

**A Recursive Doubling Algorithm  
for Solution of Tridiagonal Systems  
on Hypercube Multiprocessors**

*Omer Egecioglu* <sup>†‡</sup>

Department of Computer Science  
University of California  
Santa Barbara, CA 93106

*Cetin K. Koc* <sup>††</sup>

*Alan J. Laub* <sup>††</sup>

Scientific Computation Laboratory  
Department of Electrical & Computer Engineering  
University of California  
Santa Barbara, CA 93106

---

This article was presented at the Third SIAM Conference on Parallel Processing for Scientific Computing, Los Angeles, California, December 1-4, 1987, and The Third Conference on Hypercube Concurrent Computers and Applications, California Institute of Technology, JPL, Pasadena, California, January 19-20, 1988.

<sup>†</sup> Supported in part by NSF Grant No. DCR-8603722.

<sup>† †</sup> Supported in part by Lawrence Livermore National Laboratory Contract No. LLNL-7526225 and the National Science Foundation (and AFOSR) under Grant No. ECS84-06152.

<sup>‡</sup> Author to whom correspondence should be addressed.

## ABSTRACT

The recursive doubling algorithm as developed by Stone can be used to solve a tridiagonal linear system of size  $n$  on a parallel computer with  $n$  processors using  $O(\log n)$  parallel arithmetic steps. In this paper, we give a limited processor version of the recursive doubling algorithm for the solution of tridiagonal linear systems using  $O(\frac{n}{p} + \log p)$  parallel arithmetic steps on a parallel computer with  $p < n$  processors.

The main technique relies on fast parallel prefix algorithms, which can be efficiently mapped on the hypercube architecture using the binary-reflected Gray code. For  $p \ll n$  this algorithm achieves linear speed-up and constant efficiency over its sequential implementation as well as over the sequential LU decomposition algorithm. These results are confirmed by numerical experiments obtained on an Intel iPSC/d5 hypercube multiprocessor.

Categories and Subject Descriptors: F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems-*Computations on matrices*; G.1.0 [**Numerical Analysis**]: General-*Numerical algorithms, parallel algorithms*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra-*Linear systems (direct and iterative methods)*; G.4 [**Numerical Analysis**]: Mathematical Software-*Algorithms Analysis, Efficiency*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)-*Multiple-instruction-stream, multiple-data-stream processors (MIMD)*

General Terms: Tridiagonal systems, hypercube multiprocessors

Additional Key Words and Phrases: Recursive doubling, mapping, binary-reflected Gray code, parallel prefix, LU decomposition, speed-up and efficiency.



$$\mathbf{A} = \mathbf{L}\mathbf{U} = \begin{bmatrix} 1 & & & & & \\ e_1 & 1 & & & & \\ & & \ddots & & & \\ & & & e_{n-2} & 1 & \\ & & & & & e_{n-1} & 1 \end{bmatrix} \begin{bmatrix} f_0 & c_0 & & & & \\ & f_1 & c_1 & & & \\ & & & \ddots & & \\ & & & & f_{n-2} & c_{n-2} \\ & & & & & f_{n-1} \end{bmatrix}$$

The algorithm then proceeds to solve for  $\mathbf{y}$  from  $\mathbf{L}\mathbf{y} = \mathbf{d}$  and then finds  $\mathbf{x}$  by solving  $\mathbf{U}\mathbf{x} = \mathbf{y}$ . More precisely, the LU decomposition algorithm ( the LU Algorithm ) to solve the system (1) consists of the following steps:

**The LU Algorithm**

*Step 1.* Compute LU decomposition of  $\mathbf{A}$  given by

$$\begin{aligned} f_0 &= b_0 \\ e_i &= a_i / f_{i-1} & 1 \leq i \leq n-1 \\ f_i &= b_i - e_i * c_{i-1} & 1 \leq i \leq n-1 \end{aligned}$$

*Step 2.* Solve for  $\mathbf{y}$  from  $\mathbf{L}\mathbf{y} = \mathbf{d}$  using

$$\begin{aligned} y_0 &= d_0 \\ y_i &= d_i - e_i * y_{i-1} & 1 \leq i \leq n-1 \end{aligned}$$

*Step 3.* Compute  $\mathbf{x}$  by solving  $\mathbf{U}\mathbf{x} = \mathbf{y}$  using

$$\begin{aligned} x_{n-1} &= y_{n-1} / f_{n-1} \\ x_i &= ( y_i - c_i * x_{i+1} ) / f_i & 0 \leq i \leq n-2 \end{aligned}$$

We record the number of arithmetic operations required by the algorithm as

**THEOREM 1.**

*The LU Algorithm solves the tridiagonal linear system of size  $n$  using  $8n - 7$  arithmetic operations.*

**PROOF.**

The proof is straightforward counting of the number of multiplication, division and subtraction operations performed in *Steps 1, 2, and 3* above. •

**2. SOLUTION OF TRIDIAGONAL SYSTEMS USING PREFIX ALGORITHMS**

The equation (1) can be represented as a three-term recurrence relation

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i \tag{2}$$

for  $1 \leq i \leq n-2$  with

$$\begin{aligned} b_0 x_0 + c_0 x_1 &= d_0 \\ a_{n-1} x_{n-2} + b_{n-1} x_{n-1} &= d_{n-1} \end{aligned}$$

Define  $a_0 = c_{n-1} = 1$  and  $x_{-1} = x_n = 0$ . Then with this convention, the relation in (2) holds for  $0 \leq i \leq n-1$ .

Solving for  $x_{i+1}$  in equation (2) we get

$$x_{i+1} = -\frac{b_i}{c_i} x_i - \frac{a_i}{c_i} x_{i-1} + \frac{d_i}{c_i} \tag{3}$$

Here we assume that all  $c_i$ 's are nonzero, since otherwise the system of equations can be broken into two decoupled tridiagonal systems which can then be treated separately. Setting

$$\alpha_i = -\frac{b_i}{c_i} \quad \beta_i = -\frac{a_i}{c_i} \quad \gamma_i = \frac{d_i}{c_i}$$

(3) can be rewritten as

$$x_{i+1} = \alpha_i x_i + \beta_i x_{i-1} + \gamma_i$$

for  $0 \leq i \leq n-1$ . This recurrence formula can be put in a matrix form neatly as

$$\begin{bmatrix} x_{i+1} \\ x_i \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_i & \beta_i & \gamma_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ x_{i-1} \\ 1 \end{bmatrix}$$

which is essentially the same idea developed in [18]. Now define

$$X_i = \begin{bmatrix} x_i \\ x_{i-1} \\ 1 \end{bmatrix} \quad \text{and} \quad B_i = \begin{bmatrix} \alpha_i & \beta_i & \gamma_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} .$$

Then we may write

$$X_{i+1} = B_i X_i \quad 0 \leq i \leq n-1 \quad (4)$$

This matrix recursion formula allows us to calculate all  $X_i$  for  $1 \leq i \leq n-1$  provided that the initial vector  $X_0$  is available. Since

$$X_0 = \begin{bmatrix} x_0 \\ x_{-1} \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \\ 0 \\ 1 \end{bmatrix}$$

all we need is to calculate  $x_0$  to start the computation. Now note that by repeated application of (4) we obtain

$$X_1 = B_0 X_0$$

$$X_2 = B_1 X_1 = B_1 B_0 X_0$$

...

$$X_n = B_{n-1} B_{n-2} \cdots B_1 B_0 X_0 .$$

Now let

$$C_i = B_i B_{i-1} \cdots B_1 B_0 \quad 0 \leq i \leq n-1 .$$

Then  $X_n = C_{n-1} X_0$ , or more explicitly

$$\begin{bmatrix} x_n \\ x_{n-1} \\ 1 \end{bmatrix} = \begin{bmatrix} g_{00} & g_{01} & g_{02} \\ g_{10} & g_{11} & g_{12} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_{-1} \\ 1 \end{bmatrix} ,$$

where

$$C_{n-1} = \begin{bmatrix} g_{00} & g_{01} & g_{02} \\ g_{10} & g_{11} & g_{12} \\ 0 & 0 & 1 \end{bmatrix}$$

and the  $g_{ij}$  depend on  $\alpha_i, \beta_i, \gamma_i$  for  $0 \leq i \leq n-1$ . Since  $x_n = x_{-1} = 0$ , by multiplying the first row of  $C_{n-1}$  with  $X_0$  we obtain

$$0 = g_{00} x_0 + g_{02} ,$$

which gives us  $x_0$  as

$$x_0 = -\frac{g_{02}}{g_{00}} . \quad (5)$$

Once  $X_0$  is available in this manner, we can calculate all  $X_i$  for  $1 \leq i \leq n-1$  by using the matrix recursion formula  $X_i = C_{i-1} X_0$ .

The sequential prefix algorithm ( The SP Algorithm ) to solve the tridiagonal system (1) thus proceeds as follows :

**The SP Algorithm**

*Step 1.* Form the matrices  $B_i$  for  $0 \leq i \leq n-1$  using

$$\alpha_i = -\frac{b_i}{c_i} \quad \beta_i = -\frac{a_i}{c_i} \quad \gamma_i = \frac{d_i}{c_i}$$

and

$$B_i = \begin{bmatrix} \alpha_i & \beta_i & \gamma_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} .$$

*Step 2.* Compute the chain products  $C_i$  by

$$C_0 = B_0$$

$$C_i = B_i C_{i-1} \quad 1 \leq i \leq n-1$$

*Step 3.* Denote  $C_{n-1}$  computed in *Step 2* by

$$C_{n-1} = \begin{bmatrix} g_{00} & g_{01} & g_{02} \\ g_{10} & g_{11} & g_{12} \\ 0 & 0 & 1 \end{bmatrix} .$$

Compute  $x_0$  and hence  $X_0$  using

$$x_0 = -\frac{g_{02}}{g_{00}} ,$$

$$X_0 = \begin{bmatrix} x_0 \\ x_{-1} \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \\ 0 \\ 1 \end{bmatrix} .$$

*Step 4.* Compute  $X_i$  and hence  $x_i$  using

$$X_i = \begin{bmatrix} x_i \\ x_{i-1} \\ 1 \end{bmatrix} = C_{i-1} X_0 \quad 1 \leq i \leq n-1 .$$

*Step 2* of this algorithm essentially calculates prefixes of the matrices  $( B_0 , B_1 , B_2 , \dots , B_{n-1} )$  (here we imagine that the matrix products in performed in reverse order). If this algorithm is used to solve a tridiagonal system of dimension  $n$  sequentially, then  $O(n)$  arithmetic operations suffice, but the algorithm turns out to be slightly less efficient than the LU Algorithm. Nevertheless it is more suitable for efficient implementation on a parallel machine than the LU Algorithm.

**THEOREM 2.**

*The SP Algorithm for the solution of the tridiagonal linear system of equations (1) requires  $15n - 11$  arithmetic operations.*

**PROOF.**

*Step 1* requires  $3n$  divisions to form the  $B_i$  matrices. In *Step 2* we perform  $n - 1$  matrix multiplications to compute the  $C_i$  matrices, but because of the special structure of the matrices each matrix multiplication can be performed using 6 floating-point multiplications and 4 floating-point additions. Hence *Step 2* requires  $6(n - 1)$  multiplications and  $4(n - 1)$  additions. *Step 3* is a single division. In *Step 4* to compute all  $x_i$  for  $1 \leq i \leq n - 1$  we perform  $n - 1$  multiplications and  $n - 1$  additions. Thus the total number of arithmetic operations sums to  $15n - 11$ . •

### 3. PARALLEL PREFIX ALGORITHMS ON HYPERCUBE MULTIPROCESSORS

In this section we show that the prefix algorithm for the solution of a tridiagonal linear system of equations can be implemented efficiently on hypercube multiprocessors.

*Step 2* of the SP algorithm where the prefixes of the matrices  $(B_0, B_1, \dots, B_{n-1})$  are computed is the bottleneck point in the algorithm. An efficient parallel implementation of the recursive doubling algorithm depends on how efficiently this computation can be performed. Various parallel algorithms have been developed for prefix computation [10] [11]. The prefixes of the quantities  $(q_0, q_1, \dots, q_{n-1})$  can be computed in  $\log n$  steps<sup>†</sup> given  $n$  processors. Here each step consists of a suitably defined binary operation performed in any of the identical processors. For  $n = 8$  the parallel prefix algorithm is given in Figure 1. This algorithm is the same as the algorithms given in [10] and [18]. For simplicity we denote the product block  $q_j q_{j-1} \dots q_{i+1} q_i$  as  $ji$ . For example  $q_7 q_6 q_5 q_4$  is denoted by the pair  $74$ .

If the element  $q_i$  is initially allocated to processor  $p_i$  then at step  $k$ , for  $1 \leq k \leq \log n$ , processor  $p_i$  sends its data to processor  $p_j$  where  $j = i + 2^{k-1}$ . Processor  $p_j$  receives this data and multiplies with its own and writes the result where its data resides.

The implementation of this algorithm on a hypercube multiprocessor will be efficient only if the communication requirements of this algorithm are minimal. This requires that we map the parallel prefix algorithm efficiently on the cube. First we give a definition of a hypercube connected parallel computer:

**Definition:** *Hypercube connected parallel computer:* If  $p = 2^d$  and  $b_d \dots b_1$  is the binary representation of  $b$  for  $b \in [0, \dots, p - 1]$  and  $b^{(i)}$  is the number whose binary representation is  $b_d \dots b_{i+1} \bar{b}_i b_{i-1} \dots b_1$ , where  $\bar{b}_i$  is the complement of  $b_i$  and  $1 \leq i \leq d$  then in a hypercube connected computer, processing element  $b$  is connected to processing element  $b^{(i)}$ , for  $1 \leq i \leq d$  [15,16,17].

Now we give the definition of the binary-reflected Gray code and a lemma related to the mapping of the parallel prefix algorithm on the cube:

**Definition:** *Binary-reflected Gray code:*  $G(b) = g_d g_{d-1} \dots g_1$  of a  $d$  bit binary number  $b = b_d b_{d-1} \dots b_1$  is defined by setting [14]

$$g_i = b_i + b_{i+1} \text{ mod } 2, \quad \text{for } i = 1, 2, \dots, d - 1, \quad g_d = b_d.$$

#### LEMMA 1.

If  $b$  and  $c$  are two  $d$ -bit binary numbers such that  $0 \leq b \leq 2^d - 1 - 2^{k-1}$  and  $c = b + 2^{k-1}$  then the Hamming distance between  $G(b)$  and  $G(c)$  is 1 if  $k = 1$  and 2 if  $2 \leq k \leq d$ . Furthermore the communication paths are disjoint.

(For proof see Lemma 5.1 in [6].)

Thus we allocate the element  $q_i$  to processor  $G(i)$ . The parallel prefix algorithm requires that at step  $k$  for  $1 \leq k \leq \log n$ , the node to which element  $q_i$  is allocated should communicate with the node to which element  $q_{i+2^{k-1}}$  is allocated. The distance between nodes  $G(i)$  and  $G(i+2^{k-1})$  is 1 if  $k = 1$  and 2 if  $2 \leq k \leq \log n$ . Hence we see that by making use of the properties of a Gray code, locality is achieved at the sole expense of slightly increasing the number of routing instructions. The hypercube implementation of the parallel prefix algorithm proposed here requires at most twice the number of routing instructions of a fully-connected system implementation.

The following pseudo-code shows the required computations. This code runs in all nodes concurrently. The binary address of each node is returned when the subroutine  $node\_id()$  is called. The

<sup>†</sup> All logarithms are base 2.

subroutine  $G^{-1}(\cdot)$  converts from Gray code to binary code. For example  $G^{-1}(110) = 100$ . Initially the node  $G(i)$  contains the element  $q_i$ . This element, which is local to node  $G(i)$ , is denoted by  $Q$ . At the end of the computation node  $G(i)$  contains the product  $q_0 q_1 \cdots q_i$ . Without loss of generality we assume that  $n = 2^d$ .

```

PROCEDURE Parallel_Prefix ( n , Q )
    i = G-1 ( node_id ( ) )
    FOR k = 1 TO log n DO BEGIN
        IF i ∈ { 0 , . . . , n - 1 - 2k-1 } THEN
            SEND Q TO PROCESSOR G ( i + 2k-1 )
        IF i ∈ { 2k-1 , . . . , n - 1 } THEN
            RECEIVE temp_Q
            Q = temp_Q * Q
    END FOR
END PROCEDURE.

```

Thus we have the following lemma:

**LEMMA 2.**

*The prefixes of n elements can be computed in log n arithmetic and in 2 log n - 1 communication steps on a hypercube with n nodes.*

**PROOF.**

It follows from Lemma 2 that the first step will cost 1 arithmetic and 1 communication step. The remaining steps cost log n - 1 arithmetic and 2 ( log n - 1 ) communication steps. •

Now we suppose that we have p processors with p < n and m p = n. Then the prefixes of n elements are computed as follows: we allocate m elements to each processor and perform sequential prefix at each processor to find prefixes of these elements. Then we find prefixes of the p product blocks by performing the parallel prefix algorithm. Processor i sends this product to processor i + 1 for 0 ≤ i ≤ n - 2 and this element is multiplied with each element in the processor except the last one. Initially we allocate the elements  $q_{(i+1)m-1}, q_{(i+1)m-2}, \dots, q_{im}$  to node  $G(i)$ . These elements, which are local to node  $G(i)$ , are denoted  $Q_1, Q_2, \dots, Q_m$ . After the sequential prefix at each node we obtain a product block at each node. This result

$$Q_1 Q_2 \cdots Q_m = q_{(i+1)m-1} q_{(i+1)m-2} \cdots q_{im}$$

also resides in node  $G(i)$ . At the end of all computations the node  $G(i)$  contains the products

$$q_0 q_1 q_2 \cdots q_{(i+1)m-1}$$

$$q_0 q_1 q_2 \cdots q_{(i+1)m-1} q_{(i+1)m-2}$$

...

$$q_0 q_1 q_2 \cdots q_{(i+1)m-1} q_{(i+1)m-2} \cdots q_{im}$$

The following code shows the required computations:

```

PROCEDURE Parallel_Prefix ( n , p , Q1 , Q2 , . . . , Qm ) { limited processor case; n = m p }
    i = G-1 ( node_id ( ) )
    FOR k = 2 TO m DO BEGIN
        Qk = Qk * Qk-1
    END FOR
    FOR k = 1 TO log n DO BEGIN
        IF i ∈ { 0 , . . . , n - 1 - 2k-1 } THEN

```



```

        SEND  $Q_m$  TO PROCESSOR  $G(i + 2^{k-1})$ 
    IF  $i \in \{2^{k-1}, \dots, n-1\}$  THEN
        RECEIVE  $temp\_Q_m$ 
         $Q_m = temp\_Q_m * Q_m$ 
    END FOR
    IF  $i \in \{0, \dots, n-1-2^{k-1}\}$  THEN
        SEND  $Q_m$  TO PROCESSOR  $G(i+1)$ 
    IF  $i \in \{1, \dots, n-1\}$  THEN
        RECEIVE  $temp\_Q_m$ 
    FOR  $k=1$  TO  $m-1$  DO BEGIN
         $Q_k = temp\_Q_m * Q_k$ 
    END FOR
END PROCEDURE.
```

**LEMMA 2.**

*The prefixes of  $n = m p$  elements can be performed in  $2 \frac{n}{p} + \log p - 2$  arithmetic and  $2 \log p$  communication steps on a hypercube with  $p$  nodes.*

**PROOF.**

First we perform sequential prefix computation which costs  $m - 1$  arithmetic steps. The parallel prefix costs  $\log p$  arithmetic and  $2 \log p - 1$  communication steps according to Lemma 2. The transfer of the last element of each block to the next processor will take 1 communication step. Then we multiply this element with each element in the processor except the last one which will take  $m - 1$  arithmetic steps. Thus the total number of arithmetic and communication steps become  $2m + \log p - 2$  and  $2 \log p$ , respectively. •

In Figure 2 we illustrate the limited processor parallel prefix algorithm for the values of  $n = 12$  and  $p = 4$ . Thus it takes  $2 \log 4 = 4$  communication steps and  $2 \frac{12}{4} + \log 4 - 2 = 6$  arithmetic steps to compute prefixes of 12 terms with 4 processors.

For parallel implementation of the SP Algorithm ( henceforth called the PP Algorithm ) we allocate  $m$  matrices to each processor and perform the limited processor parallel prefix algorithm with these matrices. Considering all four steps of the SP algorithm for the solution of (1) we have the following theorem:

**THEOREM 3.**

*The PP Algorithm solves (1) with  $n = m p$  in  $35 \frac{n}{p} + 20 \log p - 29$  parallel arithmetic and  $13 \log p$  communication steps on a hypercube with  $p$  nodes.*

**PROOF.**

*Step 1* is performed in  $3m$  divisions since there are  $m$  matrices allocated to each processor.

*Step 2* has 3 substeps. In the first we perform sequential prefix at each processor. Because of the special structure of the matrices each matrix multiplication is performed with 6 multiplications and 4 additions. Hence the first substep costs  $10(m - 1)$  arithmetic operations. In the second substep of *Step 2* we perform parallel prefix using these product blocks. We lose some of the structure in the matrices involved and perform matrix multiplication using 12 multiplications and 8 additions. Thus the parallel prefix step will take  $20 \log p$  arithmetic steps. Since only the first two rows of the matrices need to be communicated, the parallel prefix step will take  $6(2 \log p - 1)$  communication steps. In the third substep of *Step 2* we first send the product block in processor  $G(i)$  to processor  $G(i + 1)$  which will cost 6 communication steps. Then we multiply this element with all the elements in the processor except the last one. This substep costs  $20(m - 1)$  arithmetic steps since the matrices are multiplied with 12 floating-point multiplications and 8 floating-point additions.

In *Step 3* processor  $p - 1$ , which holds the matrix  $C_{n-1}$ , calculates  $x_0$  by performing a single division, and then  $x_0$  is broadcast to all other processors. This operation can be performed in  $\log p$  communication steps by embedding a suitable tree of depth  $\log p$  [15,16]. In *Step 4* we calculate all  $x_i$  by performing  $m$  multiplications and  $m$  additions per processor. The total result follows by summing the number of arithmetic operations and communication steps. •

Step	Arithmetic Complexity	Communication Complexity
1	$3 m$	-
2	$30 ( m - 1 ) + 20 \log p$	$12 \log p$
3	1	$\log p$
4	$2 m$	-
Total	$35 m + 20 \log p - 29$	$13 \log p$

Finally it is interesting to observe that an SIMD system with processor masking capability is adequate for the algorithm although in actual experiments we used the Intel iPSC/d5 which is an MIMD system.

## 5. ESTIMATED SPEED-UP AND EFFICIENCY

The speed-up and efficiency of the PP Algorithm with respect to the LU and the SP Algorithms can be estimated using the arithmetic and communication complexity figures found previously. We have performed experiments, similar to those mentioned in [12], on the Intel iPSC/d5 hypercube running XENIX 286 R3.4 and iPSC Software R3.1 to measure the time it takes to perform a floating-point operation ( $\tau_{comp}$ ), and the time it takes to transfer a floating-point number to an adjacent node ( $\tau_{comm}$ ). The experiments indicated that  $\tau_{comp} \approx 0.058$  milliseconds for floating-point multiplication, and  $\tau_{comm} \approx 1.48$  milliseconds. Using these we can estimate the speed-up of the PP Algorithm with respect to the LU and the SP Algorithms as

$$S_{PP/LU} = \frac{T_{LU}}{T_{PP}} = \frac{(8n - 7) \tau_{comp}}{(35 \frac{n}{p} + 20 \log p - 29) \tau_{comp} + (13 \log p) \tau_{comm}},$$

$$S_{PP/SP} = \frac{T_{SP}}{T_{PP}} = \frac{(15n - 11) \tau_{comp}}{(35 \frac{n}{p} + 20 \log p - 29) \tau_{comp} + (13 \log p) \tau_{comm}}.$$

Similarly the efficiency of the PP Algorithm with respect to the LU and the SP Algorithms is found as

$$E_{PP/LU} = \frac{S_{PP/LU}}{p} = \frac{(8n - 7) \tau_{comp}}{(35n + 20p \log p - 29p) \tau_{comp} + (13p \log p) \tau_{comm}},$$

$$E_{PP/SP} = \frac{S_{PP/SP}}{p} = \frac{(15n - 11) \tau_{comp}}{(35n + 20p \log p - 29p) \tau_{comp} + (13p \log p) \tau_{comm}}.$$

The results are shown in Table 1 for the value of  $p = 32$ .

## 6. EXPERIMENTAL RESULTS AND CONCLUSIONS

We have experimented on an Intel iPSC/d5 hypercube system for the values of  $n$  between 32 and 8192. The LU and the SP algorithms were run on a single node and the PP algorithm was run on 1, 2, 3, 4 and 5 dimensional subcubes. The initial loading of the data was not taken into account for any of these algorithms. The experiments were done to compute the cubic spline approximation of some random data. The types of tridiagonal matrices that arise in cubic spline approximation are diagonally dominant and

mostly symmetric [1]. It has been shown that some stability problems arise in the use of the recursive doubling algorithm when the size of system is large [3]. Since the size of memory on the Intel iPSC/d5 is about 300 kilobytes/node, experimentation was kept to tridiagonal systems of size no more than 8192.

The computation and communication time were measured using the *clock( )* routine at the beginning and end of each program. The timings of the LU, the SP, and PP algorithms are given in Table 2 in milliseconds. Using these data we can compute the measured speed-up and efficiency of the PP Algorithm with respect to its sequential counterparts. These are shown in Table 3 for the value of  $p = 32$  (compare Table 1 to Table 3). Also, in Figures 3a, and 3b we show the estimated and measured efficiency of the PP Algorithm with respect to LU algorithm as a function of dimension of the cube for values of  $n = 4096$  and  $n = 8192$ , respectively. Similarly, the PP Algorithm is compared to the SP Algorithm in Figures 4a and 4b. The small differences between the estimated and measured values are due to the fact that we assumed all floating-point operations take the same amount of time, and also overhead factors, such as loop control, memory fetch etc. were not taken into account.

The experimental results have shown the proposed algorithm achieves linear speed-up and its efficiency is somewhere between 0.50 and 0.60.

## REFERENCES

1. AHLBERG, J. H., NILSON, E. N. and WALSH, J. L. *The Theory of Splines and their Applications*, Academic Press, 1967.
2. DONGARRA, J.J., BUNCH, J. R., MOLER, C. B., and STEWART, G.W. *Linpack Users' Guide*, SIAM, Philadelphia, 1979.
3. DUBOIS, P. and RODRIGUE, G. An analysis of the recursive doubling algorithm, in *High Speed Computer and Algorithm Organization*, edited by D. J. Kuck, D. H. Lawrie and A. H. Sameh, pp. 299-305, Academic Press, 1977.
4. HELLER, D. A survey of parallel algorithms in numerical linear algebra, *SIAM Review*, pp. 740-777, October 1978
5. JOHNSON, S. L. Band matrix systems solvers on ensemble architecture, in *Supercomputers: Algorithms, Architectures, and Scientific Computation*, edited by F. A. Matsen and T. Tajima, pp. 196-216, University of Texas Press, Austin, 1986.
6. JOHNSON, S. L. Solving tridiagonal systems on ensemble architectures, *SIAM Journal on Scientific and Statistical Computing*, Vol. 8, No. 3, pp. 354-392, May 1987.
7. JOHNSON, S. L. Communication efficient basic linear algebra computations on hypercube multiprocessors, *Journal of Parallel and Distributed Computing*, No. 4, pp. 133-172, 1987.
8. JOHNSON, S. L. and HO, C. T. Multiple tridiagonal systems, the alternating direction methods and boolean cube configured multiprocessors, Research Report, Yale University, YALEU/DCS/RR-532, June 1987.
9. KOGGE, P. M. and STONE, H. S. A parallel algorithm for the efficient solution of a general class of recurrence equations, *IEEE Transactions on Computers*, Vol. C-22, No. 8, pp. 786-793, August 1973.
10. KRUSKAL, C. P., RUDOLPH, L. and SNIR, M. The power of parallel prefix, *IEEE Transactions on Computers*, Vol. C-34, No. 10, pp. 965-968, October 1985.
11. LADNER, R. and FISCHER, M. Parallel prefix computation, *Journal of ACM*, Vol. 27, No. 4, pp. 831-838, October 1980.
12. MCBRYAN, O. A. and VAN DE VELDE, E. F. Hypercube algorithms and implementations, *SIAM Journal on Scientific and Statistical Computing*, Vol. 8, No. 2, pp. s227-s287, March 1987.
13. ORTEGA, J. and VOIGT, R. Partial differential equations on vector and parallel computers, *SIAM Review*, pp. 149-240, June 1985.
14. REINGOLD, E. M., NIEVERGELT, J. and DEO, N. *Combinatorial Algorithms: Theory and Practice*, pp. 173-179, Prentice-Hall, 1977.

15. SAAD, Y. and SCHULTZ, M. H. Data communication in hypercubes, Research Report, Yale University, YALEU/DCS/RR-428, October 1985.
16. SAAD, Y. and SCHULTZ, M. H. Topological properties of hypercubes, Research Report, Yale University, YALEU/DCS/RR-389, June 1985.
17. SEITZ, C. L. The cosmic cube, *Communications of the ACM*, Vol. 28, No. 1, pp. 22-33, January 1985.
18. STONE, H. S. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations, *Journal of ACM*, Vol. 20, No. 1, pp. 27-38, January 1973.
19. STONE, H. S. Parallel tridiagonal equation solvers, *ACM Transactions on Mathematical Software*, Vol. 1, No. 4, pp. 289-307, December 1975.

**Table 1.** Estimated speed-up and efficiency for  $p = 32$

$n$	$S_{PP/LU}$	$S_{PP/SP}$	$E_{PP/LU}$	$E_{PP/SP}$
32	0.14	0.27	0.004	0.008
64	0.28	0.53	0.009	0.017
128	0.54	1.02	0.017	0.032
256	1.02	1.91	0.032	0.060
512	1.79	3.35	0.056	0.105
1024	2.87	5.39	0.090	0.168
2048	4.13	7.74	0.129	0.242
4096	5.28	9.89	0.165	0.309
8192	6.13	11.49	0.192	0.359

**Table 2.** The timings of the LU, SP and PP Algorithms (in milliseconds)

$n$	LU	SP	$PP$ $p = 2$	$PP$ $p = 4$	$PP$ $p = 8$	$PP$ $p = 16$	$PP$ $p = 32$
32	15	40	40	30	25	25	75
64	30	75	80	45	35	60	85
128	60	155	155	85	55	65	90
256	120	315	310	160	95	80	100
512	235	625	615	315	165	125	120
1024	480	1250	1225	620	320	210	150
2048	960	2495	2445	1230	625	370	230
4096	1920	4990	4885	2450	1235	655	400
8192	3840	9990	9775	4895	2455	1260	685

**Table 3.** Measured speed-up and efficiency for  $p = 32$

$n$	$S_{PP/LU}$	$S_{PP/SP}$	$E_{PP/LU}$	$E_{PP/SP}$
32	0.20	0.53	0.006	0.017
64	0.35	0.88	0.011	0.028
128	0.67	1.72	0.021	0.054
256	1.20	3.15	0.038	0.098
512	1.96	5.21	0.061	0.163
1024	3.20	8.33	0.100	0.260
2048	4.17	10.85	0.130	0.339
4096	4.80	12.47	0.150	0.390
8192	5.61	14.58	0.175	0.456