Journal of Engineering and Applied Sciences 13 (16): 6765-6770, 2018
ISSN: 1816-949X

Parallel Implementation and Comparative Study of Gauss-Jordan and Gauss-Huard Algorithms on a Cluster of Linux Workstations

M.H. Al-Towaiq
Department of Mathematics and Statistics, Jordan University of Science and Technology,
P.O. Box 3030, 22110 Irbid, Jordan

Abstract: In this study, we present parallel implementations of the Gauss-Jordan and Gauss-Huard algorithms with scaled partial pivoting strategy on a cluster of Linux workstations using MPI as a parallel programming environment. We present a comparative study of their performance. The obtained experimental results for the test problems and the analysis of the two algorithms show that the proposed parallel algorithms offer high speed, efficiency and numerical stability. The results also show that Gauss-Huard algorithm performs as good as Gaussian elimination and is more efficient than Gauss-Jordan.

Key words: Linear systems, Gauss-Jordan, Gauss-Huard, algorithms, parallel computing, MPI

INTRODUCTION

In linear algebra the solution of the system of linear Eq. 1:

$$Ax = b \quad (1)$$

where A is an n by n dense matrix, b is a known n-vector and x is n-vector to be determined is probably the most important class of problems. It is needed in many areas of science and engineering. Gaussian Elimination (GE) and related strategies is the most powerful approaches for the solution of (1). It is highly efficient, stable, easy to understand and can be organized such that it performs well on parallel systems (Al-Towaiq, 2013; Duff and Vorst, 1999; Marrero, 2016; Peters and Wilkinson, 1975; Zhu and Sameh, 2007). It $\frac{2}{3}n^3+O(n^2)$ requires floating-point operations.

Two variants of GE reduce matrix A to the identity (or diagonal) matrix: Gauss-Jordan (G-J) which is similar in efficiency to GE is a viable alternative. In Gauss-Jordan elimination, the coefficients above the diagonal as well as those below the diagonal are reduced to zero during the reduction stage. This means that no backward substitution is required. This method requires about $n^3+O(n^2)$ floating-point operations. G-J with scaled partial pivoting strategy is numerically stable (Jia and Jiang, 2015; Sidi, 2008; Trefethen and Schreiber, 1990). The other variant of GE is Guass-Huard (G-H) method (Dekker *et al*., 1997) which reduces matrix A to the identity (or diagonal) matrix. This method "Zeroing" all the elements in the pivot column below and above the current pivot equation. The method is numerically stable as shown by Al-Towaiq (2007) and Duff and Vorst (1999). In both variants, the final step will yield the unique solution, if one exists.

GE and its strategies become the most useful method for linear systems due to the parallelism, stability and memory efficiency benefits it offers (Aizenbud *et al*., 2016; Al-Towaiq and Al-Aamri, 2002; Al-Towaiq *et al*., 2008; Zhang and Dai, 2016). The similarity in appearance between G-J and G-H algorithms inspired us to implement the two algorithms on a cluster of Linux workstations using PVM as a parallel programming environment, verify and validate them and conduct experimental testing. We compare the two algorithms based on the obtained experimental results and their analysis.

Description of the sequential algorithms: We consider Gauss-Jordan and Gauss-Huard with scaled partial pivoting strategy to achieve numerical stable algorithms.

Gauss-Jordan algorithm with scaled partial pivoting: One variant of Gaussian elimination is the Gauss-Jordan strategy by means of forward and backward elimination. At stage k, the variable x_k is eliminated from all equations other than the kth. After n-1 stages, A is reduced to the identity matrix, provided that the operations are simultaneously applied to the right hand side vector, so, the solution vector can be immediately obtained. We describe the algorithm with scaled partial pivoting as follows:

Algorithm 1; Gauss-Jordan algorithm:

Step 1: Initialize the row index vector $l = (1, 2, \ldots, n)^t$

Step 2: Compute the scaled vector $S = (s_1, s_2, \ldots, s_n)^t$ where

$$si = \max_{1 \le j \le n} (|aij|), \ 1 \le i \le n$$

Phase 1; Forward Elimination:
In general, after the $(k-1)^{th}$ stage of elimination we are left with a system of linear equations $A^{(k-1)} x = b^{(k-1)}$ where the augmented matrix is of the form; The Augmented matrix after the $(k-1)^{th}$ stage of G-J:

$$\left[\begin{array}{ccccccccc|c}
1 & a_{12} & \cdots & a_{1,k-1} & a_{1,k} & \cdots & \cdots & a_{1n} & b_1 \\
0 & 1 & \cdots & a^{(1)}_{2,k-1} & a^{(1)}_{2,k} & \cdots & \cdots & a^{(1)}_{2n} & b^{(1)}_2 \\
\vdots & & \ddots & \vdots & \vdots & & & & \vdots \\
 & & & 1 & a^{(k-1)}_{k-1,k} & \cdots & \cdots & a^{(k-1)}_{k-1,n} & b^{(k-1)}_{k-1} \\
 & & & 0 & a^{(k-1)}_{kk} & \cdots & \cdots & a^{(k-1)}_{kn} & b^{(k-1)}_k \\
\vdots & & & \vdots & \vdots & & & \vdots & \vdots \\
 & & & & & & & & \\
0 & \cdots & \cdots & 0 & a^{(k-1)}_{nk} & \cdots & \cdots & a^{(k-1)}_{nn} & b^{(k-1)}_n
\end{array}\right]$$

and the updated index vector is $l = (l_1, l_2, \ldots, l_n)^t$

for i = k to n repeat steps 3 to 6:

Step 3: let i be the first index corresponding to the largest of the ratio

$$\left|a_{l_i,k}\right| / s_{l_i}, \quad k \le i \le n$$

Step 4: interchange l_i and l_k in the index vector l

Step 5: scale row k by dividing the row by the new pivot element $a^{(k-1)}_{l_k,k}$

Step 6: update the elements below the diagonal elements other than the k^{th} equation and the right hand side as follows:

$$a^{(k)}_{ij} = a^{(k-1)}_{ij} - a^{(k-1)}_{ik} a^{(k-1)}_{kj}$$

$$b^{(k)}_i = b^{(k-1)}_i - a^{(k-1)}_{ik} b^{(k-1)}_k$$

Step 7: Phase 2: Backward Elimination

This phase will multiply equation l_k, k = n-1, n-2, ..., 1 by suitable multipliers $a_{l_i k}$ to reduce all the elements above the diagonal of matrix $A^{(n)}$ to zeros. simultaneously update the right hand side vector $b^{(n)}$. The resulting matrix is the identity matrix and the right hand side vector is the solution vector.

Gauss-Huard algorithm with scaled partial pivoting: The other variant of Gaussian elimination is Gauss-Huard (Dekker *et al*., 1997) which reduces the coefficient matrix to the identity (or diagonal) matrix. Using this method with row pivoting strategy based on column interchanges instead of column pivoting is proven to be numerically stable (Al-Towaiq, 2007; Dekker *et al*., 1997). In this paper, we consider Gauss-Huard method with row scaled partial pivoting based on column interchanges to maintain the stability of the algorithm. We describe the algorithm as follows:

Algorithm 2; Gauss-Huard algorithm:
Step 1: Initialize the column index vector $c = (1, 2, ..., n)^t$
Step 2: Compute the scaled vector $S = (s_1, s_2, \ldots, s_n)^t$

where $s_i = \max_{1 \le j \le n} (|a_{ij}|)$, 1• i• n

After the (k-1)th stage of elimination we assume that the upper left hand matrix of order (k-1) is transformed to the identity matrix andall the elements a_{ij}, 1• i• n and k• j• n including the right hand side are modified according to the elimination process as it appears in algorithm2. The Augmented matrix after the (k-1)th stage of G-H.

$$\left[\begin{array}{ccccccccc|c}
1 & 0 & \cdots & 0 & a^{(k-1)}_{1k} & \cdots & \cdots & a^{(k-1)}_{1n} & b^{(k-1)}_1 \\
0 & 1 & \cdots & 0 & a^{(k-1)}_{2,k} & \cdots & \cdots & a^{(k-1)}_{2n} & b^{(k-1)}_2 \\
\vdots & & \ddots & \vdots & \vdots & & & & \vdots \\
 & & & 0 & a^{(k-1)}_{k-1,k} & \cdots & \cdots & a^{(k-1)}_{k-1,n} & b^{(k-1)}_{k-1} \\
0 & 0 & \cdots & 1 & a^{(k-1)}_{k-1,k} & \cdots & \cdots & a^{(k-1)}_{k-1,n} & b^{(k-1)}_{k-1} \\
a_{k1} & a_{k2} & \cdots & a_{k,k-1} & a_{kk} & \cdots & \cdots & a_{kn} & b_k \\
a_{k+1,1} & a_{k+1,2} & \cdots & a_{k+1,k-1} & a_{k+1,k} & \cdots & \cdots & a_{k+1,n} & b_{k+1} \\
\vdots & & & \vdots & \vdots & & & \vdots & \vdots \\
 & & & & & & & & \\
a_{n1} & a_{n2} & \cdots & a_{n,k-1} & a_{nk} & \cdots & \cdots & a_{nn} & b_n
\end{array}\right]$$

Step 3: Eliminate the elements a_{kj}, j = 1, ..., k-1 by the rows of the identity matrix

Simultaneously, the elements a_{kj}, j = k, ..., n and b_k are modified as follows:

```
for i = 1 to k-1  do
    for j = k  to n  do
        a_kj = a_kj - a_ki (k-1) a_ij
    end do
end do

for i = 1 to  k-1  do
        b_k = b_k - a_ki b_i^(k-1)
end do
```

Step 4: let j be the first index corresponding to the largest of the ratios $|a_{kj}|/s_k$, k• j• n
Step 5: interchange columns l_j and l_k in the index vector l, Simultaneously update the column index vector
Step 6: scale row k by dividing row k by the new pivot element a_{kj}
Step 7: perform column elimination by eliminating elements a_{ik}., 1• i• k-1 in column k by suitable multipliers of equation k.
Step 8: repeat steps 3-8

The resulting matrix is an n by n identity matrix and the right hand side vector is replaced by a permutation of the solution vector.

MATERIALS AND METHODS

Parallel implementation: In this study, we propose a parallel implementation of Gauss-Jordan and Gauss-Huard with scaled partial pivoting for solving the dense linear system of Eq. 1. The performance of the two algorithms is evaluated on a cluster of Linux workstations using PVM. We start this section by mapping the elements of matrix A onto the processors. We then present the parallel implementation and comparative study of the two algorithms.

Mapping the matrix elements onto the processors: Solving a linear system Ax = b using G-J or G-H requires to distribute the n-by-n matrix *A* over a set of p processors $p_0, \ldots, p_{p-1}$. The effective mapping of matrix elements to processors is the key factor to efficient implementation of the algorithms in parallel. There are different types of matrix distribution techniques, namely column blocked layout, row blocked layout, column cyclic layout, row cyclic layout, column block cyclic layout andcolumn and row (or 2d) block cyclic layout (Hoffmann, 1998; Kahou *et al*., 2008; Lastovetsky and Reddy, 2007). The main issues of choosing a mapping are the load balancing and communication time. In this study, all the parallel complexities, both in computation and communication, will be analyzed based on the column cyclic layout for G-J algorithm and row cyclic layout for G-J algorithm. column (row) model is a simple and practical model for parallel computation. We select to use column (row) cyclic mapping as it has offered good performance in previous studies of GESPP (Al-Towaiq, 2013).

We consider p processors numbered from 0 to p-1 and n matrix columns (or rows) numbered from 0 to n-1. The Column Cyclic Layout assigns column i to processor i mod p. The Row Cyclic Layout is the transpose of the Column Cyclic Layout. We found out that the load balance is good but it takes more time for communication time (Al-Towaiq, 2013).

Parallel algorithms

Guass-Jordan algorithm: We now define the processes (tasks) that are required for the parallel algorithm of G-J with scaled partial pivoting. We choose to employ two types:

Algorithm 3; A master process:

The following steps describe the parent operations in G-J

Step 1: allocate the processors ids, then store them in the vector tids
Step 2: broadcast vector tids to all processors
Step 3: receive the largest element of sub row i from each processor then compute the largest element of row i then store it in the scaled vector as S[i]
Step 4: broadcast S to all processors
Step 5: receive the index vector l from p_0
Step 6: receive the sub-solutions from processors
Step 7: combine the sub-solutions obtained by the processors into the final solution

The worker (slave) process: The worker processors together andconcurrently, perform the floating-point operations required for the algorithm. The following algorithm describes the operations performed by each worker process p_i:

Algorithm 4; Worker (slave) process:

Input: columns $C_i, C_{i+p}, \ldots, C_{i+(n/p-1)p}$ of matrix A and $b_i, b_{i+p}, \ldots, b_{I+(n/p-1)p}$ of vector b
Output: sub-solution $b_i, b_{i+p}, \ldots, b_{i+(n/p-1)p}$
Process: Step 1: receive the id of processor i from master.
Step 2: find the index C of the id in vector tids
Step 3: read from a data file the columns of A and the elements of b that correspond to p_i.
Step 4: search for the largest (maximum) element in each column as follows:
-for (j = 0, j< n-1, j = j+1) do
$l_{[j]} = j$
find maximum element for row j
send the largest element of row j to master
end do
Step 5: receive the scaled vector S from master
Step 6: reduce A to the identity matrix by performing the following steps
for (k = 0, k<n-1, k = k+1)
determine the index of the processor holding the current pivot
ck = k mod p and its corresponding index in this processor (rowi = k/p)
if (ck = c) then
find the pivot element for the k^{th} column that is $|A[l[i]][R_{ik}]|/S[l[i]]$ is the largest for k• i<n
-store the index i in j
-interchange rows l[i] and l[k]
-broadcast the index vector l
-for (i=k+1, i< n, i++)
A[l[I]][j/p] = A[l[i]][j/p]-value[l[i]]*A[l[k]][i/p,]
k• i, j• n and j = c, p+c, …, (n/p-1)p+c
b[l[i]/p] = b[l[i]/p]- value[l[i]]*b_k
end for i
end for k

Step 7: divide every element in row i, i• 0<n by the pivot element A[l[i]][i] as follows
-if (C = i mod p) then broadcast the scaled element A[l[i]][i] else receive the scaled element
-divide elements of row i by the scaled element.
-If (C = l[k]] mod p) then divide b[l[i]] by the scaled element
Step 8: reduce all the elements above the diagonal to zeros as follows:
-for (k = n-2, k• 0, k--) do
- if (C = (k+1) mod p) then
broadcast column (k+1) mod p as value
else receive value
- if (C = l[k+1]] mod p) then
Broadcast $b_{[[k+1/p]]}$ as b_k
else receive b_k
- update the matrix elements and the right hand side as follows
for (i = k, i • 0,i--)
A[l[i]][j/p] = A[l[i]][j/p] -value[l[i]]*A[l[k+1]][/pj]
(n-1)-(p-(C+1) • j• i
b[l[i]/p] = b[l[i]/p]- value[l[i]]*bk
end for i
end for k
Step 9: if (C = 0) send the latest l
Step 10: send sub-solution to master
Step 11: exit PVM

Guass-Huard algorithm

The master process: The same as in Gauss-Jordan but we use the rows instead of the columns.

The worker (slave) process: The worker processors together and concurrently, perform the floating-point operations required for the algorithm. The following algorithm describes the operations performed by each worker process p_i:

Algorithm 5; The worker (slave) process:

Input: Rows R_i, R_{i+p}, …, $R_{(n/p-1)p+i}$ of matrix A, and
b_i, b_{i+p}, …, $b_{(n/p-1)p+i}$ from vector b
Output: sub-solution b_i, b_{i+p}, …, $b_{(n/p-1)p+i}$
Process: Step 1: receive the id of processor i from master
Step 2: find the index C of the id in vector tids
Step 3: read rows of A and the elements of b that correspond to pi
Step 4: initialize index vector l, l[i] = i, 0• i<n
Step5: perform the elimination operations as follows
-for (k = 0, k• n, k = k+1) do
-determine which processor hold row k and its corresponding index in this processor (Ri = k/p)
-if (C = k mod p)
-Find the scaled element in row k and save its column index in j
-interchange rows l[k] with l[j]
-divide each element in row Ri by the scaled element
-divide b[Ri] (that corresponding to row Ri) by the scaled element
-Broadcast row RI as vector value ,b[Ri] as b_k and l
-else receive value, b_k and l
-reduce all the elements that are above A[Ri][l[k]] to zeros as follows
-for(j = C, j<k,j = j+p) do
-multiplier = A[j/p][l[k]]
A[j/p][l[k]] = 0
-For(m = k+1,m<n,m = m+1) do
A [j/p][l[h]] = A[j/p][l[h]]-multiplier*value [l[m]]
b[j/p] = b[j/p]-multiplier*bk
end for j
-reduce the elements in row k (from A[k+1][0] to A[k+1][k]) to zeros as follows
-row_j, j• k and its corresponding b are sent to $p_{((k+1) \bmod p)}$ as value and b_k respectively
-if(C = (k+1) mod p) then
for (j = 0, j• k, j = j+1) do
receive value and b_k from $p_{j \bmod p}$
multiplier = A [(k+1)/p][l[j]]
for i = k+1 to n-1 do
A[k+1][l[i]] = A[k+1][l[i]]-multiplier *value[l[i]]
end for i
b[(k+1)/p] = b[(k+1)/p]-multiplier*bk
end for j
end for k
Step 6: if (C = 0) send the latest l
Step 7: send sub-solution to master
Step 8:exit MPI

RESULTS AND DISCUSSION

Performance evaluation: In this study, time models for the proposed parallel algorithms are presented. Let t_p be the parallel execution time, t_p is composed of two parts, the computational time t_{comp} and the communication time t_{comm}.

$$t_p = t_{comp} + t_{comm}$$

We calculate the computational time with respect to the number of long operations (multiplications and divisions). For the communication time we use the commonly used model:

$$t_{comm} = t_{startup} + a.t_{data}$$

where $t_{startup}$ is startup time (message latency). It includes the time to pack a message at the source and unpack the message at the destination. The startup time is assumed to be constant. The parameter • is the message size and t_{data} is the transmission time per data item.

We have experimentally determined the startup time to be in the range of $32*10^{-6}$ sec and the transmission time in the range of $3000*10^{-6}$ sec. We assume that the time for sending and the time for broadcasting are the same.

Algorithm 6; Gauss-Jordan algorithm:

We divide the algorithm into the following three main tasks

T1: finding the scaled vector
T2: forward elimination
T3: backward elimination

In T1 takes O(n) computational time to find maximums. However, there is a communication time where each processor sends the maximum element in each row to master. Hence, the total communication time here is $n*(t_{startup} +1*t_{data})$ that is O(n).

T2 is divided into the following subtasks

Computational time
1. finding the pivot row, takes n*(n-k) floating-point operations.
2. dividing the pivot row by the pivot element, takes n*(n-(k+1))
3. forward elimination, takes n*[(n-(k+1))*(n/p+1)]

Communication time
1. broadcasting the index vector l, n times, takes $n*(t_{start\ up} + n*t_{data})$
2. broadcasting row Ri, n times, takes $n*(t_{startup}+n*t_{data})$
3. broadcasting b_k, n times, costs $n*(t_{startup}+1*t_{data})$

Hence, the total number of floating-point operations for T2 is $n*[(3n-3k-2)+n/p*(n-k-1)]$ that is $O(n^3)$ and the communication time is $n*(3\ t_{start\ up}+(2n+1)*t_{data})$ that is $O(n^2)$

T3 is divided into two main subtasks

1. dividing each row by the pivot element which costs about n/p floating-point operations and $n*(t_{startup}+1*t_{data})$ communication time
2. reducing all the elements above the diagonal to zeros. This costs (n-2)*(k*n/p+k*1) floating-point operations that is $O(n^2/p)$. In this task n-2 messages (column) will be sent and this costs $n*(t_{start\ up}+n*t_{data})$

Thus, T3 costs $k*n/p(n-2)-k(n+2)$ that is $O(n^2/p)$ floating-point operations While it takes $n*(2t_{start\ up}+(n+1)*t_{data})$ that is $O(n^2)$ communication time

Overall, the total time t_{GJ} for this algorithm is

$$t_{GJ} = \sum_{i=1}^{3} T_i(\text{computational\_time}) + \sum_{i=1}^{3} T_i(\text{computational\_time}) \quad (2)$$

That is the total cost is approximately $O(n^3)+O(n^2/p)$ computational time and $O(2.n^2)$ communication time

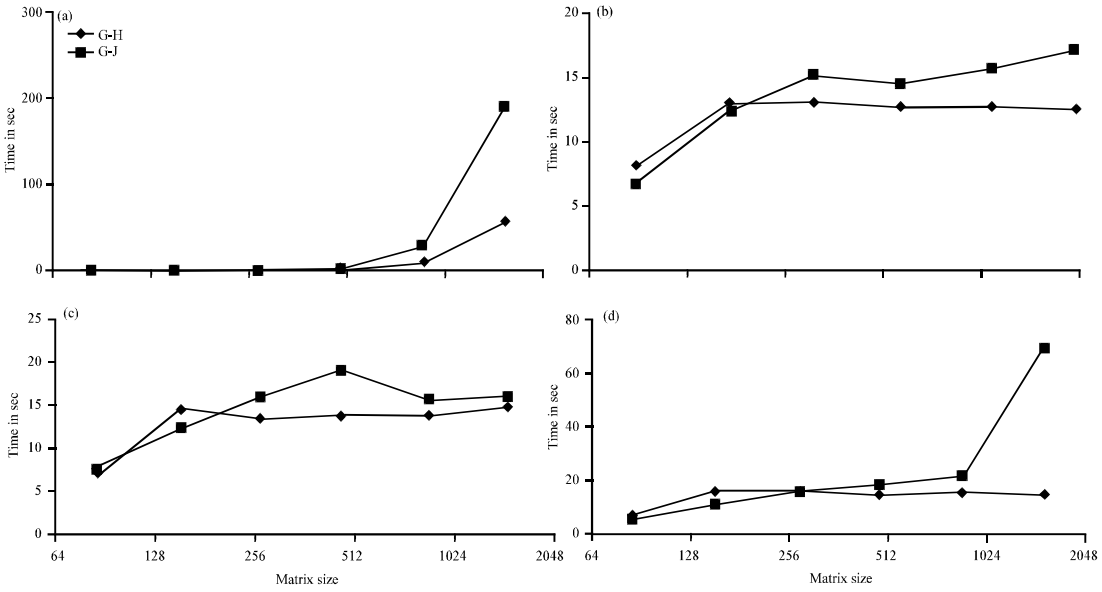

Fig. 1: Algorithm for a system of order: a) One processor; b) Two processor; c) Four processor and d) Eight processor

Algorithm 7; Gauss Haurd's algorithm:
We divide this algorithm into the following three main tasks

T1: finding and scaling the pivot column
T2: reducing all elements above the diagonal to zeros
T3: reducing elements that are in the left of the diagonal element to zeros

In T1 no communication time is needed. But it costs only n*(n-k+1) floating-point operations

In T2 , the elements of the pivot rows and b_k are broadcasted n times, so it costs $n*(t_{startup}+n*t_{data})$ and $n*(t_{start\ up}+1*t_{data})$ respectively. Also, there are about n^3/p floating-point operations to reduce all elements that are above the diagonal to zeros

There are 2n*(n/p) messages (half to send b_k and the other half to send the rows) that must be sent. This means T3 cost $n*(n/p)*(2*t_{startup}+(1+n)*t_{data})$ communication time. There are n*(n-(k+1)) floating-point operations are needed to reduce the left diagonal elements to zeros

Overall, the total time t_{GH} for this algorithm will be

$$t_{GH} = \sum_{i=1}^{3} T_i(\text{computational\_time}) + \sum_{i=1}^{3} T_i(\text{computational\_time}) \quad (3)$$

That is the total cost is approximately $O(n^3/p)+O(2.n^2)$ computational time and $O(n^3/p)+O(n^2)$ communication time

From the above, we conclude that Gauss-Huard algorithm has less computation time but more communication time than Gauss-Jordan algorithm.

The two algorithms are implemented using the academic cluster built in the Department of Computer Science at Jordan University of Science and Technology. This cluster contains 1 management node and 18 Linux (Kernel 2.4.20.8 RedHat 9) workstations connected as a star network, each of which has a single IBM Pentium 4 with 2.4 GHz, 512 Cache, 512 MBs of memory and 40 GBs disk space. These hosts are connected together by fast ethernet, 1 GB switch and 1 optical interconnection switch. We use the Message Passing Interface (MPI) with the MPICH version 1.5.2 as a message passing library throughout the implementations. The barrier synchronization and blocking point-to-point communication are used. The graphs reported in the figures represent the average speedup and efficiency over many runs of the algorithms.

In the implementation process we used the same matrix A, vector b and column/row cyclic layout in both algorithms. Without loss of generality we choose the order of the systems as a power of 2 such as 64, 128, 256, 512, 1024 and 2048. Timing measurements for the execution time were conduced. The obtained results are shown in the following four figures. These results suggest the following.

The required execution time increases as the order of the matrix increases in both algorithms. Gauss-Huard algorithm takes less execution time than Gauss-Jordan algorithm in all cases. As matrix size increases the number of exchange messages in both algorithms increases. Beyond a certain limit this increase causes communication congestion resulting in high communication delays (Fig. 1).

The speedup and efficiency ratios relative to the sequential execution time for the two algorithms for a system of order 2048 on 8 processors are shown in Fig. 2. The speedup decreases as number of processors increases on Gauss-Jordan algorithm but not too many changes in the speedup ratios on Gauss-Huard algorithm. The speedup ratios are comparable beyond 8 processors.

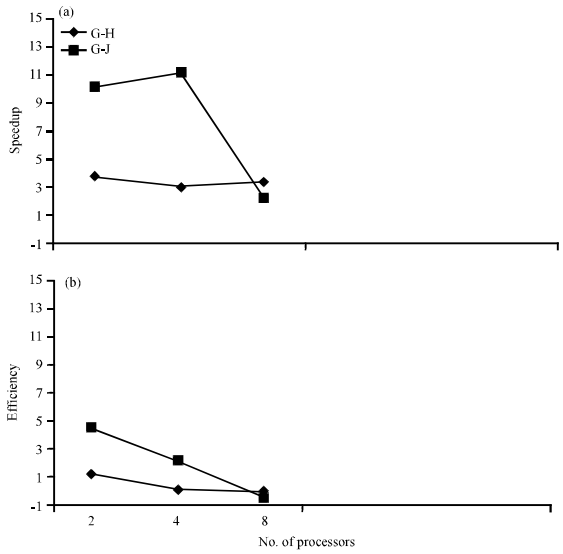

Fig. 2: Two algroithm for a system of order 2048 on 8 processors

After this limit the difference in the speedup ratios decreases between the two algorithms because the required number of exchange messages for Gauss-Huard algorithm increases more than what is required for Gauss-Jordan algorithm .

CONCLUSION

This study has conducted a comparable study of parallel implementation of Gauss-Jordan and Gauss-Huard algorithms for solving dense systems of linear equations. Timing models are first derived. Then experimental performance results are obtained for an implementation of the algorithms on a cluster of Linux workstations interconnected by a high speed star network. The obtained results indicate that both proposed parallel algorithms offer high speed, efficiency and numerical stability. However, the Gauss-Huard parallel implementation has slightly outperformed than Gauss-Jordan parallel implementation.

REFERENCES

Aizenbud, Y., G. Shabat and A. Averbuch, 2016. Randomized LU decomposition using sparse projections. Comput. Math. Appl., 72: 2525-2534.

Al-Towaiq, M. and H. Al-Aamri, 2002. A parallel implementation of GESPP on a cluster of Silicon Graphics workstations. Proceedings of the 9th International Conference on Parallel and Distributed Systems, December 17-20, 2002, IEEE, Taiwan, China, pp: 226-230.

Al-Towaiq, M., 2007. Clustered Gauss-Huard algorithm for the solution of Ax = b. Appl. Math. Comput., 184: 485-495.

Al-Towaiq, M., F. Masoud, A.B. Mnaour and K. Day, 2008. An implementation of A parallel iterative algorithm for the solution of large banded systems on a cluster of workstations. Intl. J. Model. Simul., 28: 378-386.

Al-Towaiq, M.H., 2013. Parallel implementation of the Gauss-Seidel algorithm on K-Ary n-Cube machine. Appl. Math., 4: 177-182.

Dekker, T.J., W. Hoffmann and K. Potma, 1997. Stability of the Gauss-Huard algorithm with partial pivoting. Comput., 58: 225-244.

Duff, I.S. and H.A.V.D. Vorst, 1999. Developments and trends in the parallel solution of linear systems. Parallel Comput., 25: 1931-1970.

Hoffmann, W., 1998. The Gauss-Huard algorithm and LU factorization. Linear Algebra Appl., 275: 281-286.

Jia, J. and Y. Jiang, 2015. Two symbolic algorithms for solving general periodic pentadiagonal linear systems. Comput. Math. Appl., 69: 1020-1029.

Kahou, G.A.A., L. Grigori and M. Sosonkina, 2008. A partitioning algorithm for block-diagonal matrices with overlap. Parallel Comput., 34: 332-344.

Lastovetsky, A. and R. Reddy, 2007. Data distribution for dense factorization on computers with memory heterogeneity. Parallel Comput., 33: 757-779.

Marrero, J.A., 2016. A numerical solver for general bordered tridiagonal matrix equations. Comput. Math. Appl., 72: 2731-2740.

Peters, G. and J.H. Wilkinson, 1975. On the stability of Gauss-Jordan elimination with pivoting. Commun. ACM., 18: 20-24.

Sidi, A., 2008. Vector extrapolation methods with applications to solution of large systems of equations and to PageRank computations. Comput. Math. Appl., 56: 1-24.

Trefethen, L.N. and R.S. Schreiber, 1990. Average-case stability of Gaussian elimination. SIAM. J. Matrix Anal. Appl., 11: 335-360.

Zhang, J. and H. Dai, 2016. Inexact splitting-based block preconditioners for block two-by-two linear systems. Appl. Math. Lett., 60: 89-95.

Zhu, Y. and A.H. Sameh, 2007. PSPIKE+: A family of parallel hybrid sparse linear system solvers. J. Comput. Appl. Math., 311: 682-703.