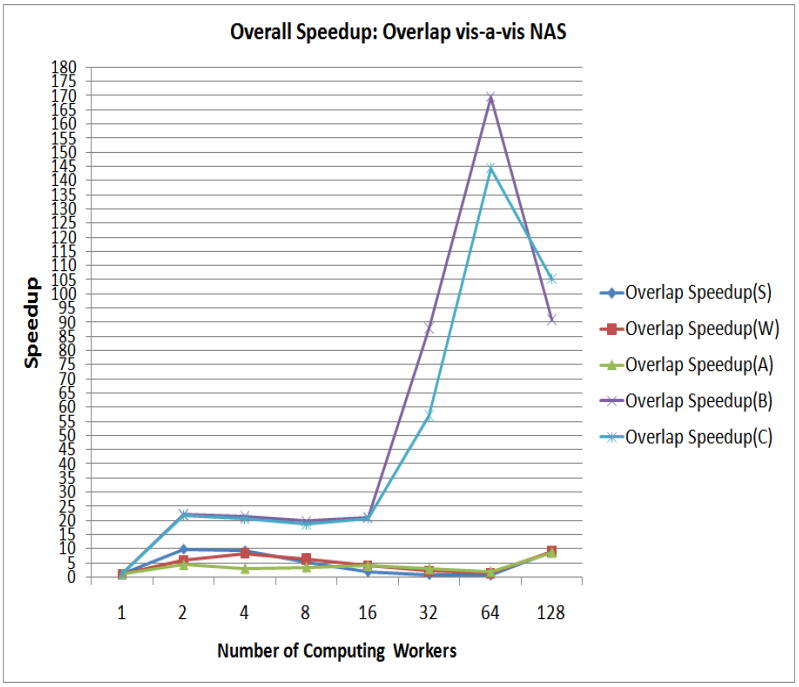

Figure 17. Overall speedup of overlap-based approach vis-a-vis NAS distributed benchmark.

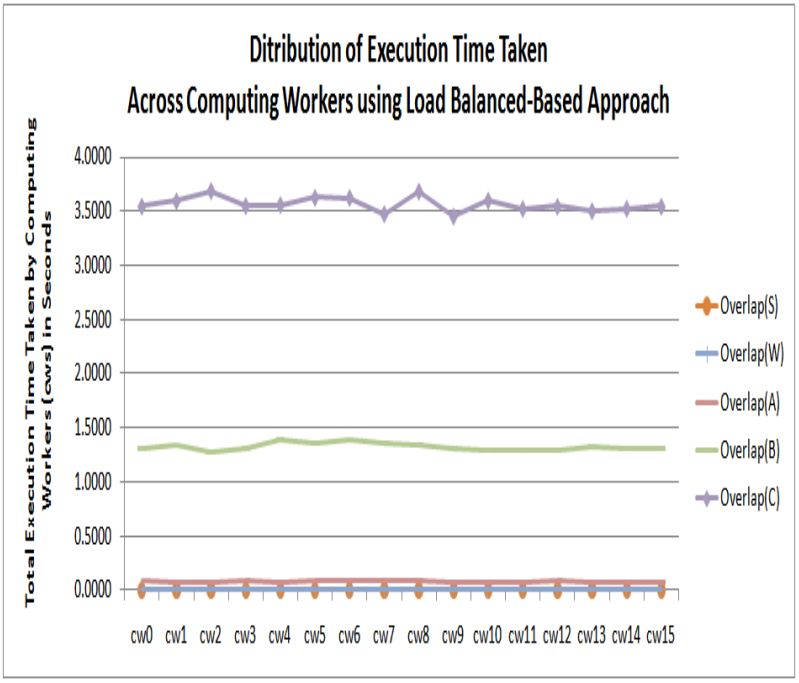

Figure 18. Distribution of execution time across computing workers using our load-balanced-based approach.

include the load associated to them; i.e., for the CG method, a terminal index represents a row index for which all the non-zero elements should be associated.

DOI: 10.1002/spe

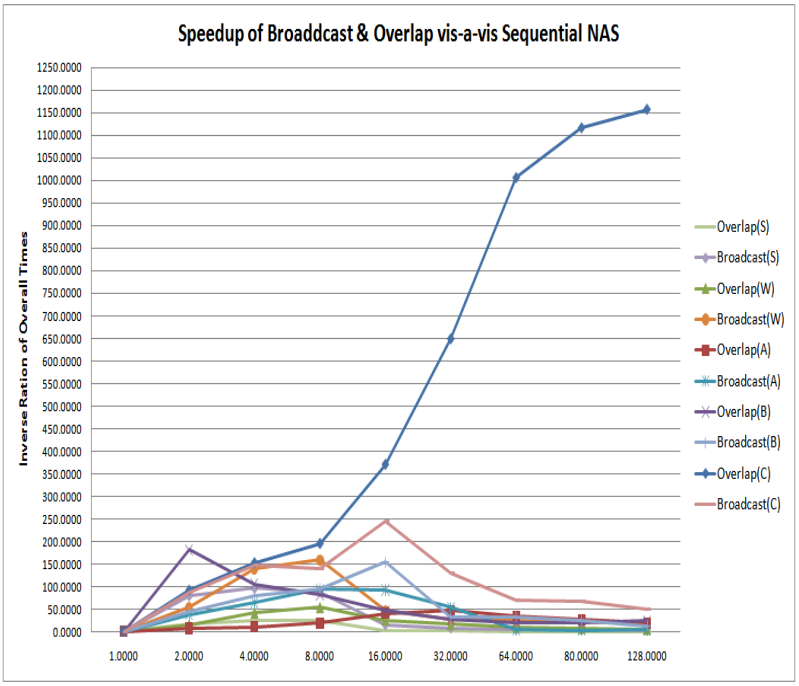

Figure 19. Speedup of the broadcast-based approach and the overlap-based approach vis-a-vis NAS sequential execution with increased number of computing workers and increased matrix sizes.

13. ACKNOWLEDGEMENTS

The authors would like thank the UAE University for supporting this research work. The research is part of project that has been ranked `Highly Competitive` by international peer reviewers following a UAE national competition organized by the UAE National Research Foundation and won by the first author. Special thanks to Dr. Eyad Abed, the Dean of the Faculty of Information Technology of the UAE University for his help and support to the project. The authors would like also to thank the anonymous reviewers for their feedbacks which have contributed to the paper.

REFERENCES

1. R. Buyya, C.S. Yeo, and S. Venugopal, Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities, Keynote Paper, in Proc. 10th IEEE International Conference on High Performance Computing and Communications (HPCC 2008), IEEE CS Press, Sept. 2527, 2008, Dalian, China
2. Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems. Volume 25, Issue 6, June 2009
3. M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia. Above the Clouds: A Berkeley View of Cloud computing. Technical Report No. UCB/EECS- 2009-28, University of California at Berkley, USA, February 10, 2009
4. Christian Vecchiola1, Suraj Pandey1, and Rajkumar Buyya, "High-Performance Cloud Computing: A View of Scientific Applications", Proceedings of the 10th International Symposium on Pervasive Systems, Algorithms and Networks (I-SPAN 2009, IEEE CS Press, USA), Kaohsiung, Taiwan, December 14-16, 2009
5. Peter Mell and Timonthy Grance, "A NIST Definiton of Cloud Computing", National Institute of Standards and Technology, Special Publication 800-145, September 2011, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf
6. Philip Papadopoulos, "Virtual Clusters, Extended Clusters in to Amazon EC2 w/Condor", UC Cloud Summit, UCLA, 19 April 2011
7. Service Level Agreement Zone, "The Service Level Agreement", @Copyright 2007, http://www.sla-zone.co.uk/index.htm

DOI: 10.1002/spe

8. Abou-Kassem, J.H., Farouq Ali, S.M., and Islam, M.R., 2006, "Petroleum Reservoir Simulation: A Basic Approach", Gulf Publishing Company, Houston, TX, USA, 480 pp.
9. Dogru, Ali h., "From Mega-Cell to Giga-Cell Reservoir Simulation", Saudi Aramco Journal of Technology, Spring 2008.
10. Y. Kim, C.C. Swan, and J.-S. Chen, "Performance of parallel conjugate gradient solvers in meshfree analysis of nonlinear continua", Computational Mechanics, 2006.
11. Piero Lanucara, and Sergio Rovida, "Conjugate Gradients Algorithms: An MPI-OpenMP Implementation on Distributed Shared Memory Systems"
12. P. Kloos, P. Blaise, and F. Mathey, "Open MP and MPI Programming with a CG Algorithm"
13. Martin F. Moller, "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning", Neural Networks, Vol. 6, Issue 4, 25 1993, pp. 525-533.
14. Lewis and Van de Geijn, "Distributed memory matrix-vector multiplication and conjugate gradient algorithms"', in Proc. Supercomputing'93, Portland, Oregon, pp. 484-492.
15. Lewis et al, "Matrix-vector multiplication and conjugate gradient algorithms on distributed memory computers", in Proc. Scalable High Performance Computing Conference, 1994, pp. 542-550.
16. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga, "The NAS Parallel Benchmarks", RNR Technical Report RNR-94-007, March 1994.
17. Jonathon Richard Shewchuk, "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain", School of Computer Science, Carnegie Mellon University, Edition 1 1/4
18. Kevin B. Theobald, Gagan Agrawal, Rishi Kumar, Gerd Heber, Guang R. Gao, Paul Stodghill, and Keshav Pingali, "Landing CG on EARTH: A Case Study of Fine-Grained Multithreading on an Evolutionary Path", Proceedings of the ACM/IEEE conference on Supercomputing, pp. 4-4, 2000
19. Kevin B. Theobald, Gagan Agrawal, Gerd Heber, Ruppa K. Thulasiram, and Guang R. Gao, "Developing a Communication Intensive Application on the EARTH Multithreaded Architecture", Euro-Par 2000 parallel processing, LNCS 1900, pp. 625-637, 2000
20. Kevin B. Theobald, Rishi Kumar, Gagan Agrawal, Gerd Heber, Ruppa K. Thulasiram, and Guang R. Gao6, "Implementation and Evaluation of a Communication Intensive Application on the EARTH Multithreaded System", Vol. 14, Issue 3, pp. 183-201, March 2002
21. Fei Chen, Kevin B. Theobald, and Guang R. Gao, "Implementing Parallel Conjugate Gradient on the EARTH Multithreaded Architecture", Sixth IEEE International Conference on Cluster Computing, pp. 459 - 469, 2004
22. Marty R. Field, "Optimizing a Parallel Conjugate Gradient Solver", SIAM Journal on Scientific Computing, Vol. 19, issue 1, pp. 27 - 37, 1998
23. Andrzej Jordan and Robert Piotr Bycul, "The Parallel Algorithm of Conjugate Gradient Method", Proceedings of the NATO Advanced Research Workshop on Advanced Environments, Tools, and Applications for Cluster Computing-Revised Papers, Vol. 2326, pp. 156 - 165, 2001
24. Robert Piotr Bycul, Andrzej Jordan, Marcin Cichomski, "A New Version of Conjugate Gradient Method Parallel Implementation", Proceedings of the International Conference on Parallel Computing in Electrical Engineering, pp. 318, 2002
25. John G. Lewis, Robert A. van de Geijn, "Distributed Memory Matrix-Vector Multiplication and Conjugate Gradient Algorithms", 1993
26. Mark Hoemmen, "Communication-Avoiding Krylov Subspace Mehthods", PhD Disseration, Spring 2010, http://www.cs.berkeley.edu/ mhoemmen/pubs/thesis.pdf
27. John G. Lewis, David G. Payne, "Matrix-Vector Multiplication and Conjugate Gradient Algorithms on Distributed Memory Computers", 1994
28. Dianne P. O'Leary, "Parallel Implementation of the Block Conjugate Gradient Algorithm", Parallel Computing, Vol. 5, pp. 127 - 139, 1987
29. Dianne P. O'Leary, "The Block Conjugate Gradient Algorithm and Related Methods", Linear Algebra and its Applications, Vol. 29, pp. 293 - 322, 1980
30. Haee-Dae Kwon, "Efficient Parallel Implementations of Finite Element Methods Based on the Conjugate Gradient Method", Applied Mathmatics and Computation, Vol. 145, Issue 2-3, 25 December 2003, pp. 869 - 880
31. Leila Ismail, k. Shuaib, "Empirical Study for Communication Cost of Parallel Conjugate Gradient on a Star-Based Network", ams, pp.498-503, In Proceedings of The 2010 Fourth Asia International Conference on Mathematical/Analytical Modeling and Computer Simulation, May 2010
32. Leila Ismail, "Communication Issues in Parallel Conjugate Gradient Method using a Star-Based Network". 2010 International Conference on Computer Applications and Industrial Electronics (ICCAIE 2010),December 2010
33. InfiniBand General Specifications, Vol. 1, Release 1.2.1, January 2008 available at. (http://www.infinibandta.org)
34. William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir and Marc Snir, "MPI: The Complete Reference", Vol. 2, ISBN-10:0-262-57123-4, ISBN-13:978-0-262-57123-4

DOI: 10.1002/spe