Journal of Computational and Applied Mathematics 27 (1989) 95–108
North-Holland

A recursive doubling algorithm for solution of tridiagonal systems on hypercube multiprocessors*

Ömer EĞECIOĞLU**
Department of Computer Science, University of California, Santa Barbara, CA 93106, U.S.A.

Cetin K. KOC*** and Alan J. LAUB***
Scientific Computation Laboratory, Department of Electrical & Computer Engineering, University of California, Santa Barbara, CA 93106, U.S.A.

Received 25 January 1988
Revised 15 June 1988

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 $O(n/p + \log p)$ 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 speedup 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.

Keywords: Tridiagonal systems, parallel algorithm, parallel prefix, hypercube multiprocessor.

1. Introduction

We are interested in solving the following system of linear equations

$$Ax = d, \tag{1}$$

* Technical Report No. TRCS88-1, Department of Computer Science, University of California, Santa Barbara, January 1988. 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.

** Supported in part by NSF Grant No. DCR-8603722.
*** Supported in part by Lawrence Livermore National Laboratory Contract No. LLNL-7526225 and the National Science Foundation (and AFOSR) under Grant No. ECS87-18897.

where A is a (nonsymmetric) tridiagonal matrix of order n

$$A = \begin{bmatrix} b_0 & c_0 & & & & \\ a_1 & b_1 & c_1 & & & \\ & a_2 & b_2 & c_2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-2} & b_{n-2} & c_{n-2} \\ & & & & a_{n-1} & b_{n-1} \end{bmatrix}$$

and x and d are vectors of dimension n

$$x = (x_0, x_1, \ldots, x_{n-2}, x_{n-1})^{\mathrm{T}},$$

$$d = (d_0, d_1, \ldots, d_{n-2}, d_{n-1})^{\mathrm{T}}.$$

We shall assume that A, x, and d have real coefficients. Extension to the complex case is straightforward.

Tridiagonal systems of equations appear frequently in the solution of partial differential equations, cubic spline interpolation, and in numerous other areas of science and engineering. There has been a considerable amount of work to solve (1) on parallel computers; see, for example, the review articles [5,15,22]. More recently Johnsson et al. have developed algorithms to solve such systems on ensemble architectures [6–9]. The recursive doubling algorithm is one of the first algorithms that has resulted from considering parallelism in computation. This approach relates the LDU decomposition of A to first- and second-order linear recurrences. The well-known relationship between (1) and linear recurrences was utilized by Stone to develop an algorithm to solve (1) in $O(\log n)$ parallel arithmetic steps[1] with n processors [21]. This algorithm can be generalized to solve banded linear systems as well [10].

The recursive doubling algorithm is suitable when a large number of processing elements are available, such as the Connection Machine. In this paper we give a limited processor version of the recursive doubling algorithm on hypercube multiprocessor architectures with $p < n$ processors. This algorithm is more suitable for hypercubes of smaller dimension such as the Caltech Hypercube, the Intel iPSC series, and the NCUBE. We show that the limited processor version recursive doubling algorithm solves a tridiagonal system of size n with arithmetic complexity $O(n/p + \log p)$ and communication complexity $O(\log p)$ on a hypercube multiprocessor with p processors. The algorithm becomes more efficient if $p \ll n$. The main techniques rely on fast parallel prefix algorithms for which we describe an efficient mapping using the binary-reflected Gray code. These techniques can also be extended to solve banded or block tridiagonal linear systems.

We compare the algorithm proposed here to the LU decomposition algorithm and to a sequential version of the recursive doubling algorithm. The theoretical estimates for speedup and efficiency, as well as the experimental results on an Intel iPSC/d5 hypercube multiprocessor indicate that the limited processor recursive doubling algorithm achieves linear speedup and its efficiency is more than 0.5.

[1] All logarithms are base 2.

2. The LU decomposition algorithm

One of the most efficient existing sequential algorithms for solving (1) relies on the LU decomposition of A; see, for example, [2]. Here A is decomposed into a product of two bidiagonal matrices L and U as follows:

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

The algorithm then proceeds to solve for y from $Ly = d$ and then finds x by solving $Ux = y$. 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 A given by

$$f_0 = b_0,$$
$$e_i = a_i / f_{i-1}, \quad 1 \leqslant i \leqslant n-1,$$
$$f_i = b_i - e_i * c_{i-1}, \quad 1 \leqslant i \leqslant n-1.$$

Step 2. Solve for y from $Ly = d$ using

$$y_0 = d_0,$$
$$y_i = d_i - e_i * y_{i-1}, \quad 1 \leqslant i \leqslant n-1.$$

Step 3. Compute x by solving $Ux = y$ using

$$x_{n-1} = y_{n-1} / f_{n-1},$$
$$x_i = (y_i - c_i * x_{i+1}) / f_i, \quad 0 \leqslant i \leqslant n-2.$$

By counting the number of operations at each step, we see that the LU algorithm solves a tridiagonal linear system of size n using $8n - 7$ arithmetic operations.

3. Solution of tridiagonal systems using prefix algorithms

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

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i \quad \text{for } 1 \leqslant i \leqslant n-2 \tag{2}$$

with

$$b_0 x_0 + c_0 x_1 = d_0, \qquad a_{n-1} x_{n-2} + b_{n-1} x_{n-1} = d_{n-1}.$$

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

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

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

(3) can be rewritten as

$$x_{i+1} = \alpha_i x_i + \beta_i x_{i-1} + \gamma_i \quad \text{for } 0 \leqslant i \leqslant n-1.$$

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 [21]. 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

$$X_{i+1} = B_i X_i \quad \text{for } 0 \leqslant i \leqslant n-1. \tag{4}$$

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

$$X_0 = \begin{bmatrix} x_0 \\ x_{-1} \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \\ 0 \\ 1 \end{bmatrix}, \tag{5}$$

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

$$\begin{aligned} X_1 &= B_0 X_0, \\ X_2 &= B_1 X_1 = B_1 B_0 X_0, \\ &\vdots \\ X_n &= B_{n-1} B_{n-2} \cdots B_1 B_0 X_0. \end{aligned}$$

Now let

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

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

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

The g_{ij} depend on α_i, β_i, γ_i for $0 \leqslant i \leqslant n-1$. 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

$$x_0 = -g_{02}/g_{00}. \tag{6}$$

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

The sèquential prefix algorithm (the SP algorithm) to solve the tridiagonal (1) thus proceeds as follows.

The SP Algorithm

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

$$\alpha_i = -\frac{b_i}{c_i}, \qquad \beta_i = -\frac{a_i}{c_i}, \qquad \gamma_i = \frac{d_i}{c_i} \quad \text{and} \quad B_i = \begin{bmatrix} \alpha_i & \beta_i & \gamma_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Step 2. Compute the chain products C_i by

$$C_0 = B_0,$$
$$C_i = B_i C_{i-1}, \quad 1 \leqslant i \leqslant n-1.$$

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 (5) and (6).

Step 4. Compute X_i and hence x_i using

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

Step 2 of this algorithm essentially calculates the prefixes of the matrices $(B_0, B_1, B_2, \ldots, B_{n-1})$ (here we imagine that the matrix products are 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 1. *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 matrices B_i. In Step 2 we perform $n-1$ matrix multiplications to compute the C_i, 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 $1 \leqslant i \leqslant n-1$ we perform $n-1$ multiplications and $n-1$ additions. Thus the total number of arithmetic operations sums to $15n - 11$. □

4. 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 $(B_0, B_1, \ldots, B_{n-1})$ 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 [11,12]. The prefixes of the quantities $(q_0, q_1, \ldots, q_{n-1})$ can be computed in log n steps 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 Fig. 1. This algorithm is the same as the algorithms given in [11] and [21]. For simplicity we denote the product block $q_j q_{j-1} \cdots q_{i+1} q_i$ as $j\ i$. For example $q_7 q_6 q_5 q_4$ is denoted by the pair 7 4.

If the element q_i is initially allocated to processor p_i, then at step k, for $1 \leqslant k \leqslant \log n$, 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 the 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.

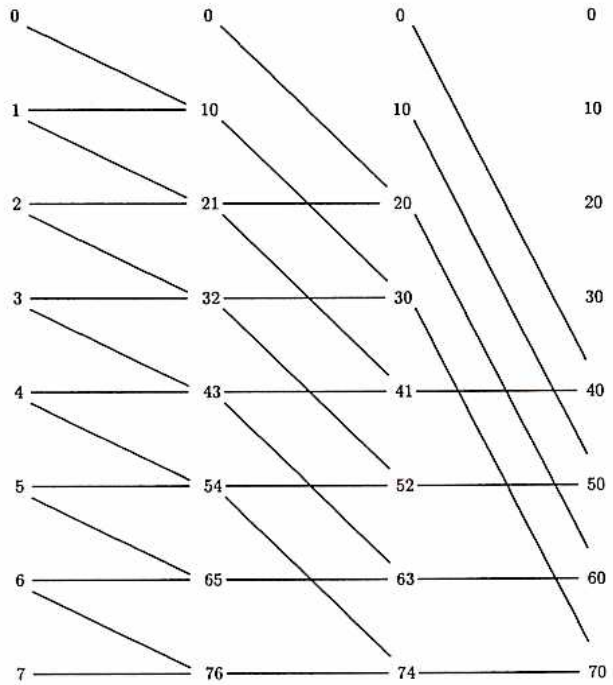

Fig. 1. The parallel prefix algorithm for $n = 8$.

Definition 1. *Hypercube connected parallel computer.* If $p = 2^d$ and $b_d \cdots b_1$ is the binary representation of b for $b \in [0, \ldots, p-1]$ and $b^{(i)}$ is the number whose binary representation is $b_d \cdots b_{i+1}\bar{b}_i b_{i-1} \cdots b_1$, where $\bar{b}_i$ is the complement of b_i and $1 \leqslant i \leqslant d$, then in a hypercube connected computer, processing element b is connected to processing element $b^{(i)}$, for $1 \leqslant i \leqslant d$ [18–20].

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 2. *Binary-reflected Gray code* $G(b) = g_d g_{d-1} \cdots g_1$ of a d-bit binary number $b = b_d b_{d-1} \cdots b_1$ is defined by setting [16]

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

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

For a proof see [7, Lemma 5.1].

Thus we allocate the element q_i to processor $G(i)$. The parallel prefix algorithm requires that at step k for $1 \leqslant k \leqslant \log n$, 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 $2 \leqslant k \leqslant \log n$. 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 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 $q_i q_{i-1} \cdots q_0$. 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 − 2^{k−1}} then
            send Q to processor G(i + 2^{k−1})
        if i ∈ {2^{k−1}, ..., n − 1} then
            receive temp_Q
        Q = temp_Q * Q
    end for
end Procedure.
```

Thus we observe that 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. This follows from Lemma 1 since the first step will cost 1 arithmetic and 1 communication step and 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 $mp = n$. 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 $0 \leqslant i \leqslant n-2$ and this element is multiplied with each element in the processor except the last one. Initially we allocate the elements $q_{(i+1)m-1}, q_{(i+1)m-2}, \ldots, q_{im}$ to node $G(i)$. These elements, which are local to node $G(i)$, are denoted $Q_m, Q_{m-1}, \ldots, Q_1$. After the sequential prefix at each node we obtain a product block at each node. This result

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

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

$$\begin{aligned} &q_{im} \cdots q_1 q_0, \\ &\vdots \\ &q_{(i+1)m-2} \cdots q_{im} \cdots q_1 q_0, \\ &q_{(i+1)m-1} q_{(i+1)m-2} \cdots q_{im} \cdots q_1 q_0. \end{aligned}$$

The following code shows the required computations:

```
Procedure Parallel_Prefix (n, p, Q_1, Q_2, ..., Q_m) { limited processor case; n = mp }
    i = G^{-1}(node_id())
    for k = 2 to m do begin
        Q_k = Q_k * Q_{k-1}
    end for
    for k = 1 to log n do begin
        if i ∈ {0, ..., n - 1 - 2^{k-1}} then
            send Q_m to processor G(i + 2^{k-1})
        if i ∈ {2^{k-1}, ..., n - 1} then
            receive temp_Q_m
        Q_m = temp_Q_m * Q_m
    end for
    if i ∈ {0, ..., n - 1 - 2^{k-1}} then
        send Q_m to processor G(i + 1)
    if i ∈ {1, ..., 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.
```

An inspection of the above algorithm shows that the prefixes of $n = mp$ elements can be computed in $2n/p + \log p - 2$ arithmetic and $2 \log p$ communication steps on a hypercube with

p nodes. 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 as we remarked earlier. The transfer of the last element of each block to the next processor will take 1 communication step. Then we multiply this elements 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 Fig. 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 4 steps of the SP algorithm for the solution of (1) we have the following theorem.

Theorem 2. *The PP algorithm solves* (1) *with* $n = mp$ *in* $35n/p + 20\log p - 29$ *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

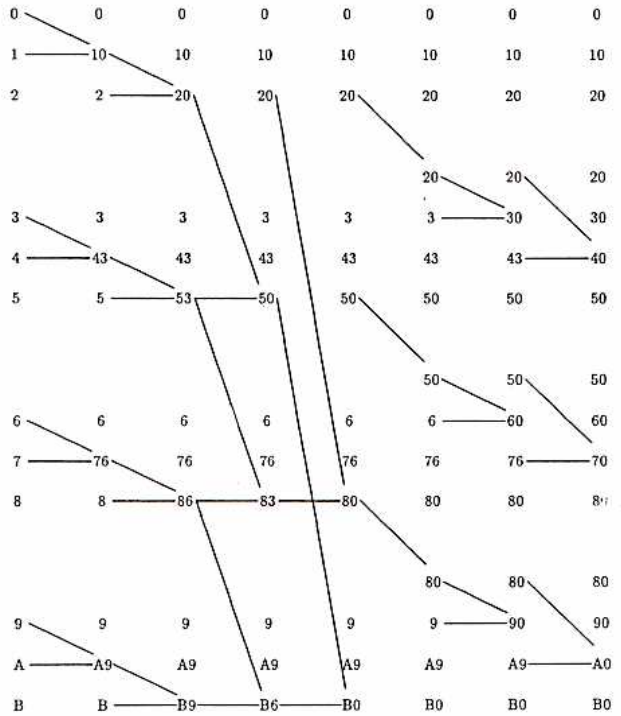

Fig. 2. The limited processor version of the parallel prefix algorithm for $n = 12$ and $p = 4$.

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 $\log p$ [18,19]. 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	$3m$	—
2	$30(m-1) + 20 \log p$	$12 \log p$
3	1	$\log p$
4	$2m$	—
Total	$35m + 20 \log p - 29$	$13 \log p$

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 speedup and efficiency

The speedup 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 [13], on an 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 (τ_{comp}), and the time it takes to transfer a floating-point number to an adjacent node (τ_{comm}). The experiments indicated that $\tau_{\text{comm}} \approx 1.48$ milliseconds, and if the floating-point operation is taken to be multiplication, addition, or subtraction then $\tau_{\text{comp}} \approx 0.058$ milliseconds. Division takes a little longer (around 0.072 milliseconds). Using these we can estimate the speedup of the PP algorithm with respect to the LU and SP algorithms as

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

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

Table 1
Estimated speedup 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

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 $p = 32$ for the values of $\tau_{comp} = 0.058$ and $\tau_{comm} = 1.48$. The efficiency of the PP algorithm with respect to its sequential counterparts is a function of the ratio, $\tau = \tau_{comm}/\tau_{comp}$, for fixed values of n and p. For the Intel cube we have $\tau = 25.51$. Given $n = 8192$ and $p = 32$, we see that $E_{PP/SP}$ takes the values of 0.359. $E_{PP/SP}$ will take values between 0.422 and 0.247 as τ takes values between 1 and 100. This ratio is a crucial parameter in message-passing parallel computers, and its value changes between 2 and 1000 for different hypercubes (see Gordon Bell on the Future of Computers, *SIAM News*, Vol. 20, No. 2, March 1987).

6. Experimental results and conclusions

We have experimented on an Intel iPSC/d5 hypercube system for values of n between 32 and 8192. The LU and 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].

The computation and communication times were measured using the *clock()* routine at the beginning and end of each program. The timings of the LU, SP, and PP algorithms are given in Table 2 in milliseconds. Using these data we can compute the measured speedup 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 Figs. 3(a) and 3(b) we show the

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

estimated and measured efficiency of the PP algorithm with respect to the 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 Figs. 4(a) and 4(b). 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 achieved linear speedup and its efficiency is somewhere between 0.50 and 0.60.

It has been observed that some numerical stability problems can arise in the use of the recursive doubling algorithm for certain classes of problems when the size of the system is large [3]. Since memory size on the Intel iPSC/d5 is about 300 kilobytes/node, experimentation was kept to tridiagonal systems of size no more than 8192. In attaching high importance to speed, numerical stability problems involved in the use of parallel algorithms are occasionally ignored. As pointed out in [14], a parallel algorithm may become completely useless if its numerical stability properties are undesirable. Methods for analyzing the numerical stability of parallel algorithms have been developed in [17] and classification schemes have been proposed in [4] based on the theoretical foundations of *forward error analysis* [23].

Table 3
Measured speedup 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

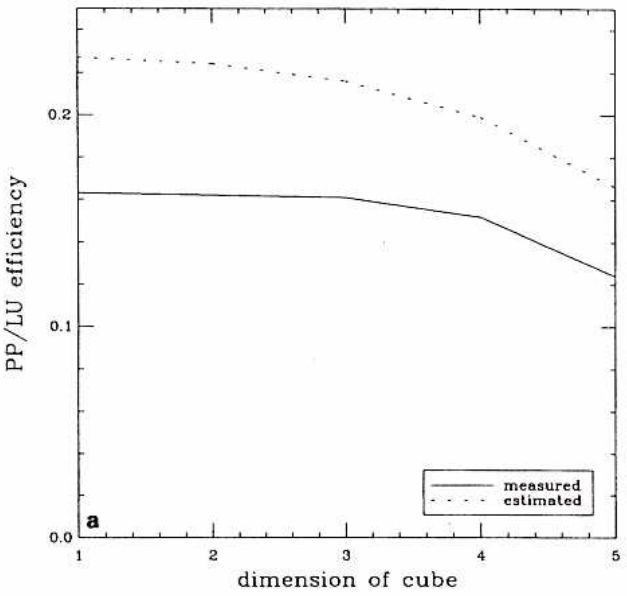

Fig. 3(a). Efficiency of the PP algorithm with respect to the LU algorithm as a function of cube dimension for $n = 4096$.

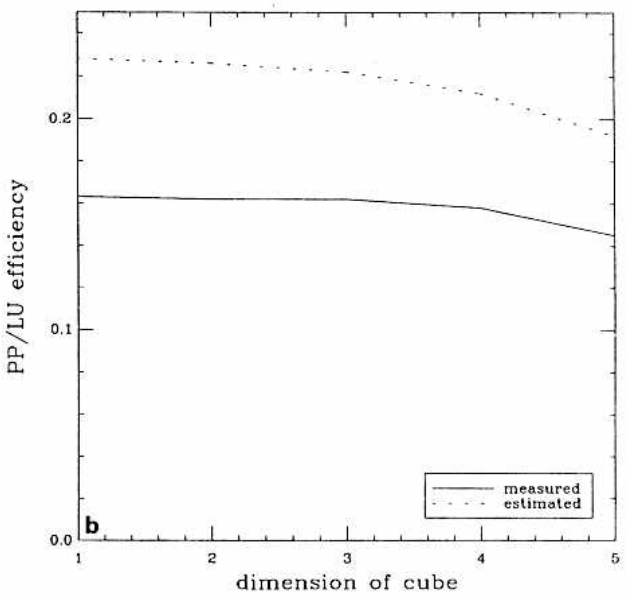

Fig. 3(b). Efficiency of the PP algorithm with respect to the LU algorithm as a function of cube dimension for $n = 8192$.

Solution of tridiagonal or banded systems has been done using (block) Gaussian elimination, (block) cyclic reduction [6] and the recursive doubling algorithm. In [4] Gao has applied the forward error analysis technique to pipelined algorithms for the solution of first- and second-order

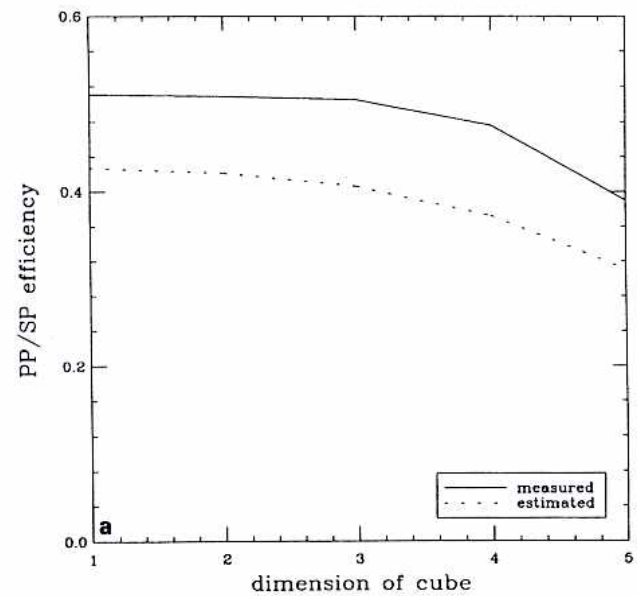

Fig. 4(a). Efficiency of the PP algorithm with respect to the SP algorithm as a function of cube dimension for $n = 4096$.

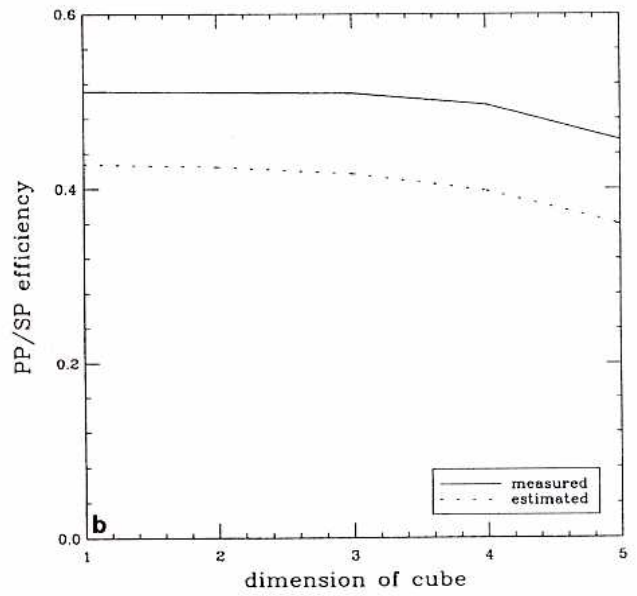

Fig. 4(b). Efficiency of the PP algorithm with respect to the SP algorithm as a function of cube dimension for $n = 8192$.

recurrences. We are currently implementing parallel algorithms for the solutions of general recurrence relations, tridiagonal, block tridiagonal, and banded linear systems on the Intel hypercube, and investigating their numerical stability properties.

References

[1] J.H. Ahlberg, E.N. Nilson and J.L. Walsh, *The Theory of Splines and their Applications* (Academic Press, New York, 1967).

[2] J.J. Dongarra, J.R. Bunch, C.B. Moler and G.W. Stewart, *Linpack Users' Guide* (SIAM, Philadelphia, PA, 1979).

[3] P. Dubois and G. Rodrigue, An analysis of the recursive doubling algorithm, in: D.J. Kuck, D.H. Lawrie and A.H. Sameh, Eds., *High Speed Computer and Algorithm Organization* (Academic Press, New York, 1977) 299–305.

[4] G.R. Gao, A stability classification method and its application to pipelined solution of linear recurrences, *Parallel Comput.* **4** (3) (1987) 305–321.

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

[6] S.L. Johnsson, Band matrix systems solvers on ensemble architecture, in: F.A. Matsen and T. Tajima, Eds., *Supercomputers: Algorithms, Architectures, and Scientific Computation* (University of Texas Press, Austin, 1986) 196–216.

[7] S.L. Johnsson, Solving tridiagonal systems on ensemble architectures, *SIAM J. Sci. Statist. Comput.* **8** (3) (1987) 354–392.

[8] S.L. Johnsson, Communication efficient basic linear algebra computations on hypercube multiprocessors, *J. Parallel Distr. Comput.* **4** (1987) 133–172.

[9] S.L. Johnsson and C.T. Ho, Multiple tridiagonal systems, the alternating direction methods, and Boolean cube configured multiprocessors, Research Report, Yale University, YALEU/DCS/RR-532, 1987.

[10] P.M. Kogge and H.S. Stone, A parallel algorithm for the efficient solution of a general class of recurrence equations, *IEEE Trans. Comput.* **C-22** (8) (1973) 786–793.

[11] C.P. Kruskal, L. Rudolph and M. Snir, The power of parallel prefix, *IEEE Trans. Comput.* **C-34** (10) (1985) 965–968.

[12] R. Ladner and M. Fischer, Parallel prefix computation, *J. Assoc. Comput. Mach.* **27** (4) (1980) 831–838.

[13] O.A. McBryan and E.F. van de Velde, Hypercube algorithms and implementations, *SIAM J. Sci. Statist. Comput.* **8** (2) (1987) s227–s287.

[14] W. Miller, Computational complexity and numerical stability, *SIAM J. Comput.* **4** (2) (1975) 97–107.

[15] J. Ortega and R. Voigt, Partial differential equations on vector and parallel computers, *SIAM Rev.* (1985) 149–240.

[16] E.M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice* (Prentice-Hall, Englewood Cliffs, NJ, 1977) 173–179.

[17] W. Ronsch, Stability aspects in using parallel algorithms, *Parallel Comput.* **1** (1) (1984) 75–98.

[18] Y. Saad and M.H. Schultz, Data communication in hypercubes, Research Report, Yale University, YALEU/DCS/RR-428, 1985.

[19] Y. Saad and M.H. Schultz, Topological properties of hypercubes, Research Report, Yale University, YALEU/DCS/RR-389, 1985.

[20] C.L. Seitz, The cosmic cube, *Comm. ACM* **28** (1) (1985) 22–23.

[21] H.S. Stone, An efficient parallel algorithm for the solution of a tridiagonal linear system of equations, *J. Assoc. Comput. Mach.* **20** (1) (1973) 27–38.

[22] H.S. Stone, Parallel tridiagonal equation solvers, *ACM Trans. Math. Software* **1** (4) (1975) 289–307.

[23] F. Stummel, Perturbation theory for evaluation algorithms of arithmetic expressions, *Math. Comp.* **37** (156) (1981) 435–473.