Applied Mathematical Sciences, Vol. 8, 2014, no. 17, 837 - 849
HIKARI Ltd, www.m-hikari.com
http://dx.doi.org/10.12988/ams.2014.312698

Comparison of Conjugate Gradient Method and Jacobi Method Algorithm on MapReduce Framework

Muhamad Fitra Kacamarga, Bens Pardamean and James Baurley

Information Technology Graduate Program, Bina Nusantara University
Jl. Kebon Jeruk Raya No. 27, Jakarta, Indonesia

Abstract

As the volume of data continues to grow across many areas of science, parallel computing is a solution to the scaling problem many applications face. The goal of a parallel program is to enable the execution of larger problems and to reduce the execution time compared to sequential programs. Among parallel computing frameworks, MapReduce is a framework that enables parallel processing of data on collections of commodity computing nodes without the need to handle the complexities of implementing a parallel program. This paper presents implementations of the parallel Jacobi and Conjugate Gradient methods using MapReduce. A performance analysis shows that MapReduce can speed up the Jacobi method over sequential processing for dense matrices with dimension ≥ 14,000.

Keywords: Parallel Computing, MapReduce, Jacobi Method, Conjugate Gradient, Iterative Method

1. Introduction

A frequently occurring problem in scientific computing is solving a large linear equation Ax=b. Iterative methods are commonly used because they offer faster

computation than direct solvers e.g., Gauss Elimination. This approach is used in many industrial applications such as estimating animal breeding values for animal livestock selection [1], [2], analyzing electronic circuits in electrical engineering, and calculating mass conservation of materials in electrical engineering. All these applications have in common problem: a large dense linear equation to solve.

As the volume of data continues to grow, methods are needed that can scale well and be parallelized. Parallel computing approaches allow flexibility to execute larger applications. Among parallel computing frameworks, MapReduce is a scalable and fault-tolerant data processing tool that processes data in parallel on a collection of commodity computing nodes. This framework has recently been attracting much attention because it is simple, easy to use and can handle very large data [3].

Hadoop is open source software that implements MapReduce. Hadoop's distributed file system (HDFS) uses a storage system that lets the user connect to commodity computers over which data files are distributed [4]. MapReduce is a parallel computing programming model designed for executing smaller subsets of a larger data sets. This provides the scalability that is needed for big data computation [5]. This paper implements the Jacobi and Conjugate Gradient methods using MapReduce. We use Hadoop on the Amazon Elastic MapReduce service and show the speedup compare to sequential processing.

2. MapReduce Framework

Dean and Ghemawat [5] from Google introduce a programming model called MapReduce for processing large data sets using many computers. The MapReduce method was built for abstracting the common operation on large datasets into Map and Reduce steps. The framework was intended to simplify the difficulty of parallel programming. An open source implementation of MapReduce was released called Hadoop. Hadoop's MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output. The programmer may specify a mapping and reducing function. In the mapping phase, MapReduce extracts the input data and applies the mapper function. In the reducing phase, all the output from the mapper is aggregated to produce the final result.

map	`(k1,v1)`	$\rightarrow$ `list(k2,v2)`
reduce	`(k2,list(v2))`	$\rightarrow$ `list(v2)`

Figure 1. MapReduce Framework

The input to the application must be structured as a list of key-value pairs list (k1,v1). This input is broken up and each individual key-value pair, (k1,v1) is processed by calling the map function. Each (k1, v1) pair are converted into list of (k2,v2) by the mapper [6]. The details of this conversion are defined in the MapReduce program.

The output of all the mappers are aggregated into one list of (k2,v2) pairs. All pairs that have the same k2 are grouped together into a new key-value pair, (k2,list(v2)). The reducer processes each one these new key-value pairs individually. MapReduce automatically collects all the list(v2) and writes them to a file. Hadoop lets the programmer perform pre-aggregation on the list of (k2,v2) pairs optionally using a function called combiner, so the communication cost taken to transfer all the intermediate outputs to reducers is minimized [4].

Hadoop provides an API to MapReduce that allows programmers to write map and reduce functions in many languages including Java. Hadoop Streaming uses UNIX standard streams as the interface between Hadoop and the program, so the programmer can use any language that can read standard input and write to standard output to write MapReduce programs.

3. Introduction to the Jacobi Method and Conjugate Gradient Method

The Jacobi method is a stationary iterative algorithm for determining a solution of a linear system [7]. This method makes two assumptions: first, the system $Ax = b$ has a unique solution. And second, the coefficient matrix A has no zeros on its main diagonal. The Jacobi method is initialized to values of an approximate solution. Then the method is performed iteratively to improve the approximated values to a more accurate solution. This method stops when the desired accuracy is reached. The method is guaranteed to converge if the matrix A is strictly diagonally dominant. This method can be expressed in Algorithm 1.1 [8].

Algorithm 1.1 Jacobi method algorithm

```
1: k ← 0
2: while convergence not reached do
3:     for i := 1 → n do
4:         s ← 0
5:         for j := 1 → n do
6:             if j ≠ i then
7:                 s = s + a_ij x_j^(k)
8:             end if
9:         end for
10:        x_i^(k+1) = (b_i − s)/a_ij
11:    end for
12:    check if convergence is reached
13:    k ← k + 1
14: end while
```

The Conjugate Gradient method is one of the Krylov subspace methods designed for solving *symmetric positive definite* (SPD) linear system [9]. The Conjugate Gradient method is able to find the solution by adjusting the search direction using gradients. It avoids repeated searches by modifying the gradient at each step until the gradient becomes zero. The Conjugate Gradient method works from an initial guess x_0 and taking $s_0 = r_0 = b - Ax_0$, until convergence. This method can be expressed by Algorithm 1.2 [10].

Algorithm 1.2 Conjugate gradient method algorithm

$k \leftarrow 0$
$r \leftarrow b - Ax$
$p \leftarrow r$
while $k < k_{max}$ **do**
    $\alpha \leftarrow \frac{r^T r}{p^T Ap}$
    $x \leftarrow x + \alpha p$
    $r_{old} \leftarrow r$
    if k is divisible by 50 **then**
        $r \leftarrow b - Ax$
    else
        $r \leftarrow r - \alpha Ap$
    end if
    $\beta \leftarrow \frac{r^T r}{r_{old}^T r_{old}}$
    $p \leftarrow r + \beta p$
    $k \leftarrow k + 1$
end while

4. Dataset

To make our parallel Jacobi method and Conjugate Gradient implementation as general-purpose as possible, we have implemented it using dense matrices. The datasets were obtained by generating matrices from a MATLAB program. There were 3 sets of files that were used: matrix A as a known, square matrix, matrix B as a known vector, and X with a current solution x. Each set is represented in text.

The matrix A consists of random real numbers from -9 to 9. All columns are separated by delimiter *","*, and *"\n"* denotes a new row. The first value is the number of the row. An example of matrix A can be seen in Figure 2. In this study, Matrix A was generated with dimensions 10,000, 12,000, 14,000, 16,000, 18,000 and 20,000 as shown in Table 1. Matrix A data is strictly diagonally dominant.

Figure 2. Example data of matrix A

Dimension	File Size (KB)
10,000 x 10,000	291,853
12,000 x 12,000	420,259
14,000 x 14,000	571,967
16,000 x 16,000	747,061
18,000 x 18,000	945,488
20,000 x 20,000	1,167,223

Table 1. File size of each matrix A

The Matrix B consists of random real numbers from -9 to 9. This matrix only has 1 row. The following entries are used to represent this matrix in text: *0, random value of row 1, ... , random value of row n*. Matrix B uses delimiter "," to distinguish between rows. The first value of the matrix B is 0 to denote that this line is vector B. An example of matrix B can be seen in Figure 3.

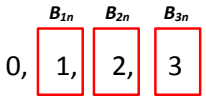

Figure 3. Example of matrix B

The X values file is used to accommodate the current solution x. This value is entered into the text file in the following format: *1 \t 0*. This file uses delimiter *"\t"* to separate the number X and the value. For example, in the first line of Figure 4, *"1"* means X1 and *"0"* means the values of the X1 is 0. This data uses the text file format. In this study, the initial guess value was 0 for all the x values.

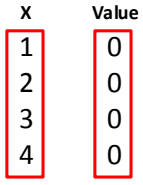

Figure 4. Example of X values file

5. The Jacobi Method MapReduce Implementation

In our MapReduce implementation of the Jacobi method, each iteration is one MapReduce job. Each job consists of a map function and a reduce function. In this application, the Map function is used to parse input data and perform a Jacobi calculation to find the solution x. After that, each result is aggregated by the reduce function and exported into output. The job is described in Table 2.

Job name	**Job description**	**Map function**	**Reduce function**
Calculation x values	Calculate the next approximated solution x	Calculate x values	Aggregate all result values, export into output and increment the number of iterations by one

Table 2. Job description for the Jacobi method MapReduce implementation

At the end of reduce function, the job increments the iteration variable by one and check whether the value is equal the number of iterations *k* specified by the user. If the iteration variable is less than the variable *k*, another job is launched. This algorithm was coded in Java.

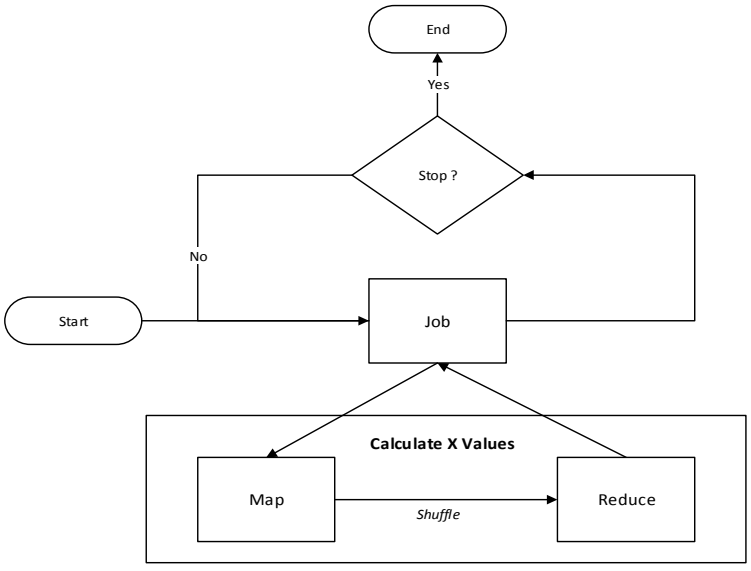

Figure 5. Jacobi method implemented with MapReduce

6. The Conjugate Gradient Method MapReduce

In our implementation of the Conjugate Gradient method using MapReduce, each iteration is five MapReduce jobs. Each job consists of a map function and a reduce function. In this application, the Map function is used to parse input data and perform calculations. Afterwards, each result is aggregated by the reduce function and exported into output. Details of each job are described in Table 3.

Job name	Job description	Map function	Reduce function
Calculation r values	Calculate residual vector	Calculate residual vector value	Aggregated all result values and exported into output
Calculation beta value	Calculate scalar β to be used to determine direction of p	Calculate $r_k^t * r_k$ and $r_{k-1}^t * r_{k-1}$ for each line	Sum $r_k^t * r_k$ and $r_{k-1}^t * r_{k-1}$, calculate $r_k^t * r_k$ / $r_{k-1}^t * r_{k-1}$, and exported into output
Calculation p values	Calculate the next search direction p	Calculate p value	Aggregated all result values and exported into output
Calculation alpha value	Calculate the scalar α	Calculate $r_k^t * r_k$ and $p_k^t * A * p_k$ for each line	Sum $r_k^t * r_k$ and $p_k^t * A * p_k$, calculate $r_k^t * r_k$ / $p_k^t * A * p_k$ and exported into output
Calculation x values	Calculate the next approximation solution x	Calculate x values	Aggregated all result values, exported into output and increment number of iteration by one

Table 3. Each jobs description for the Conjugate Gradient method MapReduce implementation

After finishing the calculation of x values, the iteration variable is incremented and checked against k, a variable specified by the user. If the iteration variable was less than the variable k, another job is started. The algorithm was coded in Java.

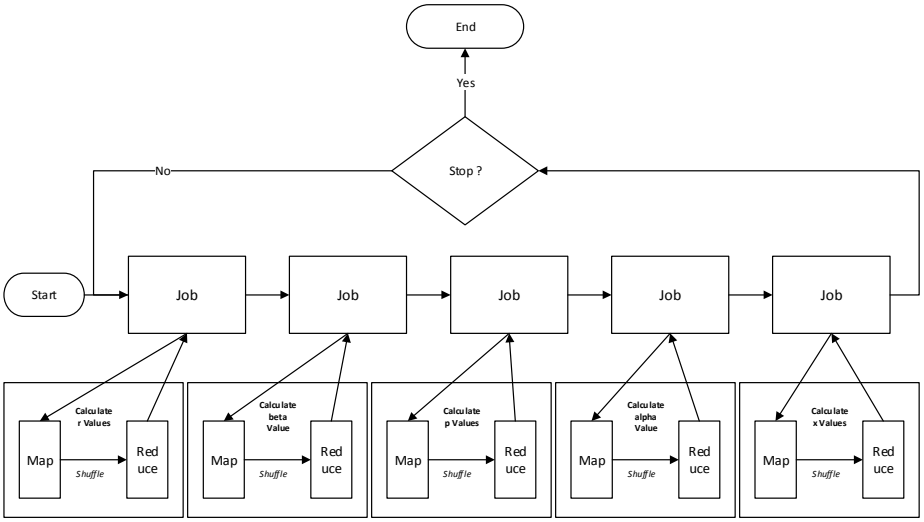

Figure 6. Conjugate Gradient method implemented with MapReduce

7. Experimental Result and Discussion

The evaluation of the parallel Jacobi method and Conjugate Gradient method were implemented on Hadoop MapReduce and executed on 10 High-CPU medium nodes using Amazon Elastic MapReduce. These instances have 1.7 GB of memory, 2 virtual cores with 2.5 EC2 Compute Units each (CU equivalent to 1.0-1.2 GHz processor) per node [11]. This cluster is installed with Hadoop version 1.0.3.

In this evaluation, we compare two different scenarios using the Jacobi method and Conjugate Gradient method for dense matrices, varying the matrix dimension from 10,000 to 20,000. The first version uses the MapReduce framework and the second uses sequential processing on a single High-CPU medium instance. The Jacobi method and Conjugate Gradient method were performed for 31 iterations. For MapReduce, the HDFS block size was configured to 128 MB. The number of map tasks was set using the default *inputFormat* attribute in Hadoop which is to split the total number of bytes in the input file into the correct number of fragments [12]. The number of reduce task is 1. In this study, the median execution time was used to calculate the speed up, calculated as:

$$S_p(C', I) = \frac{median(X)}{median(Y)}$$

where Sp(C',I) is the speedup of the MapReduce approach using data input I; *X* is the median execution time of the sequential approach and *Y* is the median execution time of the MapReduce approach. If the speed up is less than one, there was no speed up, otherwise if *Sp* > *1*, there is speed up. Touati et al., [13] explain the median is a better choice for reporting speedups because the median is less sensitive to *outliers*. They also explain the in most practical cases, the distribution of the executions times are skewed, making the median a better candidate for summarizing execution times. This method is also used by the Standard Performance Evaluation Corporation (SPEC) organization [13].

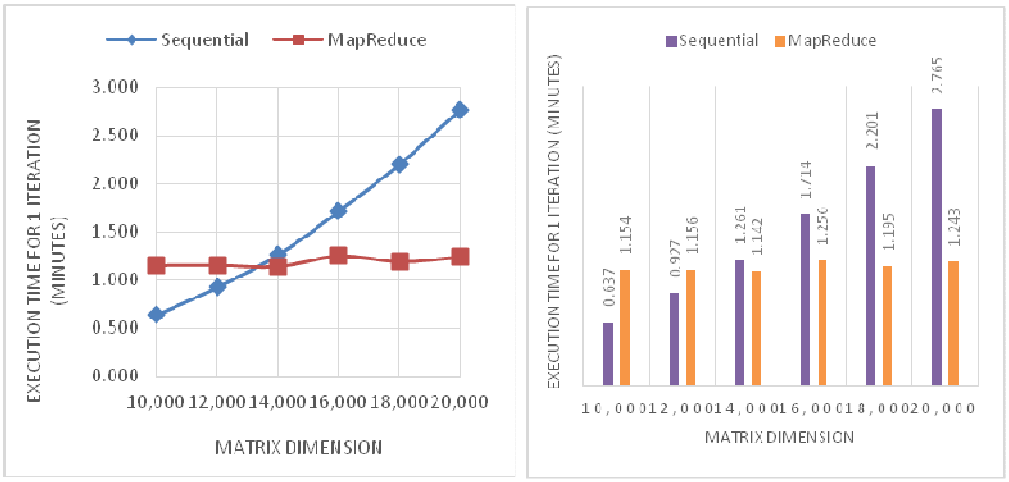

Figure 7. The comparison of median execution time (minutes) for 1 iteration of the Jacobi method

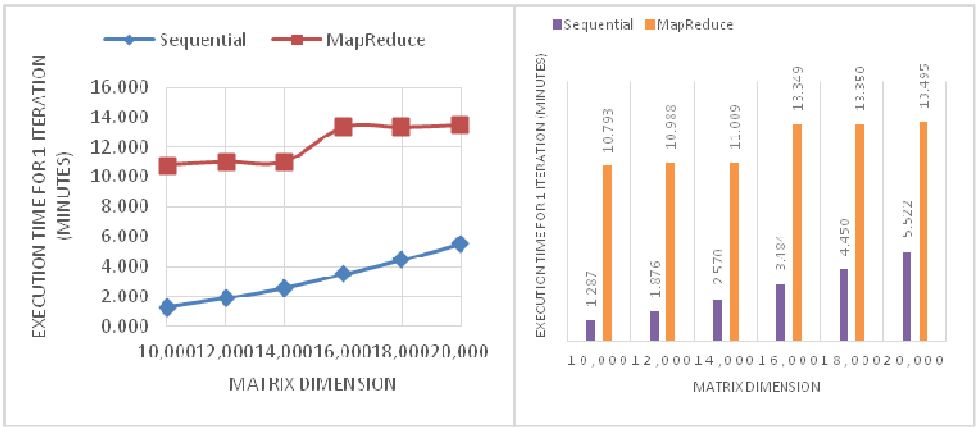

Figure 8. The comparison of median execution time (minutes) for 1 iteration of the Conjugate Gradient method

Matrix Dimension	**Median Speedup**	
	Jacobi Method	Conjugate Gradient Method
10,000	0.553	0.119
12,000	0.802	0.171
14,000	1.104	0.233
16,000	1.365	0.261
18,000	1.842	0.333
20,000	2.225	0.409

Table 4. Speedup gain of the Jacobi method and Conjugate Gradient method in 10 High-CPU medium instances using MapReduce framework

Table 3 shows the speedup of the Jacobi method and Conjugate Gradient over sequential processing. For the Jacobi method, when the input matrix had dimension 10,000 and 12,000, there was no speedup. This is likely due to the overhead associated with Hadoop's MapReduce. When the dimension was equal or larger than 14,000, MapReduce performed better than sequential, which a speedup increasing up to 2.2 for the largest matrix considered.

For the Conjugate Gradient method there was no speedup for any of the matrices considered. It appears that the conjugate gradient method using MapReduce perform worse than sequential for our implementation. It should be noted that the number of map tasks and the number of nodes also affects the execution times. So it is recommended to find an optimum number of nodes and maps. Furthermore, when applications use Hadoop's distributed cache, it is not recommended to assign too many map tasks when using a small number of nodes (e.g., 4 nodes for 200 Map tasks) because it may result in the job running out of memory.

8. Conclusion

We have presented the parallel Jacobi method and conjugate gradient implementation using the MapReduce framework. We used Hadoop on a cluster of 10 High-CPU medium nodes on Amazon Elastic MapReduce. The algorithms performed 31 iterations for increasing matrix dimension from 10,000 to 20,000.

Our experiment showed that the MapReduce implementation of the Jacobi method can give speed up for dense matrices with dimension ≥ 14,000. The conjugate gradient method using MapReduce, however, performed worse than the sequential version. The implementation of the Jacobi method using MapReduce is useful to solve very large scale linear systems. MapReduce is simple, easy to use, fault-tolerant and hides the complexity of parallel programming.

9. Future Work

Future work includes studying the performance of other iterative methods that are easy to parallelize on the MapReduce framework. It would also be interesting to study other implementations of the MapReduce system such as Haloop, Twister and Apache HAMA [8], [9], [10] which are optimize for iterative applications.

References

[1] A. Legarra, "Methods based on SNP estimation (BLUP_SNP and BayesCPi etc.)," 2012. [Online]. Available: http://nce.ads.uga.edu/wiki/lib/exe/fetch.php?media=uga_2_normal_lasso_bayesc.pdf. [Accessed: 24-May-2013].

[2] I. Strandén and M. Lidauer, "Parallel Computing Applied to Breeding Value Estimation in Dairy Cattle," *J. Dairy Sci.*, vol. 84, no. 1, pp. 276–285, Jan. 2001.

[3] K. Lee, Y. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with MapReduce: a survey," *ACM SIGMOD Rec.*, vol. 40, no. 4, pp. 11–20, 2012.

[4] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[5] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] A. Holmes, "Hadoop In Practice." Manning Publications Co, 2012.

[7] M. T. Heath, *Scientific Computing: An Introductory Survey*. The McGraw-Hill Companies Inc., New York, 2002.

[8] R. Barret, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. der Vorst, "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods," *Philadelphia, PA*, 1994.

[9] S. Sickel, M. Yeung, and J. Held, "A Comparison of Some Iterative Methods in Scientific Computing," 2005.

[10] J. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," 1994.

[11] "Amazon EC2 Instances," 2009. [Online]. Available: http://aws.amazon.com/ec2/instance-types/. [Accessed: 20-May-2013].

[12] "HowManyMapsAndReduces," 2009. [Online]. Available: http://wiki.apache.org/hadoop/HowManyMapsAndReduces. [Accessed: 26-Aug-2013].

[13] S.-A.-A. Touati, J. Worms, S. Briais, S. T. Ouati, and J. W. Orms, "The Speedup-Test: a statistical methodology for programme speedup analysis and computation," *Concurr. Comput. Pract. Exp.*, 2012.

[14] Y. Bu, B. Howe, M. Balazinska, and M. D. M. Ernst, "HaLoop: efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1–2, pp. 285–296, Sep. 2010.

[15] T. Gunarathne, B. Zhang, T.-L. Wu, and J. Qiu, "Scalable parallel computing on clouds using Twister4Azure iterative MapReduce," *Futur. Gener. Comput. Syst.*, vol. 29, no. 4, pp. 1035–1048, Jun. 2013.

[16] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "HAMA: An Efficient Matrix Computation with the MapReduce Framework," *2010 IEEE Second Int. Conf. Cloud Comput. Technol. Sci.*, pp. 721–726, Nov. 2010.

Received: December 7, 2013