Implementation of A Parallel Algorithm for the Symmetric Positive Definite Systems of Equations on the CRAY-T3E

Technical Report B-17-99
Nov. 30, 1999

Rajeev Wankar[1]
Rainald Ehrig*
N.S.Chaudhari**
Elfriede Fehr

Freie Universität Berlin
Institut für Informatik
wankar@inf.fu-berlin.de
ehrig@zib.de
fehr@inf.fu-berlin.de
*Konrad-Zuse-Zentrum für Informationstechnik (ZIB) Berlin
**School of Computer Science, DAVV Indore, India

Abstract

A parallel algorithm for the solution of dense Symmetric Positive Definite (SPD) systems of equations Ax = b has been designed for the implementation on the CRAY T3E. One of the numerically stable methods for the solution of this system is proposed by Delosme & Ipsen [3]. In order to implement this algorithm on the CRAY T3E, we require to handle the procedures involved in a slightly different way. These implementation issues are discussed in detail. The actual timings for different communication schemes, on different sets of data values and varying number of processors have been tested and reported.

Keywords: SPD, MPI, SHMEM.

Introduction: Symmetric Positive Definite (SPD) systems of equations are a special class of problem which arises very frequently in many applications like linear programming, weather forecasting, seismic data processing etc. One of the numerically stable approaches to solve this system of equations is Cholesky factorization. Besides factorization, this method based on Cholesky factorization involves forward elimination and back substitution. Delosme and Ipsen[3] have given a method which uses hyperbolic rotations to obtain the solution of such system. They replaced sequential steps with parallel ones. Straight forward implementation of this algorithm on

1. This work is supported by the German Academic Exchange Services (DAAD) under the “Sandwich Model” fellowship with the author.

a computer requires much more memory compared to the standard Cholesky method. In this technical report we discuss the issue of effective utilisation of space by modifying many intermediate steps of the algorithm given by Delosme and Ipsen. Theoretically, speed up can be achieved by replacing multiplications to additions, this issue is also handled. The report consists of 3 sections: In Section 1 we present the brief description of the method proposed by Delosme and Ipsen. In Section 2 we present sequential algorithm, followed by a parallel algorithm and discuss the issue of reducing the floating point operations. In Section 3 implementation related issues are addressed. We also discuss space utilisation and time complexity and give the execution timing details of different implementations in the form of graphs (Figure 2-7).

Section 1: Algorithm based on hyperbolic Cholesky factorization.

In this section a brief discussion on the method of Delosme and Ipsen[3], based on hyperbolic rotations, for the solution of linear systems of equations

$$Ax = b \tag{1.1}$$

with a symmetric positive definite coefficient matrix *A*, is given. The Cholesky factor of *A* is first determined by essentially pre-multiplying *A* with appropriate hyperbolic rotations, whose product is called *Q*. Next simple matrix vector multiplication involving *Q* to the right hand side of (1.1) provides a novel way of solving the system. Avoiding forward elimination and back substitution is a very desirable feature for a parallel implementation.

1.1 The hyperbolic Cholesky factorization.

The computation of the Cholesky decomposition,

$$A = U^T U,\ U_{n \times n} \text{ upper triangular,} \tag{1.2}$$

of real symmetric positive definite(spd) $n \times n$ matrix $A = a_{ij}$, by means of hyperbolic rotations is called **hyperbolic Cholesky factorization**. Its derivation is based on a particular decomposition of the matrix *A:*

$$A = \sum_{k=1}^{n} A^{(k)}, \tag{1.3}$$

where $A^{(k)}$ has elements

$$a_{i,j}^{(k)} = \begin{cases} a_{i,j} & i = k,\ j \ge i \text{ or } j = k,\ i \ge j, \\ 0 & \text{otherwise,} \end{cases} \tag{1.4}$$

Since *A* is spd, $a_{k,k}$ is strictly positive and $A^{(k)}$ can be written as the difference of the outer product.

$$A^{(k)} = v_k^T v_k - w_k^T w_k, \tag{1.5}$$

where v_k and w_k are row vectors with elements

$$v_{k,j} = \begin{cases} a_{k,k}^{-1/2}\, a_{k,j} & j \ge k \\ 0 & \text{otherwise,} \end{cases} \tag{1.6}$$

$$w_{k,j} = \begin{cases} v_{k,j} & j \ne k \\ 0 & j = k \end{cases} \tag{1.7}$$

That is, v_k consists of non zero rows of $A^{(k)}$ scaled by the square root of the diagonal elements, while w_k differs from v_k only in its k^th^ entry. Stacking the v_k and w_k respectively in upper triangular matrices

$$V = \begin{bmatrix} v_1 \\ v_2 \\ \cdot \\ v_n \end{bmatrix},\ W = \begin{bmatrix} w_1 \\ w_2 \\ \cdot \\ w_n \end{bmatrix}, \tag{1.8}$$

one has $A = V^T V - W^T W$. (1.9)

Definition 1.1: A $2m \times 2m$ matrix Θ is called pseudoorthogonal if it satisfies

$$\Theta^T \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \Theta = \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix}, \text{ where I is the } m \times m \text{ identity matrix.}$$

***Lemma 1.1*:** Let R and S be upper triangular $n \times n$ matrices such that $R^T R - S^T S$ is positive definite, and let $\mu_{k,k} = r_{k,k}^{-1} s_{k,k}$, $1 \le k \le n$. Let

$$\begin{bmatrix} \tilde{R} \\ \tilde{S} \end{bmatrix} = \hat{Q} \begin{bmatrix} R \\ S \end{bmatrix}, \qquad \text{where } \hat{Q} = \tilde{Q}^{(n)} \ldots \tilde{Q}^{(1)} \text{ and}$$

$$\tilde{q}_{ij}^{(k)} = \begin{cases} 1 & i = j \neq k \text{ or } i = j \neq n + k, \\ (1 - \mu_k^{-2})^{-1/2} & i = j = k \text{ or } i = j = n + k, \\ -(1 - \mu_k^2)^{-1/2} \mu_k & (i, j) = (k, n + k) \text{ or } (i, j) = (n+k, k), \\ 0 & \text{otherwise,} \end{cases} \tag{1.10}$$

then $\hat{Q}$ is pseudoorthogonal, $\tilde{R}$ is upper triangular and $\tilde{S}$ is strictly upper triangular (upper triangular with zero diagonal).

Remarks:

1. Since the matrix $\tilde{Q}^{(k)}$ constitute disjoint rotations, they commute and can be applied in any order. Their product $\hat{Q}$ has a very simple expression:

$$\hat{q}_{k,k} = \hat{q}_{n+k,n+k} = (1 - \mu_k^2)^{-1/2}, \quad 1 \leq k \leq n, \tag{1.11}$$

$$\hat{q}_{k,n+k} = \hat{q}_{n+k,k} = -(1 - \mu_k^2)^{-1/2} \mu_k, \quad 1 \leq k \leq n, \tag{1.12}$$

$$\hat{q}_{ij} = 0, \qquad i \neq j (\text{mod n}). \tag{1.13}$$

2. The diagonal elements of $\tilde{R}$ have same sign as the corresponding diagonal elements of R; thus if R has a positive diagonal, $\tilde{R}$ also has a positive diagonal.

Theorem 1.1 **(The Hyperbolic Cholesky Algorithm):**

Let A be a $n \times n$ spd matrix and V and W be upper triangular matrices as defined in (1.6) and (1.7), so that $A = V^T V - W^T W$. Set

$$\begin{bmatrix} R^{(0)} \\ S^{(0)} \end{bmatrix} = \begin{bmatrix} V \\ W \end{bmatrix}, \tag{1.14}$$

and apply the sequence of operations

$$\begin{bmatrix} R^{(l+1)} \\ S^{(l+1)} \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & P \end{bmatrix} \hat{Q}^{(l)} \begin{bmatrix} R^{(l)} \\ S^{(l)} \end{bmatrix}, \tag{1.15}$$

$$l = 0, \ldots, n-1,$$

where $\hat{Q}^{(l)}$ is obtained from $R^{(l)}$ and $S^{(l)}$ in the same way as $\hat{Q}$ is constructed from R and S in lemma 1.2, and where P is the $n \times n$ circular permutation matrix with $P_{1,n} = 1$ and $P_{i,i-1} = 1$, $2 \le i \le n$. Then $R^{(n)} = U$, the Cholesky factor of A, and $S^{(n)} = 0$.

Remarks:

The transformation performed by the hyperbolic Cholesky algorithm is denoted by

$$Q = \begin{bmatrix} I & 0 \\ 0 & P \end{bmatrix} \hat{Q}^{(n-1)} \cdots \begin{bmatrix} I & 0 \\ 0 & P \end{bmatrix} \hat{Q}^{(l)} \cdots \begin{bmatrix} I & 0 \\ 0 & P \end{bmatrix} \hat{Q}^{(0)}. \tag{1.16}$$

The matrix Q is pseudoorthogonal, since it is the product of pseudoorthogonal matrices.

1.1 Application of hyperbolic rotations to the solution of linear systems:

The hyperbolic Cholesky algorithm determines simultaneously the Cholesky factor U of an spd matrix A and a set of $n(n-1)/2$ parameters $\mu_k^{(l)}$, $1 \le l \le n-1$, $1 \le k \le n$, which defines the hyperbolic rotations that make up the matrix Q. For the solution of the spd system $Ax = b$, the above algorithm could be used to find U followed by forward elimination to solve system $U^T y = b$ and by back substitution to solve $Ux = y$. Instead, the above algorithm can also be used to find parameter $\mu_k^{(l)}$ followed by the application of the hyperbolic rotations to the b in a particular way to get the solution vector x.

The algorithm for the solution of $Ax = b$ can be summed up as follows: Let A_α be the upper part of A and A_β its strictly upper triangular part ($A_\alpha = D^2 + A_\beta$). Using the hyperbolic Cholesky algorithm as specified in the theorem, express the matrix Q as a product of hyperbolic rotations such that

$$Q\Delta \begin{bmatrix} A_\alpha \\ A_\beta \end{bmatrix} = \begin{bmatrix} U \\ 0 \end{bmatrix}, \tag{1.17}$$

where

$$\Delta = \begin{bmatrix} D^{-1} & 0 \\ 0 & D^{-1} \end{bmatrix}, \tag{1.18}$$

and $D = \mathrm{diag}\,(a_{11}^{1/2}, \ldots, a_{nn}^{1/2})$.

Apply the same operation to $\begin{bmatrix} b \\ b \end{bmatrix}$ to obtain

$$Q\Delta\begin{bmatrix} b \\ b \end{bmatrix} = \begin{bmatrix} U^{-T}b \\ L^{-T}b \end{bmatrix}, \tag{1.19}$$

and then apply these operations essentially in reverse order to get x from

$$x = (I, I)\Delta Q^T\begin{bmatrix} \alpha U^{-T}b \\ (1-\alpha)L^{-T}b \end{bmatrix}, \tag{1.20}$$

where α is an arbitrary real number, which can be equal to 0 or 1 for convenience.
and Q^T is given by

$$Q^T = \hat{Q}^{(0)}\begin{bmatrix} I & 0 \\ 0 & P^{-1} \end{bmatrix}\hat{Q}^{(1)}\begin{bmatrix} I & 0 \\ 0 & P^{-1} \end{bmatrix}\ldots\hat{Q}^{(n-1)}\begin{bmatrix} I & 0 \\ 0 & P^{-1} \end{bmatrix}. \tag{1.21}$$

Section 2: A parallel algorithm for SPD:

In this section we first present the sequential algorithm and then parallel algorithms for the SPD. We discuss the modifications made for the implementation on the CRAY T3E and analyse time complexity. The issue of saving memory by avoiding many intermediate calculations is also elaborated. By observing the parallel algorithm presented in [3], we see that the maximum time is needed to obtain the hyperbolic Cholesky factorization. In this step besides the hyperbolic Cholesky factor R, we obtain hyperbolic rotations μ which makes a $2n \times 2n$ pseudoorthogonal matrix Q. We observe the following things.

1. The multiplication of a $2n \times 2n$ matrix with another $2n \times 2n$ matrix $\begin{bmatrix} I & 0 \\ 0 & P \end{bmatrix}$, where P is a $n \times n$ circular permutation matrix, is a sequence of row interchange operations. The row *2n* of the second matrix becomes the row *n+1*, the row *n+1* becomes row *n+2* and so on. The row *2n-1* becomes row *2n*.

2. Instead of calculating a $2n \times 2n$ pseudoorthogonal matrix Q^k in each iteration, we can calculate rotations μ and later use them in the right hand side b to get the solution vector.

The above mentioned observations leads us to a simple sequential algorithm 2.1.

Algorithm 2.1

```
for i:= 1 to n-1 do
  begin
     for  j := i+1 to n
     begin
         A(j,i) := S(j,j-i)/R(j,i)
         t1 := 1/(sqrt(1 - A^2(j, i)))
         t2 := -A(j,i) * t1;
          for k := j to n
          begin
              t3 := R(k,j);
              R(k,j) := t1*t3+t2*S(k,j-i);
              S(k,j-i) := t2*t3+t1*S(k,j-i);
          end;
      end;
end;
```

Algorithm 2.1 shows that the second and the third loop with indices *j* and *k* can be parallelised since within this loop the algorithm works in every step on different rows of *A, R* and *S*. The Parallelisation of the algorithm 2.1 leads to the algorithm 2.2.

Algorithm 2.2

```
for i:= 1 to n-1 do
  begin
     for all  j := i+1 to n in parallel do
     begin
         A(j,i) := S(j,j-i)/R(j,i)
         t1 := 1/(sqrt(1 - A^2(j, i)))
         t2 := -A(j,i) * t1;
          for all k := j to n in parallel do
          begin
              t3 := R(k,j);
              R(k,j) := t1*t3+t2*S(k,j-i);
              S(k,j-i) := t2*t3+t1*S(k,j-i);
          end;
      end;
end;
```

From algorithm 2.2 we further observe that:

1. The first row of R is never used and the row k of R is used only if $j = k$ and $i = k-1$. Therefore we do not need to re-compute row $i+1$ of R for every value i of the outermost loop.

2. Each element $S_{p,q}$ of S is used if $i = q-p$ and $j = q$ and later set to zero. Therefore it is not needed to re-compute $S_{j-i,j}$ for $k = j$, for all *i, j*.

These considerations lead us to the following modified algorithm 2.3.

Algorithm 2.3

```
for i := 1 to n-1 do
begin
    for all  j := i+1 to n in parallel do
    A(j,i) := S(j,j-i)/R(j,j)
    begin
        A(j,i) := S(j,j-i)/R(j,i);
        t1 := 1/(sqrt(1 - A^2(j, i)));
        t2 := -A(j,i) * t1;
        if (j=i+1) then
          for all k := j+1 to n in parallel do
             S(k,j-i) := t2*R(k,j) + t1*S(k,j-i)
        else
        begin
            R(j,j) := t1 *R(j,j) + t2*S(j,j-i);
            for all k := j+1 to n in parallel do
            begin
                t3 := R(k,j);
                R(k,j) := t1*t3+t2*S(k,j-i);
                S(k,j-i) := t2*t3+t1*S(k,j-i);
            end;
        end;
    end;
end;
```

2.1 Operation Counts: The innermost loop with index k of the parallel algorithm 2.3 is one which can be used to calculate the sequential computing time of the algorithm 2.3. If T(n) represents the time needed for execution then we express:

$$T(n) = \begin{cases} n(n-1)(n-2)/6 & n > 2 \\ 0 & Otherwise \end{cases}$$

Since the innermost loop with index k consists of 4 multiplications and if the cost of each multiplication is constant then the total execution time of the two steps inside the loop is $2n^3/3$ floating point operations. Theoretically, the actual cost of the algorithm is generally determined by multiplication and not by addition/subtraction. Closely observing the algorithm, we see that the innermost loop consists of four multiplications and two additions. By applying the following observations we can replace multiplications to additions and get a new parallel algorithm 2.4 with reduced sequential time $n^3/3$.

Algorithm 2.4

```
for i := 1 to n-1 do
begin
    for all  j := i+1 to n in parallel do
    A(j,i) := S(j, j-i)/R(j, j)
    begin
        A(j,i) := S(j, j-i)/R(j, i);
        t1 := 1/(sqrt(1 - A^2(j, i)));
        t2 := -A(j, i) * t1;
        t3:= (t1-t2)/2;
        t4 := (t1+t2)/2;
        if (j=i+1) then
          for all k := j+1 to n in parallel do
             S(k, j-i) := t2*R(k, j) + t1*S(k, j-i)
        else
        begin
            R(j, j) := t1 *R(j, j) + t2*S(j, j-i);
            for all k := j+1 to n in parallel do
            begin
                t5 := t3 * (R(k, j) + S(k, j-1));
                t6 := t4 * (R(k, j) - S(k, j-1));
                R(k, j) := t5 + t6;
                S(k, j-i) := t5 - t6;
            end;
        end;
    end;
end;
```

It can be seen that the reduction in multiplications causes increase in additions/subtractions. Similar tactics can be applied to find the solution of linear systems as well.

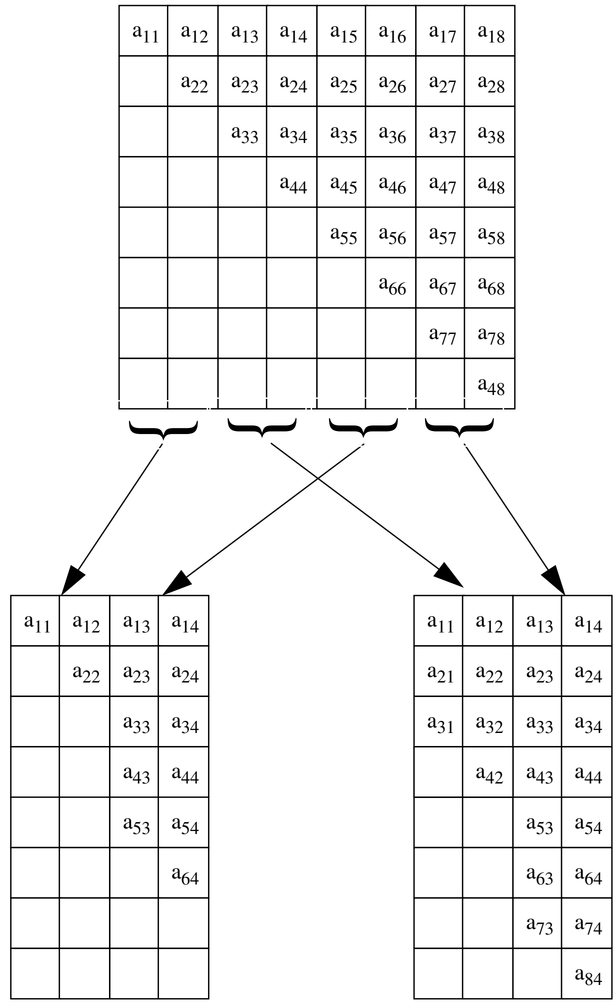
Section 3: Implementation on CRAY:

There are many strategies which can be adopted for the implementation of the SPD systems of equations on a parallel machine. In this work we have investigated the effect of different implementation strategies. During the design of any algorithm on parallel computer, one has to correctly make use of certain basic communication routines in order to exchange data between different processors. The communication patterns may be:

1. Point to point communication.
2. Common broadcast by a processor to all other processors.
3. Collective sending operation to one processor
4. Collective sending from all to all processors.

Combinations of some or all of them are normally applied for the effective communication in a parallel algorithm. Speed of the parallel algorithm depends on the routines we use. We implemented the algorithm using different communication possibilities and different sets of routines. In the first implementation we used CRAY specific shared memory communication routines (SHMEM library) [13], which are very fast but have the problem of portability. In this approach we used the cyclic row distribution on processors and the program is written in FORTRAN. In the second implementation we used Message Passing Interface (MPI) communication routines [7,12]. These are portable and the same algorithm can be executed on heterogeneous environments and it is faster than the first approach as well. In this approach the program is written in C. In C the multi dimensional arrays are stored in a row major order in the memory. We used the column distribution approach on processors since the operations are performed on elements whose column indices increase rapidly. We have implemented the same algorithm in C using row distribution approach. We have shown how features of languages can affect execution time of the algorithm using graphs. The programs are executed on the CRAY T3E having 256 processors. We tested the performance of the algorithm on two different types of PEs (small 450MHz/128MB and big 600MHz/512MB). In this section we describe only the column version of the algorithm.

For the sake of simplicity we assume that the problem size $n = k \times nblock \times nproc$, thus every processor is having *k*, *nblock* of the SPD matrix *A*. Since the matrix is symmetric and positive definite, we only need to store lower(upper) part along with the diagonal entries. In our approach we store the entries of *A* in upper triangular part and the lower part of the matrix is used to store the hyperbolic rotations, thus saving $n \times (n-1)/2$ locations. In this implementation we used a global array *processor* which is used to store the identity of the processor which holds the specific columns. The *localindex* is used to store the local index of a column obtained from the global index. We used arrays, *row* and *rowlower,* locally at each processor. Array *row* is used to find the counter for the column on a processor and array *rowlower* is used for knowing the number of columns managed by the particular processor in the right hand side of the actual column. Figure 1 shows the distribution of columns of the matrix *A* on different processors, entries are converted in to local indices. Here we have taken *n = 8*, *blocksize = 2* and number of processors = 2. The variable *nloc = 4.*



Matrix A on Processor 0 Matrix A on Processor 1

Figure 1.

The most time consuming step of the algorithm is obviously the factorization step. Referring to the subsection 1.1, it is clear that the entries of $R^{(l)}$, $S^{(l)}$ are determined by the entries of $R^{(l-1)}$, $S^{(l-1)}$, $1 \le l \le n$, in a manner described in Theorem 1.1. Since we have distributed columns of *A, S* on *n* processors, we calculate μ at every processor according to the definition given in Lemma 1.1 and store them to a portion of a one dimensional array *buffer*. The array *buffer* is then broadcasted by all the processors in each iteration of the factorization step, since, the entries to calculate the rotations must be available at other processors. After this broadcast step is over, the rotations are calculated according to the equation 1.8 and stored in the lower triangular part of the matrix *A*, but only at the respective processors. Entries of *A* are over written by the new entries of *R*. The loop is repeated n times. Here we do not calculate a $2n \times 2n$ pseudoorthogonal matrix in each iteration of the factorization step, instead we perform shift operation on the rotations stored in the lower triangular part of a matrix A, while calculating the solution. We have used the highly optimized collective communication routine **MPI_Bcast** to accomplish the task in this step.

In order to get the solution of the system, we need to calculate the pseudoorthogonal matrix Q described in the equation 1.18. As we already stated in the algorithm, multiplication of a $2n \times 2n$ matrix with another $2n \times 2n$ matrix $\begin{bmatrix} I & 0 \\ 0 & P \end{bmatrix}$, where P is a $n \times n$ circular permutation matrix, is a sequence of row interchange operations. The row *2n* of the second matrix becomes the row *n+1*, the row *n+1* becomes row *n+2* and so on. The row *2n-1* becomes the row *2n*. We see in the last steps of the algorithm that the rotations are stored in the lower triangular part. The interchange operation or a shift operation can be done efficiently by every processor within its local memory. This reduces the processor communication if we use cyclic column distribution.

3.1 Analysis of space:

In the algorithm implemented for CRAY, we need arrays *processor, localindex* of dimension $n + 2$, *rowlower* of the dimension $n + 1$ at every processor. The dimension of *b, bl, bu, diag* is $nloc$, where *nloc* is the number of columns managed by a particular processor. Arrays *a* and *s* are of size $n \times nloc$. Arrays *row, qe1* are one dimensional of size *"size"* where *"size"* is the number of processors used for the execution of the algorithm. The array *buffer* is of size *n*. Some variables for intermediate processing and index handling are also used.

3.2 Analysis of time:

We have tested the algorithm for different values of *n* with varying number of processors using three different strategies. The first one is row distribution approach with FORTRAN, the second is row distribution with C and the third one is column distribution with C. The algorithm is tested for different block sizes. The common choices are 16 and 8. It has been observed that the hyperbolic Cholesky factorization takes less time with block size 16 but the overall time for the solution with block size 8 is better. The program is tested for n = 4096, 2048, 1024, 512 and the number of processors used are 2,4,8 and 16. The algorithm is compared with the fastest parallel version of the classical Cholesky algorithm on the T3E machines[14].

The amount of time needed by the algorithm on different sets of data with varying processors and with the column distribution approach is described by the table presented below.

Table 1: Overall time required by the algorithm with column distribution approach

Problem Size	P = 2	P = 4	P = 8	P = 16
512	1.03 Sec.	0.72 Sec.	0.56 Sec.	0.58 Sec.
1024	6.09 Sec.	3.96 Sec.	2.94 Sec.	2.46 Sec.
2048	39.53 Sec.	23.69 Sec.	15.69 Sec.	12.09 Sec.
4096		168.82 Sec.	94.15 Sec.	62.95 Sec.

We have presented the number of processors used and the execution time of the algorithm as graphs. Figure 2 shows the graph between the number of processors and the time required for the hyperbolic Cholesky factorization for the problem size n = 2048. From figure 4 it is observed that the communication overhead increases the execution time for n = 512 when the number of processors are more than 12. Figure 5 shows the execution time between the two approaches namely row and column. The difference between the execution time increases when the number of processors are reduced. In figure 6 we compare the execution time of the classical parallel Cholesky and Hyperbolic Cholesky algorithm and can see that the classical method is still faster than the approach used in this paper. Figure 7 gives the comparison of these two methods while obtaining the solution. It can be seen that the hyperbolic Cholesky algorithm is faster in obtaining solution when the number of processors are more than 8.

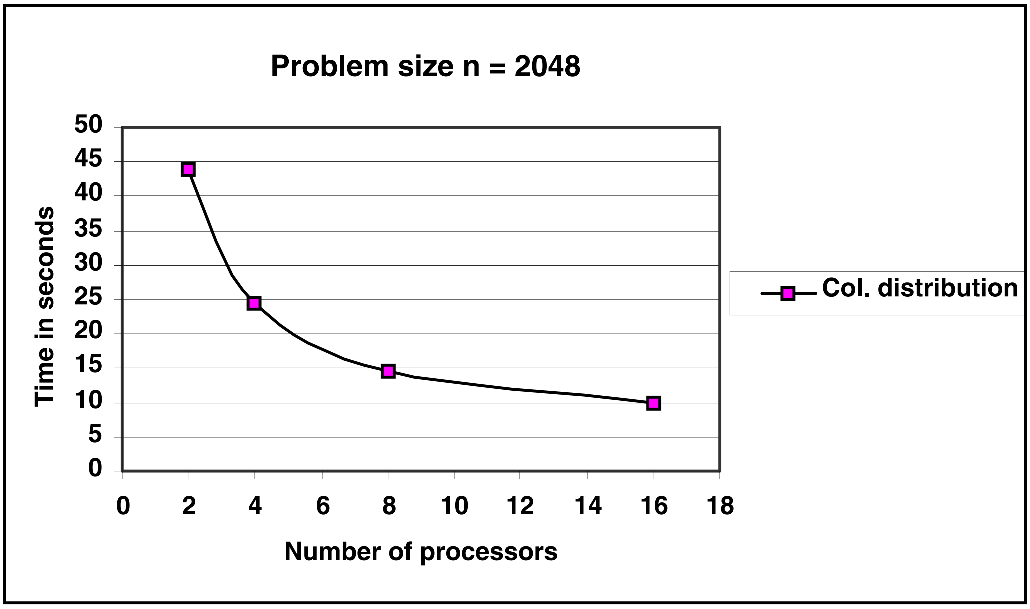


Figure 2

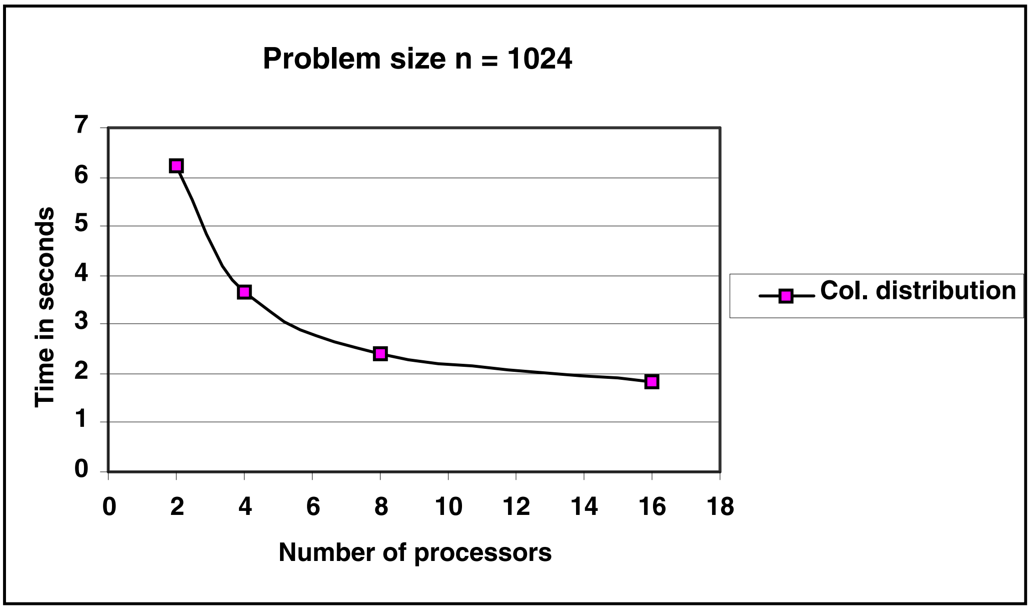


Figure 3

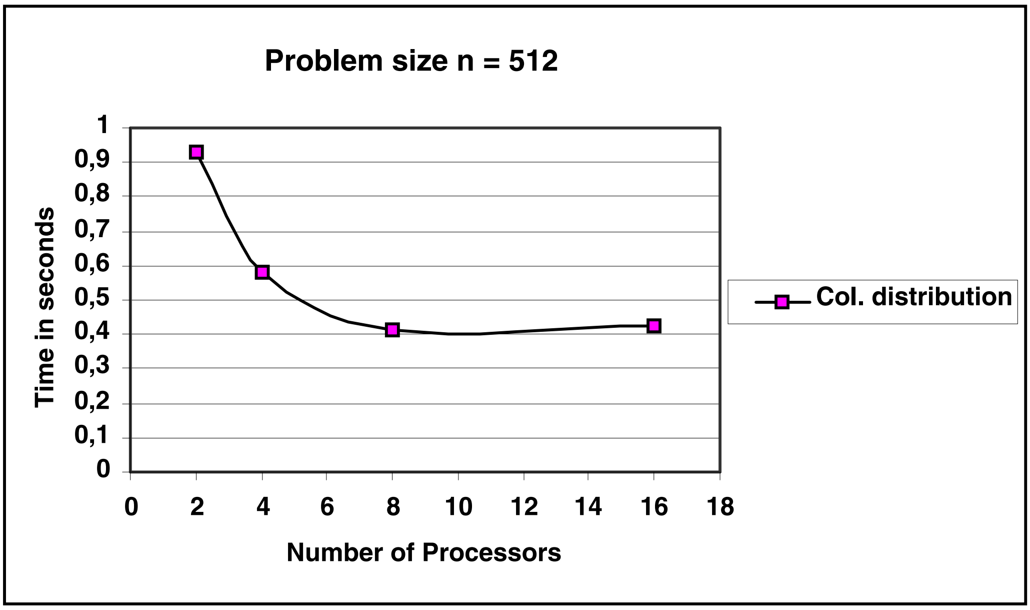


Figure 4

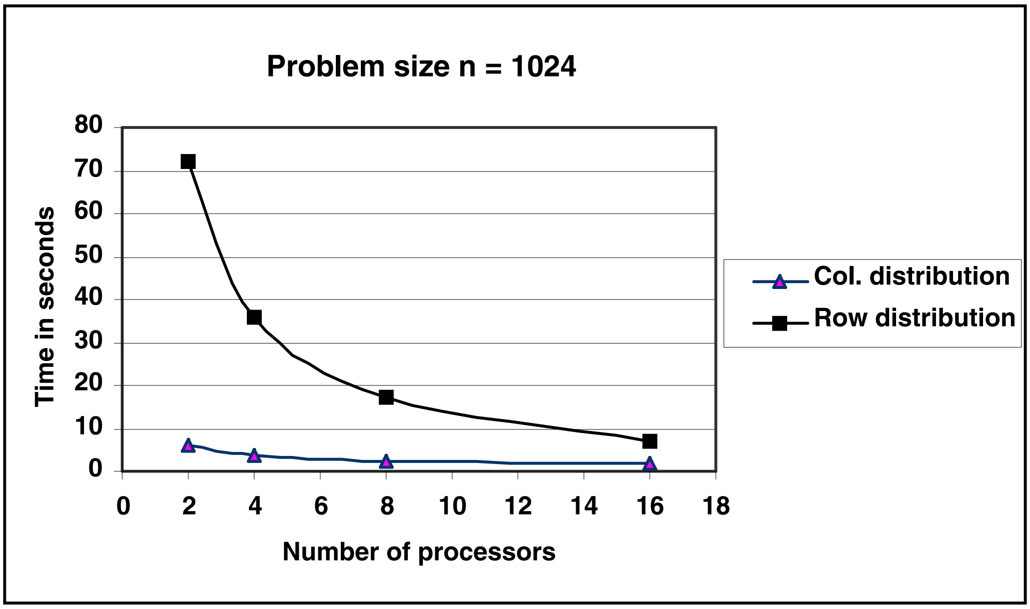


Figure 5

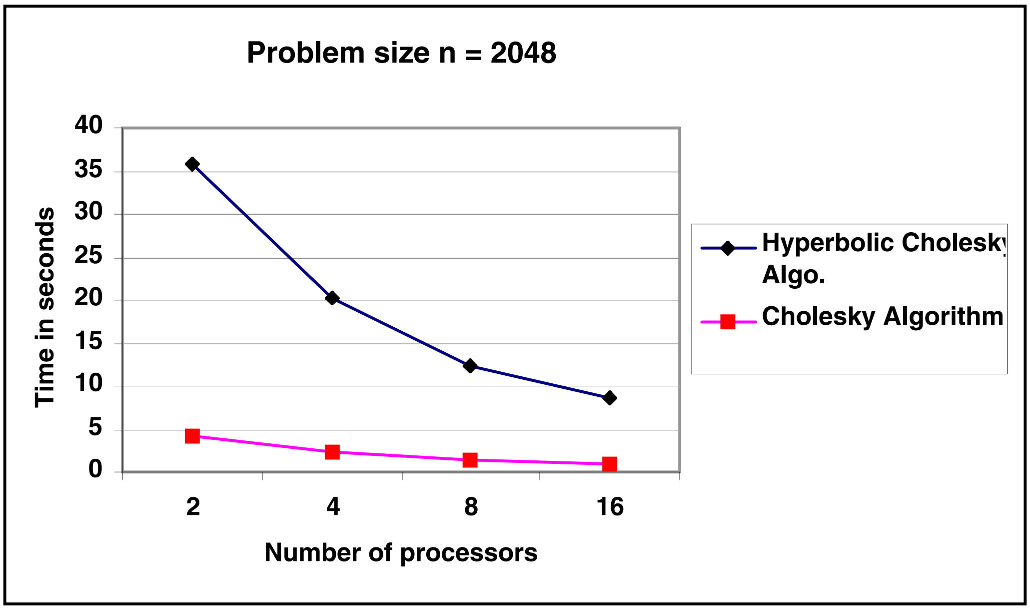


Figure 6

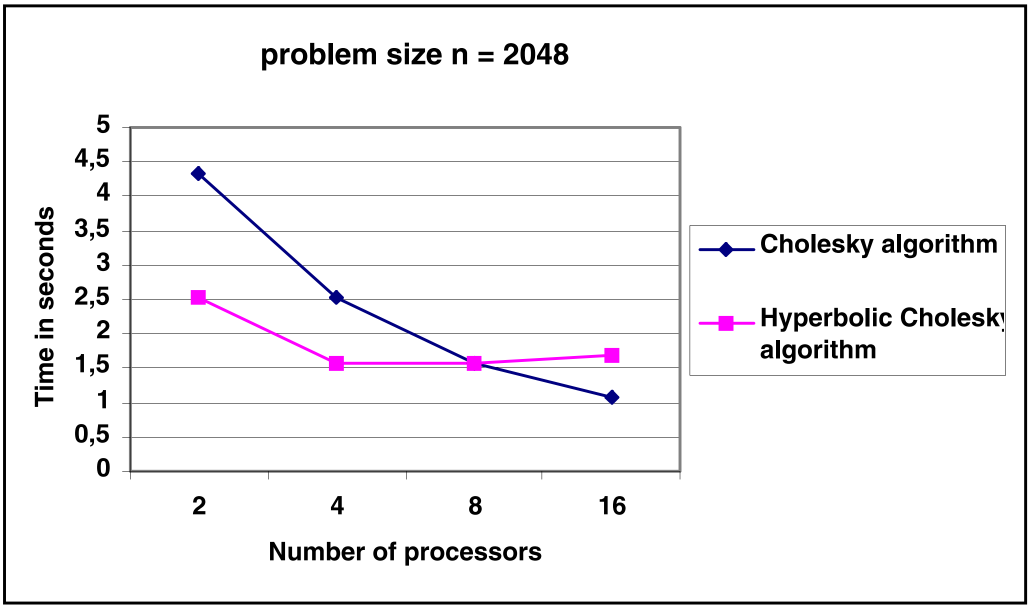


Figure 7

Conclusion:

A parallel algorithm for the solution of special linear system of equations has been presented. Space and time complexities with the different implementation routines, different set of data and different processors have been calculated and reported. The implementation of the algorithm on Message Passing Interface MPI is both fast and portable. Comparing results of this algorithm with the standard Cholesky factorization we infer that both versions of the algorithm, presented in the report, are slower. This is because the standard Cholesky factorization uses highly optimized mathematical and other routines. However, the hyperbolic Cholesky algorithm is faster compared to the Cholesky classical algorithm in obtaining the solution. We have seen the effect of the algorithm on sparse matrices too. The work can further be extended to investigate, in detail, the behaviour of the algorithm on special patterns of sparse matrices. The work is in progress to speed-up the algorithm by adopting different implementation possibilities.

References:

[1] Aho A., J. Hopcroft and J. Ullman, **The Design and Analysis of Computer Algorithms,** Addison-Wesley, 1974.

[2] Bauer B. E., **Practical Parallel Programming,** Academic Press Inc., 1992.

[3] Delosme J.M., Ilse C.F. Ipsen, “Parallel Solution of Symmetric Positive Definite Systems With Hyperbolic Rotations”, **Linear Algebra and its Applications,** Special Volume on Parallel Computing, North Holland, pp. 75-111, Vol. 77, May 1986.

[4] Gibbons A. And Wojciech Rytter, **Efficient Parallel Algorithms,** Cambridge University Press, Cambridge, May 1988.

[5] Golub G.H. and Charles and F. Van Loan, **Matrix Computations,** John Hopkins Press, Baltimore, MA, 1983.

[6] Golub G.H. and James Ortega, **Scientific Computing: An Introduction with Parallel Computing,** Academic Press Inc., 1993.

[7] Gropp W., Ewing Lusk and Anthony Skjellum, **Using MPI, Parallel Portable Programming with the Message Passing Interface**, MIT Press 1995.

[8] Horowitz E. And S.Sahni, **Fundamentals of Computer Algorithms,** Addison-Wesley, Computer Science Press, NY, 1978.

[9] Lancaster P.,M. Tismenetsky, **The Theory of Matrices,** Academic Press Inc., (1985).

[10] Lester B. P**., The Art of Parallel Programming,** Prentice Hall, Englewood Cliffs, N.J. (1993).

[11] Louis A. Hageman and David M. Young, **Applied Iterative Methods,** Academic Press Inc., NY, 1981.

[12] **MPI: A message Passing Interface Standard**, Message Passing Interface Forum, June 12, 1995.

[13] R. Barriuso, A. Knie: **SHMEM User’s Guide for FORTRAN, Rev. 2.2.** Cray Research Inc. 1994.

[14]. Robert A. van de Geijn, **Using PLAPACK**, The MIT Press, Cambridge, 1997.

[15] Stewart G.W., **Introduction to Matrix Computations,** Academic Press Inc., SanDiego California 1973.

[16] Vipin Kumar et al., **Introduction to Parallel Computing, Design and Analysis of Algorithms,** The Benjamin Cummings Publishing Company Inc., California, 1994.

[17] Wankar R. and N.S.Chaudhari, “Parallel Cholesky Factorization Algorithm”, **National Conference on current trends in Information Technology,** pp. 23-24, June 1995, Bhopal, India.

[18] Wankar R., “Parallel Algorithms for Solving Symmetric Positive Definite System (SPD) of Equations”, **International Journal of Management and Systems,** pp. 311-324, Vol. 11, No. 3, Sept.-Dec. 95.

[19] Wankar R., E. Fehr and N. S. Chaudhari, “A Fast Parallel Algorithm for Special Linear Systems of Equations using Processor Arrays with Reconfigurable Bus Systems”, **Technical Report B-2-99,** January 29, 1999, Freie Universität Berlin, Germany.