HAMA: An Efficient Matrix Computation with the MapReduce Framework

Sangwon Seo[*] Edward J. Yoon[†] Jaehong Kim[‡] Seongwook Jin[§]
KAIST NHN Corp. KAIST KAIST

Jin-Soo Kim[**]
Sungkyunkwan University

Seungryoul Maeng[††]
KAIST

CS/TR-2010-330

July 06, 2010

K A I S T
Department of Computer Science

[*] swseo@camars.kaist.ac.kr
[†] edwardyoon@apache.org
[‡] jaehong@camars.kaist.ac.kr
[§] swjin@camars.kaist.ac.kr
[**] jinsookim@skku.edu
[††] maeng@camars.kaist.ac.kr

HAMA: An Efficient Matrix Computation with the MapReduce Framework

Sangwon Seo
Computer Science Division
KAIST (Korea Advanced Institute of Science and Technology), South Korea
swseo@calab.kaist.ac.kr

Edward J. Yoon
User Service Development Center
NHN Corp., South Korea
edwardyoon@apache.org

Jaehong Kim
Computer Science Division
KAIST (Korea Advanced Institute of Science and Technology), South Korea
jaehong@calab.kaist.ac.kr

Seongwook Jin
Computer Science Division
KAIST (Korea Advanced Institute of Science and Technology), South Korea
swjin@calab.kaist.ac.kr

Jin-Soo Kim
School of Information and Communication
Sungkyunkwan University, South Korea
jinsookim@skku.edu

Seungryoul Maeng
Computer Science Division
KAIST (Korea Advanced Institute of Science and Technology), South Korea
maeng@calab.kaist.ac.kr

***Abstract*—APPLICATION. Various scientific computations have become so complex, and thus computation tools play an important role. In this paper, we explore the state-of-the-art framework providing high-level matrix computation primitives with MapReduce through the case study approach, and demonstrate these primitives with different computation engines to show the performance and scalability. We believe the opportunity for using MapReduce in scientific computation is even more promising than the success to date in the parallel systems literature.**

I. Introduction

As cloud computing environment emerges, Google has introduced the MapReduce framework to accelerate parallel and distributed computing on more than a thousand of inexpensive machines. Google has shown that the MapReduce framework is easy to use and provides massive scalability with extensive fault tolerance [2]. Especially, MapReduce fits well with complex data-intensive computations such as high-dimensional scientific simulation, machine learning, and data mining. Google and Yahoo! are known to operate dedicated clusters for MapReduce applications, each cluster consisting of several thousands of nodes. One of typical MapReduce applications in these companies is to analyze search logs to characterize user tendencies. The success of Google prompted an Apache opensource project called Hadoop [11], which is the clone of the MapReduce framework. Recently, Hadoop grew into an enormous project unifying many Apache subprojects such as HBase [12] and Zookeeper [13].

Massive matrix/graph computations are often used as primary means for many data-intensive scientific applications. For example, such applications as large-scale numerical analysis, data mining, computational physics, and graph rendering frequently require the intensive computation power of matrix inversion. Similarly, graph computations are key primitives for various scientific applications such as machine learning, information retrieval, bioinformatics, and social network analysis.

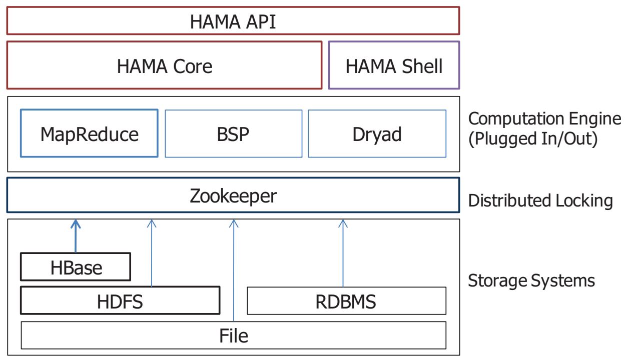


Fig. 1. The overall architecture of HAMA.

HAMA is a distributed framework on Hadoop for massive matrix and graph computations. HAMA aims at a powerful tool for various scientific applications, providing basic primitives for developers and researchers with simple APIs. HAMA is currently being incubated as one of the subprojects of Hadoop by the Apache Software Foundation [10].

Figure 1 illustrates the overall architecture of HAMA. HAMA has a layered architecture consisting of three components: *HAMA Core* for providing many primitives to matrix and graph computations, *HAMA Shell* for interactive user console, and *HAMA API*. The HAMA Core component also determines the appropriate computation engine. At this moment, HAMA supports three computation engines: Hadoop's MapReduce engine, our own BSP (Bulk Synchronous Parallel) [9] engine, and Microsoft's Dryad [3] engine. The Hadoop's MapReduce engine is used for matrix computations, while BSP and Dryad engines are commonly used for graph computations. The main difference between BSP and Dryad is that BSP gives high performance with good data locality, while Dryad provides highly flexible computations with the fine control over the communication graph.

Row	Column Families
matrix_name (alias)	actual_matrix_path
metadata	attribute:created_date
	attribute:owner
	attribute:purpose

TABLE I
HBASE TABLE SCHEMA FOR HAMA.ADMIN.TABLE.

Row	Column Families
matrix_row_index	column_vector
metadata	alias:name
	attribute:columns
	attribute:rows
	attribute:type
	eival:value
	eival:ind
	eivec:value
	cache:value

TABLE II
HBASE TABLE SCHEMA FOR ACTUAL MATRIX.

To manipulate distributed metadata and transaction control in an atomic way, HAMA makes full use of Zookeeper, Hadoop's counterpart for Google Chubby [4]. In addition, HAMA provides flexible data management interface, where the default interface is HBase on top of Hadoop Distributed File System (HDFS).

We summarize the contributions of HAMA as follows.

- **Compatibility**: HAMA can take advantage of all functionalities of Hadoop and its related packages, since HAMA preserves compatibility with the existing Hadoop interfaces.
- **Scalability**: Due to HAMA's compatibility, HAMA can fully utilize large-scale distributed Internet infrastructures and services such as Amazon EC2 without any modification.
- **Flexibility**: To leverage the flexibility needed to support different computation patterns, HAMA provides simple computation engine interface. Any computation engine conforming to this interface can be plugged in and out freely. Currently, three computation engines, namely MapReduce, BSP, and Dryad, are available for use.
- **Applicability**: Primitives offered by HAMA can be applied to various applications that require matrix and graph computations. As a practical example, Me2day [5], a famous social networking service in Korea similar to twitter, is now about to use HAMA to cluster users based on a very large set of data.

Among three different computation engines currently provided by HAMA, this paper focuses on the MapReduce engine mainly used for matrix computations. Specifically, we share our experience of implementing high-dimensional matrix computations with the MapReduce framework and present our preliminary results. We also investigate the scalability of the proposed approach in comparison to MPI.

II. CASE STUDY: PRIMITIVES FOR LINEAR ALGEBRA

In many cases, complex scientific applications require solutions of linear algebra. As a case study, this section describes two basic primitives, *matrix multiplication* and *finding linear solution*, and goes into details of their implementations with the MapReduce framework.

A. Representing matrices on HBase

In order to manipulate matrices on HDFS, we choose HBase as a No-SQL database. The HBase project was originally initiated by Powerset in 2007, modeling after Google's Bigtable [6]. Now, it becomes one of the famous Apache Hadoop subprojects. Unlike traditional RDBMSes, HBase has a column-oriented, semi-structured data structure, which can be distributed over more than 1000 nodes with high scalability.

To represent matrices on HBase, we have designed two structures, a management table and an actual matrix structure. We named the management table as hama.admin.table and the specific matrix structure as hama.matrix_xxx. Table 1 and 2 illustrate the schema of the management table and the actual matrix, respectively. The management table shown in Table 1 consists of three metadata column families, and an "actual_matrix_path" which indicates the specific matrix in Table 2. In particular, "attribute:purpose" specifies whether the matrix is an actual matrix or it is an adjacent matrix representation for a graph. The matrix data structure shown in Table 2 includes necessary metadata for storing column/row size, the type, and eigen pairs, as well as column vectors per row index. Unlike "attribute:purpose" in Table 1, "attribute:type" in Table 2 represents whether the chosen matrix is a *sparse* matrix or a *dense* matrix. An algorithm can be optimized based on the type of a matrix.

Note that this matrix representation is effective in handling temporary matrices when computing a job. This is because the mapper and the reducer can only look into the matrix with alias provided by the management table, while system components internally manipulate temporary matrices.

B. Multiplication of two matrices

We propose two approaches to matrix multiplication: *iterative approach* and *block approach*. The former is suitable for sparse matrices, while the latter is appropriate for dense matrices with low communication overhead. Assume that square matrix A and B are used for multiplication in the following algorithms.

1) Iterative approach: The iterative approach is simple and naive. Initially, each map task receives a row index of B as a *key*, and the column vector of the row as a *value*. Then, it multiplies all columns of i-th row of A with the received column vector. Finally, a reduce task collects the i-th product into the result matrix. The pseudo-code of the iterative approach is illustrated in Algorithm 1.

2) Block approach: To multiply two dense matrices A and B, we should build the "*collectionTable*" in the preprocessing phase of MapReduce. The collectionTable is an 1-D representation, transformed from the original 2-D representation

Algorithm 1 Multiplication with the iterative approach

```
INPUT: $key$, /* the row index of B */
$value$, /* the column vector of the row */
$context$ /* IO interface (HBase) */

void map(ImmutableBytesWritable key,
    Result value, Context context)
{
    double ij-th = currVector.get(key);
    SparseVector mult  /* Multiplication */
      = new SparseVector(value).scale(ij-th);
    context.write(nKey, mult.getEntries());
}

INPUT: $key$, /* key by map task */
$value$, /* value by map task */
$context$ /* IO interface (HBase) */

void reduce(IntWritable key,
    Iterable<MapWritable> values,
    Context context)
{
    SparseVector sum = new SparseVector();
    for (MapWritable value : values) {
        sum.add(new SparseVector(value));
    }
}
```

of two matrices. Each row of the collectionTable has two submatrices of A(i,k) and B(k,j) with the row index of $(n^2 * i) + (n * j) + k$, where n denotes the row size of matrix A and B. We call these submatrices a *block*. Each map task walks only on the collectionTable instead of the original matrices, and thus it significantly reduces required data movement over the network. The following code shows the block algorithm after preprocessing. Each map task receives a blockID as a *key*, and two submatrices of A and B as its *value*, and then multiplies two submatrices, $A[i][j] * B[j][k]$. Afterward, a reduce task computes the summation of blocks, $s[i][k] += multipliedblocks$. The pseudo-code of the block approach is depicted in Algorithm 2.

Algorithm 2 Multiplication with the block approach

```
INPUT: $key$, /* the blockID */
$value$, /* two submatrices of A and B */
$context$ /* IO interface (HBase) */

void map(ImmutableBytesWritable key,
        Result value, Context context)
{
    SubMatrix a = new SubMatrix(value,0);
    SubMatrix b = new SubMatrix(value,1);
    SubMatrix c = a.mult(b); /* In-memory */
    context.write(new BlockID(key.get()),
        new BytesWritable(c.getBytes()));
}

INPUT: $key$, /* key by map task */
$value$, /* value by map task */
$context$ /* IO interface (HBase) */

void reduce(BlockID key,
        Iterable<BytesWritable> values,
        Context context)
{
    SubMatrix s = null;
    for (BytesWritable value : values) {
        SubMatrix b = new SubMatrix(value);
        if (s == null) { s = b; }
        else { s = s.add(b);}
    }
    context.write(...);
}
```

C. Solving linear system with Conjugate Gradient approach

The next case study is to obtain the solution (x) of a linear equation of

$$Ax = b \tag{1}$$

where b is a known vector, and A is a known, square, symmetric, and positive-definite matrix as a pre-requisite. Such a pre-requisite is not commonly found, but it is often used for solving partial differential equations, structural analysis, and circuit analysis. HAMA provides two methods for this problem: Cramer's rule method and Conjugate Gradient (CG) method. The cramer's rule method is used for dense matrices, and the CG method is suitable for sparse matrices [7]. These methods are automatically chosen according to the type of a matrix. In this section, we briefly introduce the idea behind the CG algorithm, and describe how the CG algorithm works well on MapReduce.

CG method is based on the quadratic form as,

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c \tag{2}$$

A is a symmetric matrix as we mentioned before.

$$f'(x) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b = Ax - b \tag{3}$$

If the gradient $f'(x)$ sets to zero, we obtain Equation (1) we want to solve. That is, the solution of Equation (1) is a critical point of Equation (2) under the assumption that A is a symmetric and positive-definite matrix. This is a simple idea behind the CG method.

Fundamentally, the CG method is similar to the Gradient Descent with smoothing and adaptive step size. Like Gradient Descent method, the CG method is able to find the solution with high accuracy through iteratively adjusting the search direction and the step size until the gradient becomes zero. Accordingly, estimating the appropriate search direction and step size is a key to find the solution quickly. For the CG, search direction is obtained from the conjugate direction method, and the step size is simply calculated by the line search method. In other words, the conjugate direction method finds the search direction as a line, and then the line search tries to discover a minimum along the line.

The pseudo-code of the CG method is described in Algorithm 3, where w, d, g, and *alpha* denote weight vector, conjugate direction, gradient, and step size, respectively. In this algorithm, we use Fletcher form [7] in order to compute search direction. Basically, we use matrix multiplication and transpose primitives running in parallel, which are already implemented in HAMA. In this way, we often reuse many primitives to implement a new feature.

Due to CG's iterative dependencies, we devised *nested-map* interface as shown in Algorithm 3. The nested-map interface allows a single map task to iterate recursively until the termination condition is met. Its input is assigned from HBase directly without the shuffling process between mapper and reducer.

In practice, the CG method can be applied to various areas. For instance, in Artificial Intelligence, it is efficiently used to minimize the training error using the quadratic error metrics such as Euclidean function. Particularly, for MLP (Multi Layer Perception) [18], it is often used to optimize the weight vector of relevant networks. For various applications running in an iterative way as well as the CG method, MapReduce is often better with respect to scalability compared to other computation alternatives such as MPI and OpenMP. In the next section, we show the scalability of matrix multiplication and the CG method with MapReduce and MPI as well as their overall performance.

Algorithm 3 Conjugate Gradient method in HAMA

```
/* Invoked once by nested map interface */
void initialize() {
    g = b.add(-1.0, A.mult(x).getRow(0));
    d = g.mult(-1); /* d = -g */
    SparseMatrix q = A.mult(d);
    alpha = g.transpose().mult(d)
            / d.transpose().mult(q);
    x = x.add(d.mult(alpha));
}

/* Using nested-map interface */
 void map(ImmutableBytesWritable key,
        Result value, Context context)
{
    /* For line search */
    g = g.add(-1.0, mult(x).getRow(0));
    alpha_new = g.transpose().mult(d)
                / d.transpose().mult(q);
    /* Find the conjugate direction */
    d = g.mult(-1).add(d.mult(alpha));
    q = A.mult(d);
    alpha =  g.transpose().mult(d)
            / d.transpose().mult(q);
    /* Update x with gradient(alpha) */
    x = x.add(d.mult(alpha));

    /* Termination check method, such that
       length of direction is sufficiently
       small or x is converged into fixed
       value */
    if (checkTermination(d, x.getRow(0)))
    {
       context.write(new BlockID(key.get()),
            new BytesWritable(x.getBytes()));
    }
    context.write(new BlockID(key.get()),
                null);
}
```

III. Evaluations

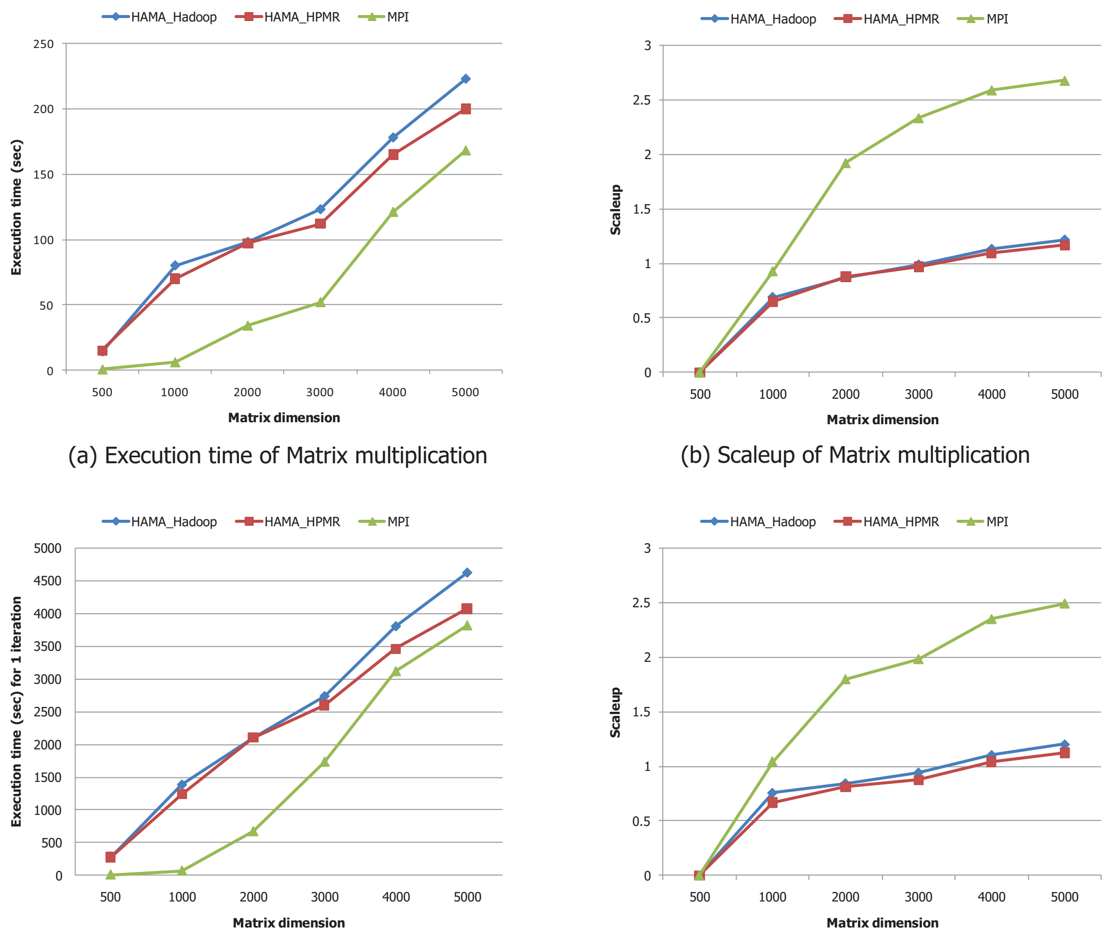
The evaluation of HAMA has been performed on 16-nodes TUSCI (TU Berlin SCI) Cluster [8]. Each node consists of two Intel P4 Xeon processors and 1GB of main memory. In particular, all nodes are connected with high-speed SCI (Scalable Coherent Interface) network interface in a 2D torus topology. Since SCI interconnections provide very low network latency, it can mitigate the data locality problem of Hadoop [1].

In this evaluation, we compare three different versions of matrix multiplication and CG algorithm with 30% sparse matrices, varying the matrix dimension from 500 to 5000. The first version uses the MapReduce engine (Hadoop MapReduce 0.20.0), and the second uses a variant of the MapReduce engine called HPMR [1] which supports prefetching and pre-shuffling. The last version is the MPI implementation for which we used Compaq Extended Math Library (CXML) [18] (version 5.2, previously known as DXML) of Hewlett-Packard (HP), based on LAPACK and BLAS. The first and second version belong to the computation engine family provided by HAMA. For MapReduce and HPMR, we configured that HDFS maintains four replicas for each data block, whose size is 128 MB. The number of mapper and reducer is 16 and 1, respectively. For MPI (version mpich2), we assigned a single task for each node. We left a single processor idle for each node in order to maintain a redundancy for fault tolerance. Figure 2 illustrates the elapsed execution time and the scaleup of performing matrix multiplication with iterative method and CG algorithm, varying the matrix dimension from 500 to 5000. The scaleup means the normalized speedup by the smallest dimension with fixed nodes, such that

$$scaleup(dimension) = log(\frac{T(dimension)}{T(500)})$$

where T denotes the execution time. The scaleup can be viewed as a metric which indicates the scalability. The scaleup is inversely proportional to the scalability.

Figure 2(a), (b) show the average execution time (confidence level = 95%) and its scaleup of matrix multiplication with an iterative method, and Figure 2(c), (d) depict the average execution time (confidence level = 95%) over one iteration and its scaleup of the CG algorithm. As we expected, MPI shows the lowest execution time result among all versions owing to its light-weight characteristics of the library we used. However its scaleup shows more sharp increase and always above than others, especially, when the dimension is larger than 1000. This scalability problem is due to its dramatic increase in synchronous communications required for maintaining iterative dependencies. Although the execution time of MapReduce and HPMR is slower than that of MPI, the performance gap with MPI is gradually shrinking after the dimension becomes larger than 1000 as shown in Figure 2(a), (c). Especially, HPMR, a variant of MapReduce for high performance, outperforms the native MapReduce with



(a) Execution time of Matrix multiplication

(b) Scaleup of Matrix multiplication

(c) Execution time of CG method

(d) Scaleup of CG method

Fig. 2. The comparison of average execution time and scaleup with Multiplication and CG.

the smoothly increased curve.

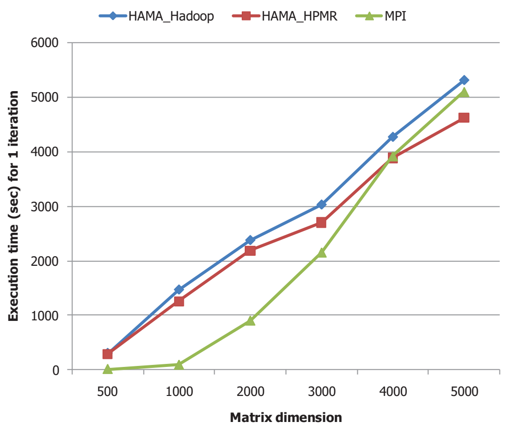
In practice, it is possible that one or more nodes experience a fault or heavy overload during the execution time. For evaluating this situation, we explicitly created a situation of a single node failure and overloaded a node randomly. Figure 3 illustrates the average elapsed execution time and the average delayed time compared to the normal execution time (confidence level = 95%) shown in Figure 2, when a single node is overloaded randomly. Since MPI shows abnormal execution behaviors on a single node failure, we compare the average execution time only when a single node is overloaded.

As depicted in Figure 3(a), HAMA maintains the good performance compared to MPI, and especially HPMR of HAMA excels MPI where the dimension is larger than 4000. In addition, Figure 3(b) shows significant performance degradation of the MPI implementation compared to others. This is because HAMA takes advantage of fault-tolerance facility such as speculative execution of MapReduce which automatically handles the failure without programmer's concerns. Additionally HPMR has an advanced fault-tolerance functionality called D-LATE [1]; when a node crashes or shows slow responses, the MapReduce engine reassigns this straggler task to another fast live node, and resumes the execution. However, MPI continues to execute regardless of the slow execution of node. The Amdahl's Law gives the insight that the overall execution time increases due to the single slow task.

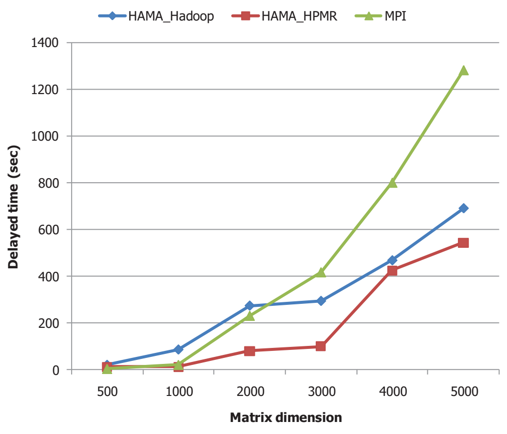
IV. Conclusion

We have proposed the high-dimensional matrix/graph computation framework called HAMA. HAMA provides compatibility with Hadoop, scalability for the large problem size, flexibility with the plug-in engine interface, and applicability for various scientific applications. In particular, through the case study on linear algebra, we have shown that the matrix primitives running on the MapReduce engine is easy to use. Finally, we demonstrated that HAMA shows better scalability than the MPI implementation, maintaining relatively good performance. Especially, in the case where a node is facing the fault, HAMA gives better performance than MPI as the problem size becomes larger.

However, MapReduce is not always appropriate for arbitrary algorithms. That is why we provide the flexible computation engine interface. The graph traversal algorithm such as Breadth First Search (BFS) is a counterexample against the



(a) Average execution time



(b) Average delayed time

Fig. 3. The comparison of average execution time with CG, when a single node is overloaded.

MapReduce algorithm [9]. As the next step, we plan to propose several graph computation primitives with our BSP engine.

References

[1] Sangwon Seo et al., HPMR: Prefetching and Pre-shuffling in Shared MapReduce Computation Environment. In the Proceedings of 11th IEEE International Conference on Cluster Computing, Sep. 2009.
[2] Jeffrey Dean and Sanjay Ghemawat, MapReduce:Simplified Data Processing on Large Clusters. In the Proceedings of the 6th Symposium on Operating Systems Design and Implementation, Dec. 2004.
[3] Michael Isard et al., Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, Mar. 2007.
[4] Mike Burrows, The Chubby lock service for loosely-coupled distributed systems. In the Proceedings of In the Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Nov. 2006.
[5] Me2day: http://me2day.net
[6] Fay Chang, et al., Bigtable: A Distributed Storage System for Structured Data. In the Proceedings of In the Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Nov. 2006.
[7] Jonathan R. Shewchuk, An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. In Technical Report of School of Computer Science, Carnegie Mellon University, Nov. 2006.
[8] http://www.kbs.tu-berlin.de/menue/forschung/projekte/tusci_-_tu_berlin_sci_cluster/parameter/en
[9] http://blog.udanax.org/2009/12/bsp-package-of-hama-on-hadoop-is-now.html
[10] HAMA: http://incubator.apache.org/hama/
[11] Hadoop: http://hadoop.apache.org/
[12] HBase: http://hadoop.apache.org/hbase/
[13] Zookeeper: http://hadoop.apache.org/zookeeper/
[14] Matei Zaharia et al., Improving MapReduce Performance in Heterogeneous Environments. In the Proceedings of the 8th Symposium on Operating Systems Design and Implementation, Dec. 2008.
[15] Guy E. Blelloch, Programming parallel algorithms. In Communications of the ACM, Mar. 1996.
[16] http://www.fhi-berlin.mpg.de/th/locserv/alphas/libs/cxml/dxml.3dxml.html
[17] Luis F. Romero et al., Nesting OpenMP and MPI in the Conjugate Gradient Method for Band Systems. Lecture Notes in Computer Science (LNCS) 3241,181-190, 2004.
[18] Gardner, et al., Artificial neural networks (the multilayer perceptron) - A review of applications in the atmospheric sciences, Atmos. Environ., 32, 2627.2636, 1998.