An Efficient Parallel Algorithm of Modified Jacobi Approach for Sparse Linear System

Bikash Kanti Sarkar[1], Shib Sankat Sana[2], G. Sahoo[3]
[1,3]Department of Information Technology, BIT, Mesra, Ranchi, India
E-mail: bk_sarkarbit@hotmail.com

[2]Department of Mathematics, Bhangar Mahavidyalaya, CU, West Bengal, India

E-mail: shib_sankar@yahoo.com

ABSTRACT

Several parallel approaches have been developed to solve sparse linear system based on well-known memory saving schemes. To solve such a linear system, this article proposes a very simple parallel version of the modified Jacobi iterative method on Distributed Memory Architecture, using the well-known Compressed Sparse Row(CSR) storage format and Recursive Graph Bisection(RGB). The prime contribution of the present investigation is that the individual processors will not update its assigned variables any more, provided the previous iteration achieves smaller than the prescribed accuracy. Consequently, such processors will stop computation as well as communication with other processors to reduce both the computation and the communication time to a great extent. In fact, the use of such local stopping criteria ensures to achieve such overall system performance. The expected benefit of this algorithm is explained through the analytical results.

Keywords : Parallel, Efficient, Communication, Optimization, Distributed Memory, Jacobi.

Date of Submission: February 04, 2011 Date of Acceptance: April 27, 2011

1. INTRODUCTION

Large number of physical problems like Air flow over an aircraft wing, Blood circulation in human body, Water circulation in an ocean, Weather Forecasting, etc, are described by Partial Differential Equations(PDE). These equations, when solved using, Finite Difference Method(FDM), generate sets of linear equations. But the linear systems of the most of the physical problems yield sparse structure after such transformation. Several well-known memory saving schemes are developed to store such kind of system. But, solution of these problems requires huge amount of computations and becomes very difficult by employing conventional computers. So, to solve such problems efficiently in parallel manner is still an attractive issue. Recently, high performance computing has emerged as a key technology into diverse areas especially for the numerical solution of large scale problems. Although, there exist several forms of parallelism[5], but introducing *data parallelism* using *clustering* will be easier.

A matrix is termed sparse, if majority of its entries are zero. As there is no reason to store and operate on huge number of zeros, it is often necessary to modify the existing algorithms to take the advantage of the sparse structure of the matrix. Such matrices can be easily compressed, yielding significant reduction in memory usage. Several sparse matrix formats exist like Compressed Sparse Row(CSR) Storage[1], Jagged Diagonal Format[2], Compressed Diagonal Storage Format[3] and Sparse Block Compressed Row Storage Format[4]. Each format takes advantage of a specific property of the sparse matrix, and therefore achieves different degree of space efficiency. In this work, the CSR storage format (discussed in section[2.1]) is used, as it is rather intuitive, straightforward and more suitable for parallelization.

The solution of a linear system of equations can be accomplished by either of the two numerical methods: *Direct* or *Iterative*. In Direct methods like Gauss Elimination, Gauss Jordon (modification of Gauss Elimination) and Matrix Inversion, the amount of computation is fixed. However , Iterative methods like Jacobi and Gauss Seidel yield values which are found iteratively starting from an *approximation* until the required accuracy is obtained, and hence the amount of computation depends on the accuracy required. Further, the parallelization of iterative approaches becomes easier as compared to the direct approaches. But some iterative methods are suitable on Multiple Instruction Stream and Multiple Data stream(MIMD) Distributed Memory Machine.

*For exampl*e, Jacobi method in comparison to Gauss-Seidel method takes less communication time because all the computations for i-th approximation must be ready before the computation for (i+1)-th approximation starts. In other words, Jacobi iterative approach do not require exchange of the most recent values of the variables, whereas, a subsequent iteration in Gauss-Seidel needs the values of some variables in that iteration too (i.e., causes *intra-iteration dependencies*). Because of this fact, Jacobi approach is preferred for parallelization on Distributed Memory computer as compared to Shared Memory computer.

For partitioning data (involved with linear system) into different processors to maintain *data locality aspect*, there are several algorithms like Multi-grid (Square Mesh Partitioning), Ellpack-Itpack(Row Partition Format), RGB(Recursive Graph Bisection discussed in section[2.2]) etc; and all of which are applied for parallel machines. In this investigation, RGB in comparison to other techniques is preferred as it influences to opt for less communication time, achieving better *static* load balancing of the *sparse graph* among the processors.

Many researchers have concentrated to solve simultaneous system of linear equations sequentially and in parallel, using Jacobi and other approaches [5], [6],[7], [8],[9],[10], [11], [12], [13],[19] [20],[21][22]. Some kinds of tilling techniques[14] are developed for solving set of linear system. Tilling is a compile-time transformation which subdivides the iteration space for a regular computation so that a new tile-based schedule(where each tile is executed atomically) exhibits better data locality. So, tilling provides a method of achieving *inter-iteration* locality. In[15], Communication optimization for irregular scientific computations on Distributed Memory architectures is focused.

Although a number of techniques has been developed till today to solve set of linear systems on Distributed Memory machine trying to reduce communication among processors, but only few of them such as [16][17][21] pay attention to the amount of work done by individual processors.

In particular, in this work, a very simple parallel version based on the *modified* Jacobi iterative method[18] and combining the capabilities of CSR and RGB approaches, is developed on Distributed Memory Architecture to stop unwanted *computation* and *communication* among the processors (in order to reduce both the costs). We compare the *analytical* results of the proposed work with Timing Models [17] and report that the proposed is a better choice.

The present article is organized as follows: section-2 gives theoretical background about CSR, graph partitioning technique, Jacobi method, parallel computers. Section-3 describes the modified version of Jacobi approach. In section-4, the proposed parallel algorithm and its proof of correctness are described. Section-5 shows the analytical results. Finally, section-6 exhibits the future scope of the work.

2. THEORETICAL BACKGROUND

2.1. COMPRESSED SPARSE ROW(CSR) FORMAT

Maximum storage schemes for sparse matrix employ the technique as follows.
Compress all the *non-zero* elements of the sparse matrix (say, A) in a linear array and then use some number of auxiliary arrays to describe the locations of the non-zeros of the original matrix A. The CSR format uses *three* arrays to store an n × n sparse matrix with 'm' nonzero entries.

(i) An m × 1 array, *nonzero*[ ], contains the nonzero elements of the linear system. These are stored in the order of their rows from 0 to (n–1). However, elements of the same row can be stored in any order.

(ii) An m × 1 array, *col_vector* [ ], stores row-wise the respective column number of each nonzero element. Indeed, each column number of a row represents also the variable with non-zero co-efficient in that row.
(iii) An n × 1 array *row_vector*[ ], and the content of *row_vector*[i] points to the first entry of the i^{th} row in nonzero[ ] and col_vector[ ].

One sparse matrix of the form AX=b and this matrix mapped into three arrays are shown in Figs.1 and 2 respectively.

$$
\begin{array}{c|cccccccccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
0 & 1 & & & & & & & & & & & 2 & 3 & & & 5 \\
1 & & 5 & 6 & & & & & 7 & & & & & 8 & & & \\
2 & & 6 & 9 & & & 10 & & & & & & & & 11 & & \\
3 & & & & 12 & & & & & 13 & & 14 & & & & & 15 \\
4 & & & & & 16 & & 17 & & & & & 18 & & & & \\
5 & & & 10 & & & 19 & 20 & & & 21 & & & 22 & & & \\
6 & & & & & 17 & 20 & 23 & & & & & 24 & & & & \\
7 & & 7 & & & & & & 25 & & & & & & 26 & 27 & 28 \\
8 & & & & 13 & & & & & 29 & & & 30 & & & & \\
9 & & & & & & 21 & & & & 31 & & & & 32 & 33 & \\
10 & & & & 14 & & & & & & & 34 & & & & 35 & 36 \\
11 & 2 & & & & 18 & 24 & & & 30 & & & 37 & & & & \\
12 & 3 & 8 & & & & 22 & & & & & & & 38 & & & \\
13 & & & 11 & & & & 26 & & 32 & & & & 39 & & & \\
14 & & & & & & & & 27 & & 32 & 35 & & & & 40 & \\
15 & 4 & & & 15 & & & 28 & & & & 36 & & & & & 41
\end{array}
\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \end{bmatrix}
=
\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \\ b_9 \\ b_{10} \\ b_{11} \\ b_{12} \\ b_{13} \\ b_{14} \\ b_{15} \end{bmatrix}
$$

Figure 1: Sparse Matrix in Equation form AX=b

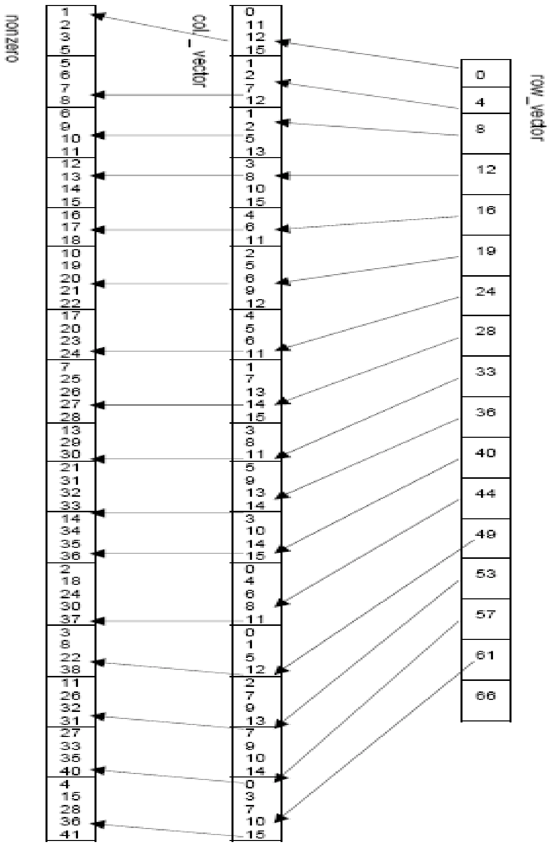

Figure2: Sparse Matrix (shown in Fig.-1) Mapped into three arrays

2.2. RECURSIVE GRAPH BISECTION(RGB) TECHNIQUE

The RGB technique partitions the domain(graph) by recursively subdividing it into two parts at each step. For p = 2^k processors, the domain yields p partitions recursively subdividing k times. This bisection involves *three* major steps. (i) Initially set the level(starting with 0). (ii) Then, find *pseudo-peripheral* node. (iii) Finally, partition the graph recursively.

To determine *pseudo-peripheral* or *peripheral* node of a graph, diameter of the graph(here, graph is represented by matrix) is required to be computed first. The diameter of a graph is defined as follows.

δ (G) = max { d(x,y) | x ε V, y ε V }, where d (x,y) is the distance (shortest path) between any *two* nodes in the graph(G) with vertex set V. Ideally, one of the two nodes in pair (x, y) that achieves the diameter can be used as *starting* node. These two nodes are called as *peripheral nodes*, and are very expensive to determine. A pseudo-peripheral node is often employed to partition the graph. For p =2^2=4, applying the above segment on the graph (represented by the matrix shown in Fig-1), the partitioned graph is shown in Fig-3.

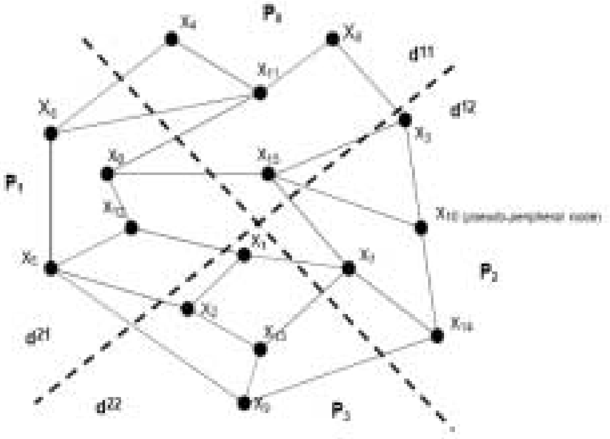

Figure- 3: Partitioned Graph using RBG method(here, d^{11}, d^{12}. d^{22}, d^{23} are the domains, as four processors are used)

2.3. JACOBI METHOD

A set of linear equations is represented as AX=b where A is a matrix of size n x n with co-efficients $a_{i,j}$, X is an nx1 vector variable to be solved and b is an nx1 vector of right side values. Jacobi method is an example of iterative method for solving linear system AX=b, typically generated while working with PDE.

To solve a linear system, AX = b, through this method, the solution vector X must satisfy the equation:

$$X_i = \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} X_j \right) \text{..........} \quad \text{...} \ (1)$$

In fact, to solve the system, one may start the process with an *initial* estimation. However, the Jacobi approach relies upon estimation of every element of vector X to come up with a new value of X. It uses values already computed for each variable X_i during iteration (t+1):

$$X_i(t+1) = \frac{1}{a_{i,i}} \left(b_i - \sum_{j \neq i} a_{i,j} X_j(t) \right)$$

After computing a new estimation, the approach computes the new value of *diff*(difference) based on the change in all elements of X(assume that the initial value of *diff* is 0). Actually, the value of *diff* ensures to stop the approach. Now, *diff* is computed as:

diff = max(abs(X_1(t) - X_1(t+1)), abs((X_n(t)–X_n(t+1)) ...(2)

2.4. PARALLEL COMPUTERS

Parallel computers are those systems that emphasize parallel processing. Parallel computers are generally divided into three architectural configurations:

- *Pipeline computers*: which belong to SISD(Single Instruction Stream and Single Data Stream) model computers and the parallelism achieved through this type of computers is called as t*emporal parallelism* .
- *Array processors* : which belong to SIMD(Single Instruction Stream and Multiple Data Stream) model computers and the parallelism achieved through this model is called *spatial* or *synchronous or data parallelism.* The global CU dispatches the same instruction to each PEs (which are organized by a particular network) and each executes the same instruction on a distinct data set.
- *Multiprocessor systems* : which belong to MIMD(Multiple Instruction Stream and Multiple Data Stream) model computers and the parallelism achieved through this type of computers is called as *control* or *asynchronous parallelism* . This type of system is again classified into two categories :
  (a) *Shared Memory model computers*(or Multi-processors) and (b) *Distributed Memory model*

computers (Message Passing Parallel Computers or Multi-computers).

Message Passing Model Computers are also called Loosely Coupled Computers as the degree of interaction among the processors is not very high. A Message Passing Computer , on the other hand, is programmed using *Send-Receive* primitives. There are several send-receive used in practice.

3. MODIFIED JACOBI APPROACH

From *eqn*(2) of section-2.3, it is clear that in the standard Jacobi iterative method, *diff* is computed as the maximum among all the *absolute* differences of the values of respective variables in the *current* iteration and the *immediate* previous iteration.

As per the standard approach, in spite of achieving the desired accuracy by some of the variables in the current iteration, the same variables are updated again in the *next* iteration to converge the remaining variables. Consequently, it causes unnecessary update of the converged variables. It is also *true* that the variables which are converged to the desired solution in the present iteration, are needed by the present not converged variables.

Thus, the *modified version* stops the updating of the converged variables in the next iteration to reduce execution time but the non-converged variables use the values of the necessary converged variables by updating their current contents with the *diff* value in the current iteration. *For example*, suppose variable X_k is not converged at the present iteration but variable X_m is converged. Then, X_m is simply updated in the successive iterations as follows.

$$X_m = X_m + \text{diff} \ldots\ldots\ldots(3)$$

where *diff* represents the value of *diff*(computed from the rest non-converged variables following *eqn(*2)) at the current iteration. In fact, X_m is here not updated following *equation*-1 (mentioned in section-2.3), i.e., no multiplication, division and more number of additions are performed. However, X_k is computed as per *eq* (1), using the value of X_m (calculated by *eqn*(3)).

However, in the modified approach, it is assumed that each row has some *non-zero co-efficients* excluding the diagonal one. Further, this method requires that the *diagona*l elements are diagonally *dominant*, means that the diagonal element is greater than the sum of the absolute values of the other elements in the given row.

In this article, a simple parallelized version of this *modified* approach, based on CSR storage format and RGB partition technique, is presented (in section-4) on the Multiprocessors to solve a sparse linear system.

4. PROPOSED PARALLEL ALGORITHM

It is well-known that, in sequential iterative approaches, we concentrate on the *approximate* values of the solution vector X and these normally depend on certain degree of accuracy. In particular, the variable *diff* is only used to continue the specified accuracy of the variables. However, instead of *global* diff(which is the maximum among the computed differences of all the variables by following *eqn*(2)), the *local diff* (which is, indeed, the maximum among all the computed differences of the variables assigned to *individual* processor, following the same *eqn*(2)), can guarantee to achieve the same. Of greater interest, the work[18] claims it.

In this section, we present a simple parallel algorithm of the modified Jacobi method on Distributed Memory Architecture to solve sparse linear system. Further, our algorithm is based on Compressed Sparse Row(CSR) storage format and Recursive Graph Bisection(RGB). The goal of this algorithm is to optimize the communication overhead among the processors, reducing computational cost too. However, in the designed algorithm, the *status* variable for every processor fulfills such great role.

***Assumptions*:**

i) All the diagonal elements of the matrix A must be *non-zero* values.

ii) The *diagona*l elements are diagonally *dominant*, means that the diagonal element is greater than the sum of the absolute values of the other elements in the given row.

iii) Processors are represented by unique *ids* such as : 0, 1, 2, 3, etc.

***Brief description of the used variables*:**

(a) To represent solution vector X, one array of structures(records) X[ ] is considered. In fact, each element of X[ ] represents one variable, and consists of *two* fields. In C like language, such structure(record) can be declared as:

struct *variable* { float *value*; int *source* } X [ ].

Clearly, X[i] represents here the i^{th} variable (like X_i). The importance of each of the fields are discussed below.

i) *value* (this field stores the latest updated value of a particular variable by a processor).

ii) *source*(field mainly stores the processor-id of the processor which is updating the particular variable). Thus, it is clear that each variable keeps more information except the value of variable, and each sub-script value of X[ ] represents one variable.

Simultaneously, another array NewX[ ] is essential to store temporarily only the current contents of updated variables(i.e., NewX[index] is used to store temporarily the updated content of X[index].*value* at the current iteration).

(b) The 1-D array *processor_status*[ ] plays here the significant role to maintain status of the participating processors. For example, *processor_status*[0] stores the status of 0^{th} processor and so on. However, its content is either 0(means its work is not over) or 1 (means its work is over). If 'p' number of processors participate in the work, then its size will be 'p'.

(c) *Location*[ ], a simple 1-*D* array, is used to store *var_indices*(i.e., variables) to be by a processor. So, If v number of variables are updated by a processor, then its size will be v (i.e., *Location*[v]).

(d) Three 1-D arrays : *row_vector*[ ], *col_vector*[ ] and *nonzero*[ ] are used to represent CSR storage format of sparse matrix (example shown through Fig-1 for sparse matrix and Fig-2 for its equivalent CSR). Sub-script of *row_vector*[ ] indicates row number.

(e) b[ ], 1-D array, is to represent the source vector, i.e., each location of this array stores the right hand side of a particular equation.

(f) The variable *diff* (local diff) is responsible for checking the desired accuracy of solution of the assigned variables to each processor.

Proposed Algorithm

A brief sketch of the algorithm is outlined below.

Step-1: Processor P_0*(root processor) initializes value* 0(zero) to the *value* part of each element of the *solution vector (X)* as well as the necessary values of the other *fields(members)* of X, and the value of the variable *diff*. It then *broadcasts* all these values to the rest processors participating in this work.

Step-2: *Assign* variables to be updated by each processor into its *local* variable *Location*[ ].
[Here, assigning variables to processors is done, seeing the partitioned graph of the matrix A.]

Step-3: *for* all the working processors P_i, where $0 \leq i \leq p-1$ *do* the following tasks.

// P_i is the processor-id and 'p' is the total number of working processors.

Step-3.1: *for* each variable X_k *assigned* to P_i (K∈ *Location*[ ]), perform the following sub-steps to update the current retrieved variable.

Step-3.1.1*:* First *retrieve* the necessary variables as well as their respective co-efficients simultaneously accessing *col_vector*[ ] and *nonzero*[ ] arrays , following *eq*-1.

Next, *collect* the values of these necessary variables from the respective processors(retrieved through *source* field of X[ ]) by *passing message* (if their work is not over). However, if work of any one is over, then first update the values of the necessary variables computed earlier by that stopped processor, following *eq*-3 (presented in section-3), and use those.

Step-3.1.2: Now, *update* X_k (following *eq*-1) and *store* the value of this variable into an *index* of the *local* array *NewX[ ]*.

// For P_i , step-3.1 ends and step -3.2 starts

Step-3.2: *Update diff(*local diff*)* following the *eqn* (2) [mentioned in section-2.3].

Step-3.3: *Copy* the updated values of the variables(stored in NewX[ ]) into the *value* part of the respective locations of *X[ ].*

Step-3.4: *If diff(*local diff*)* reaches to the desired value (say ε*:* some value is set initially), *then* assign value 1 to its *processor_status*[ ](i.e., *processor_status*[1] = 1) and *send* this value (to all other destination processors to stop further communication with it) and the latest updated values of the assigned variables to the respective processors as well as *root* processors

else the processing goes back to s*tep-3.1* for next *iteration*.

Step-4 **:** If the algorithm terminates, then the *root* processor(P_0) gets the final solutions of the variables.

5. ANALYTICAL RESULTS

Assume that the number of processors is 'p'. Now, if 'k' number of iterations are required to achieve the desired accuracy in *worst case* and total number of nonzero elements in the *nonzero* array is 'm' (m << n), then maximum number operations like *multiplication*, *addition* etc, will be 'mk' in *sequential* approach which can be expressed as *O*(*mk*).

Clearly, the *proposed parallel algorithm* takes O(mk/p) computation time. Although, almost all other existing parallel approaches also demand the same asymptotic running time. But the *status* variable *processor_status* adopted in our algorithm ensures to stop processing of the variables assigned to the individual processors when the desired accuracy computed from the respective assigned variables to them is found. In other words, there is maximum probability to be converged the respective assigned variables earlier(which is, in fact, less in number of iterations). Consequently, the processors which terminate their respective assigned tasks, can be employed to perform the task of another distinct different problem in *multi-processor environment.*

Thus, the present approach reduces *execution time,* since the processors(whose processing is over) stop computation like data access, multiplication, addition, etc. For instance, suppose processor P_0 achieves the desired accuracy over its assigned variables after 8 iterations whereas processor P_1 after 12 iterations, P_2 after 14 iterations etc , then unnecessarily P_0, P_1 need not continue execution up to maximum iteration(14) over their respective assigned variables. Since this drawback is overcome here, so it ultimately saves significant amount of execution time as compared to the existing approaches. Also, the approach claims *less communication cost* between processors than any existing parallel method, since unwanted communication among processors stops.

Now, we consider the Timing Models [17], the total parallel processing time(assuming same processing speed for each processor) can be expressed as

$$T_{par} = T_{master} + T_{worker} + T_{com},$$

where T_{master} defines the master total computation time, T_{worker} as the workers total computation time, and

$T_{com} = T_{c\_master} + T_{c\_worker}$. Here, T_{c_master} includes *broadcast* of global geometry, *distribution* of working tuples and *extraction* of working tuples, whereas T_{c_worker} includes *extraction* of global information, *extraction* of working tuples, *return* of result tuples and intermediate *exchange* of data with the *neighboring processors*.

Clearly, the proposed approach claims better system performance as compared to the mentioned approach , since *master*(root) processor need not collect solutions from any processor to compute the *global diff* and to send the same to the other processors. Consequently, T_{c_worker} does not include here extraction of global information (*diff*) from master processor and *unnecessary extraction* of working tuples, *return* of result tuples and intermediate *exchange* of data with the *neighboring processors* whenever the work of the respective processors is over.

6. CONCLUSION

The article addresses parallelization of a variant of the Jacobi method for linear system solution in distributed memory computers. In this variant, once a variable is detected to be converged it is not communicated afterwards. The status variable and graph partitioning technique used in the proposed algorithm balance the computations and reduce the communication overhead. This novel idea is very much important for the cluster computing because the connection between processors in such environment is often slower. The concept is verified and validated mathematically.

The proposed algorithm can be implemented cluster of personal computers connected by high speed network. The implementation will be using the Message Passing Interface(MPI) library as the parallel programming platform.

REFERENCES

[1]. J.D.Z. Bai, J. Dongarra, A. Ruhe, H. van der Vorst, *Templates for the solution of algebraic eigenvalue problems*: *A Practical guide* .SIAM 2000.

[2]. Y. Saad, Krylov. *Subspace methods on supercomputers*, SIAM Journal on Scientific and Statistical Computing. (1989).

[3]. J. Dongarra, *In Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, SIAM 2000.

[4]. S. Vassiliadis, S. D. Cotofana, P. Stathis, *Block based compression storage expected performance*, In Proceedings of 14th Int'l Conf. on High Performance Computing Systems and Applications (HPCS 2000) (June 2000).

[5]. M. J. Quinn, *Parallel Computing Theory and Practice,* Oregon State University, Second Edition. 2002.

[6]. Van der Vorst, H. A., *Large sparse linear systems on vector and parallel computers*, Parallel Computing, 5, 45–54 (1987).

[7]. Biswa Nath. Datta, *Jacobi Iterative Methods for Large Scale Matrix Problems in Control.* Dept. Mathematical Sciences, pp.2-21 , 2002.

[8]. F. L. Alvarado, A. Pothen, R. Schreiber, *Highly parallel sparse solution, in Technical Report*, RIACS TR 92.11, NASA Ames Research Center, Moffet Field,CA. May 1992.

[9]. C.Aykanat, F. Ozguner, F. Ercal, P. Sadayappan, *Iterative algorithms for solution of large sparse systems of linear equations*, IEEE Transactions on Computers, 37(12):1554–1567 (1988).

[10]. U. Meier, *A parallel partition method for solving sparse systems of linear equations,* Parallel Computing, 2: 33–43 (1985).

[11]. M.T. Heath, E.G.Y. Ng, B.W. Peyton, *Parallel algorithms for sparse linear systems*, SIAM Review, 33 : 420–460 (1991).

[12]. Y. Saad, *Iterative Methods for Sparse Linear Systems,* University of Minnesota, Dept. of Computer Science and Engineering.

[13]. S.G. Petiton, *Massively parallel sparse matrix computation for iterative methods*, in Technical Report. YALEU/DCS/878, Yale University, Department of Computer Science, New Haven, CT, 1991.

[14]. M.M. Strout, Larry Carter, Jeanne Ferrante, , Kreaseck Barbara, *Sparse Tilling For Stationary Iterative Methods*, International Journal of High Performance Computing Applications, vol-18, no-1, pp. 95-113(2004).

[15]. R. Das, M. Uysal, J. Saltz, Y.S.S. Hwang, *Communication Optimizations for irregular scientific computations on Distributed Memory Architectures*, Journal of Parallel and Distributed Computing, 22(3), pp - 462-478 (1994).

[16]. X. Wang, M.M. Chen, X. Wu , and X. An, *A Class of Parallel Algorithms for Solving Large Sparse Linear Systems on Multiprocessors*, in Proceedings of Int. Conf. on High Performance Computing(IEEE-CS), vol-2,1146-1149,(2000)

[17]. Kostas Blathras, Daniel B. Szyld1, and Yuan Shi, *Timing Models and Local Stopping Criteria for Asynchronous Iterative Algorithms*, Journal of Parallel and Distributed Computing, 58, 446-465 (1999)

[18]. B.K.Sarkar, S.S.Sana, and P.K. Mahanti, *A Modified Version of Jacobi Approach*, International Journal of Innovative Computing and Applications, 2, 60-65, 2009.

[19]. S. Kortas and P. Angot, *A practical and portable model for programming for iterative solvers on distributed memory machines*, Parallel Computing, 22 (1996), 487-512.

[20]. A.H. Sameh and D.J. Kuck, *On Stable Parallel Linear System Solvers*, Journal of the Association for Computing Machinery, 25(1), 81-91, 1978.

[21]. R. Chandra and C. Siva Ram Murthy, *A faster algorithm for solving linear algebraic equations on the star graph*, Journal of Parallel and Distributed Computing, 63, 465–480, 2003.

[22]. D. Heller, A survey of parallel algorithms in numerical linear algebra, SIAM Rev. 20 (4) 740–777, 1978.