Printing Requisition / Graphic Services

54231

1. Please complete unshaded areas on form as applicable.
2. Distribute copies as follows: White and Yellow to Graphic Services. Retain Pink Copies for your records.
3. On completion of order the Yellow copy will be returned with the printed material.
4. Please direct enquiries, quoting requisition number and account number, to extension 3451.

TITLE OR DESCRIPTION

Updating and Downdating the Inverse Cholesky Factor on a Hypercube Multiprocessor CS-88-46

DATE REQUISITIONED	DATE REQUIRED	ACCOUNT NO.
Feb. 1/89	ASAP	4 1 2 4 1 0 0 6 0

REQUISITIONER – PRINT: J.A. George

PHONE:

SIGNING AUTHORITY: J. Allan George

MAILING INFO – NAME: Sue DeAngelis | DEPT.: C.S. | BLDG. & ROOM NO.: DC 2314 | [X] DELIVER [ ] PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 29 NUMBER OF COPIES: 50

TYPE OF PAPER STOCK: [X] BOND [ ] NCR ___ PT. [X] COVER [ ] BRISTOL [X] SUPPLIED [ ] ___

PAPER SIZE: [X] 8½ x 11 [ ] 8½ x 14 [ ] 11 x 17 [ ] ___

PAPER COLOUR: [X] WHITE [ ] ___ INK: [X] BLACK [ ] ___

PRINTING: [ ] 1 SIDE ___ PGS. [X] 2 SIDES ___ PGS. NUMBERING: FROM ___ TO ___

BINDING/FINISHING: 3 down left side
[X] COLLATING [X] STAPLING [ ] ___ ROLL PUNCHED [ ] PLASTIC RING

FOLDING/PADDING: CUTTING SIZE:

Special Instructions

Math fronts and backs enclosed.

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

				D 0 1
				D 0 1
				D 0 1

TYPESETTING — QUANTITY

P A P 0 0 0 0 0				T 0 1
P A P 0 0 0 0 0				T 0 1
P A P 0 0 0 0 0				T 0 1

PROOF

P R F				
P R F				
P R F				

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE

F L M				C 0 1
F L M				C 0 1
F L M				C 0 1
F L M				C 0 1
F L M				C 0 1

PMT

P M T				C 0 1
P M T				C 0 1
P M T				C 0 1

PLATES

P L T				P 0 1
P L T				P 0 1
P L T				P 0 1

STOCK

				0 0 1
				0 0 1
				0 0 1
				0 0 1

BINDERY

R N G				B 0 1
R N G				B 0 1
R N G				B 0 1
M I S 0 0 0 0 0				B 0 1

OUTSIDE SERVICES

$ ___ COST

TAXES – PROVINCIAL [ ] FEDERAL [ ]

To

From

Date

memo

University of Waterloo

Jan. 30

Sue,
Please make 50 copies
of report CS-88-46.
Thanks.
Eleanor

sent print req.
to Alan
Make copy for
Zoe's file

To file

From

Date Feb. 10/89

memo

University of Waterloo

Sue,
I took 6 copies
of our report CS-88-46.

Thks.

Eleanor

Updating and Downdating the Inverse Cholesky Factor on a Hypercube Multiprocessor

Eleanor Chu
Alan George

Department of Computer Science

Research Report CS-88-46
December 1988

Faculty of Mathematics

University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Updating and Downdating the Inverse Cholesky Factor on a Hypercube Multiprocessor

Eleanor Chu
Alan George

Department of Computer Science

Research Report CS-88-46
December 1988

Updating and Downdating the Inverse Cholesky Factor on a Hypercube Multiprocessor. *

Eleanor Chu
Alan George

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Research Report CS-88-46

December 1988

Abstract

This article describes a new hypercube implementation of the two sequential algorithms recently proposed by Pan and Plemmons for updating and downdating the inverse Cholesky factor in the context of modifying least squares solutions. The two algorithms update/downdate the inverse Cholesky factor R^{-1} by applying the same sequence of orthogonal/hyperbolic rotations used to update/downdate the Cholesky factor R. Since the triangular solve required in the algorithms for updating/downdating R is avoided in this pair of new algorithms by working with the inverse of R instead, they have been regarded as good candidates for parallel implementation on a hypercube multiprocessor. Furthermore, a straightforward hypercube implementation with communication volume of $O(n \log_2 p)$ is readily available by employing the data mapping strategy known for similar computations, where n is the dimension of the factor R^{-1} and $p = 2^d$ represents the number of processors in the hypercube network.

The new hypercube implementation we propose in this paper achieves a communication volume of $O(n)$ while maintaining a balanced work load distribution among the processors. Since we lower the communication volume by sending more short messages, a desired data mapping strategy will not only balance the work load but also induce an appropriate precedence relationship on the data so that the communication can be masked by computation. We shall describe how the dual objectives can be achieved by taking full advantage of the hypercube connectivity and a novel recursive data partitioning strategy.

*Research supported in part by NASA Grant No. NAG-1-803 and by the University of Waterloo.

Contents

List of Figures

1 Background

In this article we study the problem of updating and downdating the inverse Cholesky factor R^{-1} on a distributed-memory multiprocessor in the context of modifying least squares solutions. In particular, we are concerned with *parallelizing* the two sequential algorithms recently developed in [5] for updating (and downdating) R^{-1}. Both algorithms enjoy the following features. First, highly serial triangular solves, which are considered the bottleneck in updating/downdating the Cholesky factor R on parallel computers [3], can be avoided entirely by working with R^{-1} [5]. As pointed out by Pan and Plemmons, this feature is particularly desirable in the application area of signal processing, where an inverse orthogonal factorization of the Toeplitz autocorrelation matrix T results in the initial R^{-1} being readily available to start the updating/downdating processes. Developing fast algorithms for Toeplitz factorization has been the topic of recent work by Cybenko [2] and Chun et. al. [1]. The second feature results from the following two theorems, namely that the sequence of orthogonal (hyperbolic) rotations used to update (downdate) the Cholesky factor R can also be used to update (downdate) its inverse, R^{-1}.

Theorem 1 [5] *Let R denote an $n \times n$ nonsingular upper triangular matrix and let y denote an n-vector. If Q denotes the product of a sequence of plane rotations used to solve the updating problem for R and y^T, i. e. if*

$$Q \begin{pmatrix} R \\ y^T \end{pmatrix} = \begin{pmatrix} U \\ 0^T \end{pmatrix},$$

where U is upper triangular, then

$$Q \begin{pmatrix} R^{-T} \\ 0^T \end{pmatrix} = \begin{pmatrix} U^{-T} \\ u^T \end{pmatrix},$$

where u is given by

$$u = -\frac{R^{-1}a}{\delta}$$

with $a = R^{-T}y$ and $\delta = \sqrt{1 + a^T a}$.

Theorem 2 [5] *Let R denote an $n \times n$ nonsingular upper triangular matrix and let z denote an n-vector. If H is pseudo-orthogonal with respect to $S = diag\,(I_n, -1)$ and*

$$H \begin{pmatrix} R \\ z^T \end{pmatrix} = \begin{pmatrix} D \\ 0^T \end{pmatrix},$$

where D is upper triangular, then

$$H \begin{pmatrix} R^{-T} \\ 0^T \end{pmatrix} = \begin{pmatrix} D^{-T} \\ v^T \end{pmatrix},$$

where v is given by

$$v = -\frac{R^{-1}b}{\gamma},$$

with $b = R^{-T}z$ and $\gamma = \sqrt{1 - b^T b}$.

After noting that it is advantageous to work with the inverse Cholesky factor R^{-1} in a multiprocessor environment, it is important to compute the rotations forming Q and H directly from R^{-1} instead of R. Fortunately this is indeed the case as shown in Lemma 3 and 5 in [5]. We restate Lemma 3 below and refer the readers to [5] for Lemma 5 which gives similar result on computing H.

Lemma 3 [5] *Given the initial inverse Cholesky factor R^{-1} and the new observation equation $y^T w = \sigma$. Denote*

$$R^{-T}y = a = \begin{pmatrix} a_1 \\ a_2 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \\ a_n \end{pmatrix}.$$

Let $Q \equiv Q_n \cdots Q_2 Q_1$ be a product of Givens rotation matrices chosen to zero out $-a_1$, $-a_2$, $\cdots$, $-a_n$ by the following series of transformations:

$$\begin{pmatrix} -a_1 \\ -a_2 \\ \cdot \\ \cdot \\ \cdot \\ -a_{n-1} \\ -a_n \\ 1 \end{pmatrix} \longrightarrow \begin{pmatrix} 0 \\ -a_2 \\ \cdot \\ \cdot \\ \cdot \\ -a_{n-1} \\ -a_n \\ \delta^{(1)} \end{pmatrix} \longrightarrow \begin{pmatrix} 0 \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ -a_{n-1} \\ -a_n \\ \delta^{(2)} \end{pmatrix} \longrightarrow \cdots \longrightarrow \begin{pmatrix} 0 \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \\ -a_n \\ \delta^{(n-1)} \end{pmatrix} \longrightarrow \begin{pmatrix} 0 \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \\ 0 \\ \delta^{(n)} \end{pmatrix}$$

where $\delta^{(i)} = \sqrt{1 + a_1^2 + \cdots + a_i^2}$; i. e. ,

$$Q \begin{pmatrix} -a \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ \delta \end{pmatrix},$$

where $\delta = \delta^{(n)} = \sqrt{1 + a^T a}$. Then

$$Q \begin{pmatrix} R \\ y^T \end{pmatrix} = \begin{pmatrix} U \\ 0^T \end{pmatrix}.$$

Combining the results from Theorem 1 and Lemma 3, one immediately obtains

$$Q \begin{pmatrix} -a & R^{-T} \\ 1 & 0^T \end{pmatrix} = \begin{pmatrix} 0 & U^{-T} \\ \delta & u^T \end{pmatrix},$$

where U^{-1} is the *updated* inverse Cholesky factor. It was also shown in [5] that a scaled form of the by-product u^T is the Kalman gain vector, which is immediately useful in computing the modified least squares solution w' from

$$w' = w - \frac{\left(\sigma - y^T w\right)}{\delta} u .$$

2 Parallel Implementations

Since the inverse updating algorithm is similar to the downdating algorithm in data dependency and other relevant computational aspects, it is sufficient to study one algorithm for parallel implementation, and then apply the same strategy to the other. We have chosen to examine the updating process. Given below is the sequential algorithm for the entire least squares inverse updating process as summarized in [5].

Algorithm LS-IU. (Least Squares - Inverse Updating). Given the current least squares estimator vector w, the current inverse Cholesky factor R^{-1} and the new observation $y^T w = \sigma$ to be added, the algorithm computes the updated inverse Cholesky factor U^{-1} as well as the updated least squares estimator w'. We assume that both R^{-1} and w are already in memory.

Step I. Input y and σ of the new observation.

Step II. Form the matrix-vector product $a = R^{-T} y$.

Step III. Compute

$$Q \begin{pmatrix} -a & R^{-T} \\ 1 & 0^{-T} \end{pmatrix} = \begin{pmatrix} 0 & U^{-T} \\ \delta & u^T \end{pmatrix}, \quad Q^T Q = I ,$$

where $\delta = \sqrt{1 + a^T a}$, $Q = Q_n \cdots Q_2 Q_1$. The Givens parameters c_i and s_i for Q_i are explicitly computed by

$$c_i = \frac{\alpha_{i-1}}{\alpha_i}, s_i = -\frac{a_i}{\alpha_i},$$

where $\alpha_o = 1$, $\alpha_i = \sqrt{1 + a_1^2 + \cdots + a_i^2}$, $i = 1, \cdots, n$. Note that R^{-1} is overwritten by U^{-1} in the actual implementation.

Step IV. Compute

$$w' = w - \frac{(\sigma - y^T w)}{\delta} u \,,$$

where w is overwritten by w' in the actual implementation.

Step V. Output the updated w to the application.

For easy grasp of the strategies which may be useful in parallelizing this algorithm on a distributed-memory multiprocessor, let us depict the data accessing pattern of Step III in Fig. 1 for an inverse Cholesky factor R^{-T} of dimension $n = 6$. The elements marked ♠ are accessed and/or modified by the rotation matrix Q_i in the i^{th} transformation step.

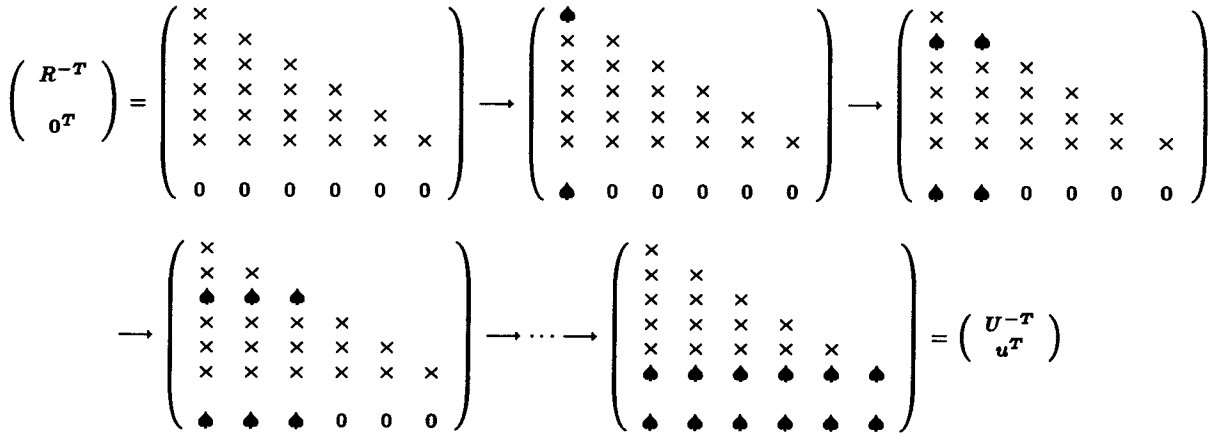

Figure 1: Elements accessed and modified by applying Q_i.

From Fig. 1 it should be clear that the application of Q_i modifies the i^{th} row of R^{-T} and the leading i elements of the bottom row. With the above picture in our mind, let us first describe a readily available parallel implementation for Step III, which is based on the recent work by Heath, Henkel and Plemmons [3] on parallelizing similar computations. As pointed out in [5], their ideas are immediately applicable to implement Step III on a distributed-memory multiprocessor without the need of communication, assuming that the columns of R^{-T} have been distributed to all processors using a wrap-mapping and that the entire vector a is available in each processor. When there are three processors, we use the example in Fig. 1 to illustrate the initial data distribution in Fig. 2.

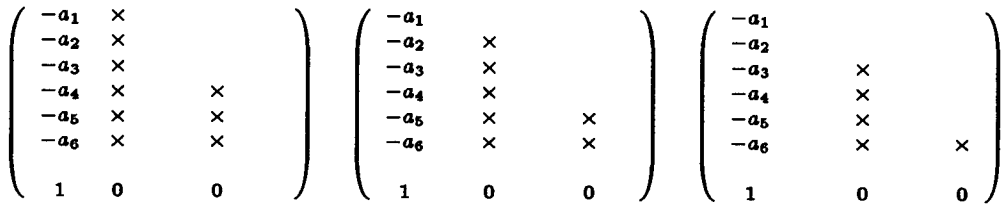

Figure 2: Initial data distribution on three processors.

Clearly every processor can independently (and redundantly) compute $Q_1, Q_2, \cdots, Q_n$ and apply each Q_i to the locally available portion of the i^{th} row of R^{-T} as well as that of the bottom row. We illustrate how all processors can concurrently apply Q_3 to update their local data in Fig. 3.

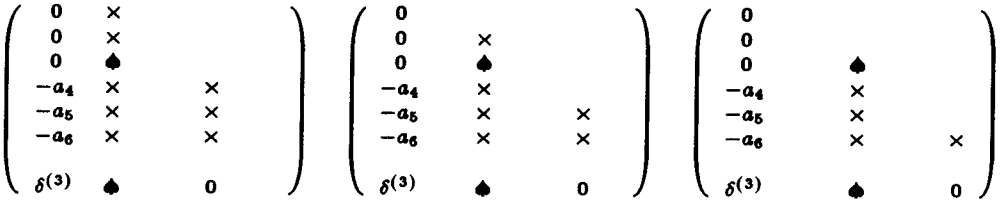

Figure 3: Three processors may apply Q_3 to their local data concurrently.

Since neither host-to-node nor node-to-node communication is required in Step III using the implementation above, one needs to consider only the communication costs incurred in Steps I, II, IV and V in parallelizing Algorithm LS-IU. To be specific, we restrict our discussion on communication costs here to the well-studied hypercube network. Step I requires the host to distribute the newly observed y and σ to all node processors. Step II computes $a = R^{-T}y$. Assuming that the columns of R^{-T} are initially wrap-mapped to the p processors of a hypercube network, so are the elements of y, it is well known that the product $a = R^{-T}y$ can be computed and made available in every processor using standard $O(\log_2 p)$ fan-in/fanout broadcast algorithms. The communication volume of $2n \log_2 p$ is a function of both n, the size of the problem, and p, the total number of processors. In [4] subcube doubling technique is used to accumulate the product $a = R^{-T}y$, which may reduce the communication volume from $2n \log_2 p$ to $n \log_2 p$ if the hypercube employed allows simultaneous bidirectional message traffic between two adjacent nodes.

Step IV computes the new estimator w' via

$$w' = w - \frac{\left(\sigma - y^T w\right)}{\delta} u \, .$$

In preparation for analyzing the communication cost incurred in Step IV, we make the following observations.

1. Updating w_i, the i^{th} component of w, requires the availability of u_i, δ, σ and $y^T w$.
2. The elements of u^T are scattered among the p processors in a wrap fashion as dictated by the implementation of Step III.
3. Each processor has computed δ at the end of Step III.
4. σ is available in each processor after executing Step I.

Thus, the processor having u_i can independently update w_i if $y^T w$ is provided. To this end, Henkel and Plemmons [4] propose to accumulate $y^T w$ in Step II along with the computation of $a = R^{-T} y$ assuming that w is currently wrap-mapped around the p processors. Since the partially computed $y^T w$ (a scalar) and a can be passed along in one message, the increase in communication cost is negligible. With u_i's appropriately distributed and the scalars δ, σ and $y^T w$ available in the local memory of each processor, Step IV can thus be parallelized without communication cost.

While the local update of w_i's in Step IV maintains the wrap-mapping which is desirable for use in subsequent updatings, it is, however, necessary to collect all of the updated w_i's in the host processor so that the new least squares estimates can be communicated to the application in Step V.

3 Motivations for an Alternative Parallel Implementation

In this section we describe our motivations for a new parallel implementation of Algorithm LS-IU, which features a smaller communication volume of "$n + O(\log_2 p)$" with a significant portion of that potentially masked by computation. Our design of a *row-oriented* block mapping strategy is motivated by the following observations.

1. As pointed out earlier, the rotation matrix Q_i modifies the i^{th} row of R^{-T} and the leading i elements of the bottom row. For easy reference, we show again in Fig. 4 the data accessed/modified by the application of Q_3 when R^{-T} is of dimension 6.

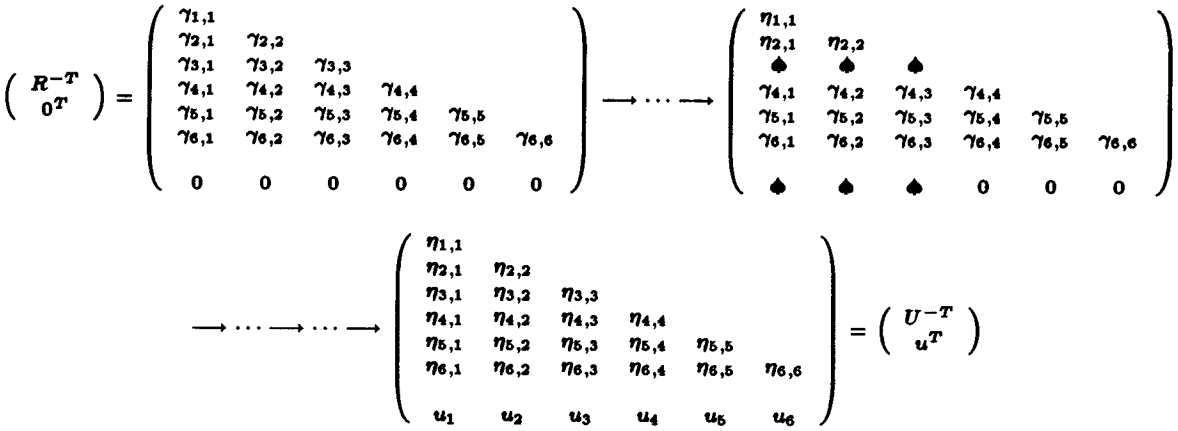

Figure 4: ♠: elements accessed and/or modified by applying Q_3.

2. Recall that

$$Q_3 = \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & c_3 & & & -s_3 \\ & & & 1 & & \\ & & & & 1 & \\ & & s_3 & & & c_3 \end{pmatrix}$$

where $c_3 = \alpha_2/\alpha_3$, $s_3 = -a_3/\alpha_3$, and α_2, α_3 are computed from $\alpha_0 = 1$ and $\alpha_i = \sqrt{1 + a_1^2 + \cdots + a_i^2}$. Observe that

$$\alpha_3 = \sqrt{1 + a_1^2 + a_2^2 + a_3^2} = \sqrt{\alpha_2^2 + a_3^2}\,.$$

In general, we can express α_i by

$$\alpha_0 = 1\ ,\ \alpha_i = \sqrt{\alpha_{i-1}^2 + a_i^2}\ , i = 2, 3, \cdots, n\ . \tag{1}$$

Note that by making use of equation (1) the processor which computes Q_i needs to access only α_{i-1} and a_i.

3. An observation which is important in exploiting the parallelism inherent in Algorithm LS-IU is that the computation of Q_i is independent of the elements being modified presuming that α_{i-1} and a_i are available. The parallel implication is that the i pairs of elements to be modified by Q_i may be processed in any order. For our example in Fig. 4, there are six possible sequences one may use to apply Q_3 to the three pairs of elements as shown in Fig. 5. We have labelled each pair of elements to indicate the intended ordering. In general, there are $i!$ alternative sequences for applying Q_i to the i pairs of elements.

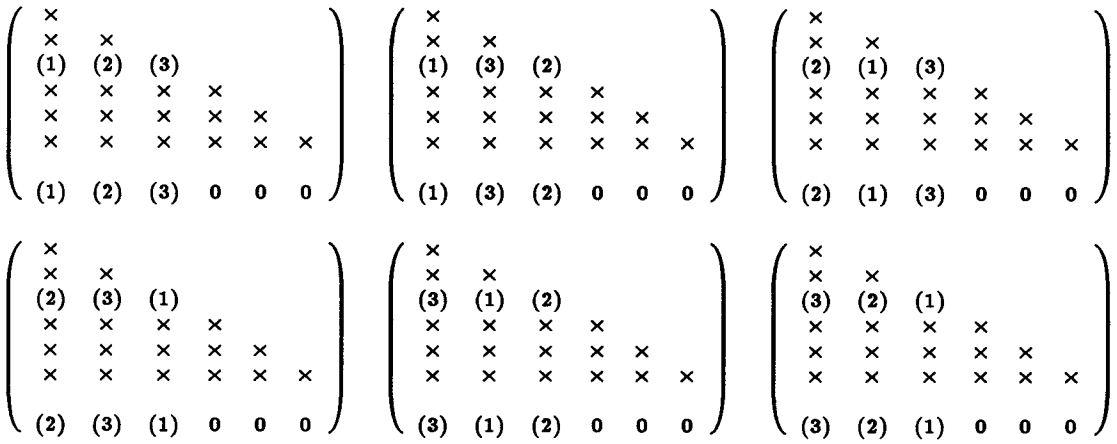

Figure 5: Six possible orderings exist in applying rotation Q_3.

4. Since the work involved in modifying a single pair of elements is the same regardless of the element locations, the number of steps may be used to measure the work load.

5. It is equally important to identify the precedence constraint which must be satisfied to ensure the correctness of the algorithm. For Algorithm LS-IU, the constraint is that $Q_1, Q_2, \cdots Q_n$ must be applied in strict order, because Q_i modifies the i^{th} row of R^{-T} using the *updated* (by rotation Q_{i-1}) bottom row. The implication is that the elements along the same *column* must be processed from top down as illustrated

in Fig. 6, where we have labelled each element by its step number. The precedence constraint requires that $x_i < x_j$ if $i < j$, where $x \in \{\imath, \jmath, \ell, \kappa, \mu, \nu\}$.

$$\begin{pmatrix} \imath_1 & & & & & \\ \imath_2 & \jmath_1 & & & & \\ \imath_3 & \jmath_2 & \ell_1 & & & \\ \imath_4 & \jmath_3 & \ell_2 & \kappa_1 & & \\ \imath_5 & \jmath_4 & \ell_3 & \kappa_2 & \mu_1 & \\ \imath_6 & \jmath_5 & \ell_4 & \kappa_3 & \mu_2 & \nu_1 \\ & & & & & \\ \times & \times & \times & \times & \times & \times \end{pmatrix}$$

Figure 6: Precedence constraint imposed by Q: $x_i < x_j$ if $i < j$.

6. An example representing a close-to-ideal data mapping is given in Fig. 7, where the data are partitioned into two blocks (as shown on the left) with each one assigned to a different processor. Each processor modifies its local data column by column from *right* to *left*. On the right, the ordering for each processor are indicated by step numbers, namely i for P_1 which has the top submatrix, and (i) for P_2 which has the bottom submatrix. Apparently the work load is close-to-equal (21 pairs versus 24 pairs of elements to be modified) and the precedence constraint is also observed. Suppose that the two processors are each equipped with local-memory only. In this case, for processor P_2 to continue with step (7), which applies Q_7 to the pair consisting of $\gamma_{7,6}$ and the corresponding element in the bottom row, the *updated* (by Q_6) element of the bottom row must be made available to P_2. Our numbering of the steps in Fig. 7 clearly indicates that if P_1 sends out this element to the neighboring P_2 immediately after it is updated by Q_6 in step 1, the communication cost can be completely masked by computation because P_2 does not need this element until step (7). Using a similar argument, there is absolutely no waiting before P_2 can execute steps (10), (13), (16), (19) unless the communication is unreasonably slower than computation.

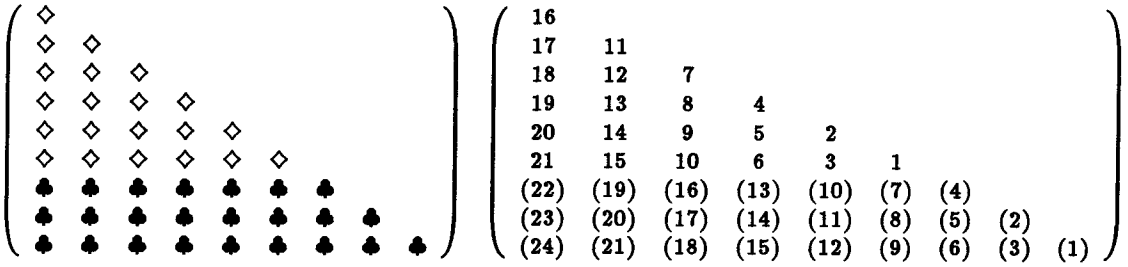

Figure 7: An example representing an ideal data mapping for $n = 9$ and $p = 2$.

It is important to recognize that the ideal mapping above is not a coincidence for this particular example. A simple analysis shows the following.

(a) The difference of six between step 1 and step (7) is accounted for by the six elements marked $\times$ in Fig. 8.

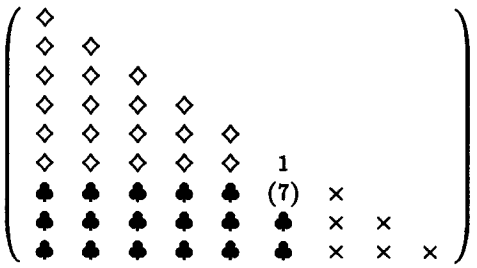

Figure 8: The six "$\times$" elements account for delay between step 1 and (7).

(b) The difference is increased to seven between step 6 and step (13) due to the further delay by element $\times$ in the bottom submatrix as shown in Fig. 9.

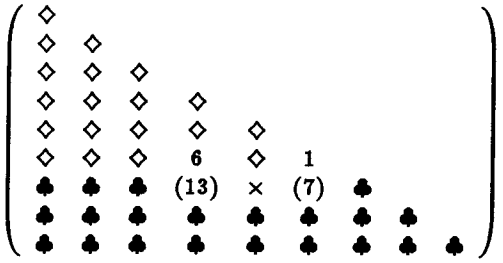

Figure 9: The "×" element accounts for delay of one more time step.

(c) The lead of seven steps by processor P_1 has been cut by six when P_2 reaches step (22) due to P_1's extra work involved in modifying the six elements marked "×" in Fig. 10.

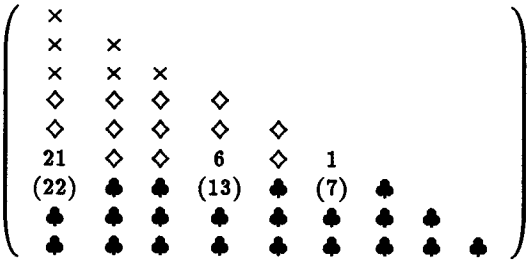

Figure 10: The six "×" elements account for P_1's extra work.

We shall show later how the observation above can be used to find such a partitioning for a given inverse factor of any dimension using $p = 2^d$ processors.

7. Using the example in Fig. 7, it is easy to see that the two processors can each compute a segment of $a = R^{-T}y$ concurrently if the n-vector y is available in each processor. The data distribution of R^{-T} dictates the partitioning of vector a as shown in Fig. 11. At the end of this step, processor P_1 would have computed $\{a_1, a_2, a_3, a_4, a_5, a_6\}$ locally, whereas P_2 would have $\{a_7, a_8, a_9\}$ computed.

   However, in order to proceed with Step III as suggested above in item 6, P_1 and P_2 must each be able to compute Q_i's for their respective needs in order to execute Step III concurrently. Recall that the Givens parameters c_i and s_i are computed from

$$c_i = \frac{\alpha_{i-1}}{\alpha_i}\,, s_i = -\frac{a_i}{\alpha_i}\,,$$

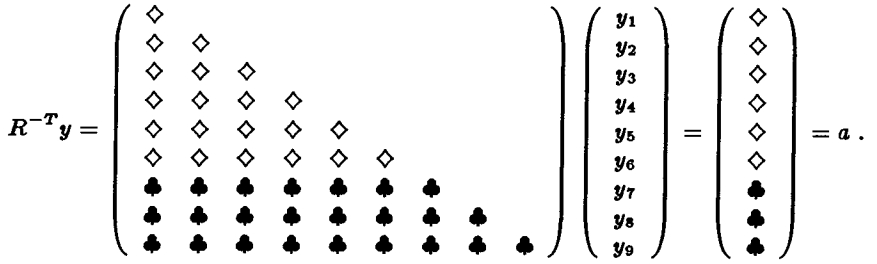

Figure 11: The concurrent computation of $a = R^{-T}y$ by two processors.

where $\alpha_0 = 1$ and $\alpha_i = \sqrt{1 + a_1^2 + \cdots + a_i^2}$. Clearly, P_1 can compute Q_1, Q_2, $\cdots$, Q_6 from $\{a_1, a_2, \cdots, a_6\}$, but P_2 will need $\{\alpha_6, a_7, a_8, a_9\}$ in order to compute α_7, α_8 and α_9 before the corresponding Q_i's can be constructed. Since $\alpha_6 = \sqrt{1 + \sum_{i=1}^{6} a_i^2}$, the only communication cost incurred is for transmitting one floating point number representing the value of $\sum_{i=1}^{6} a_i^2$ from P_1 to the neighbouring P_2.

We have designed a novel communication algorithm which generalizes the above strategy to a hypercube network of $p = 2^d$ processors. We postpone the presentation of the general form of the algorithm until the next section. The advantage of our scheme is, in short, that comparing with the "$O(n \log_2 p)$" communication volume for making the entire n-vector a available to all p processors as suggested by the column-oriented algorithm [3], we can reduce the communication volume to "$O(\log_2 p)$" by $\log_2 p$ exchanges of a single floating-point number between directly-connected processors.

4 A New Parallel Implementation for a Hypercube Multiprocessor

In the previous section we used an example to show that it is potentially advantageous to employ a row-oriented block mapping strategy. Since we only consider the case $p = 2$ in the example, the generalization to a hypercube network consisting of $p = 2^d$ processors certainly needs further explanation. Let us consider its implementation from three aspects, namely

1. how to choose and embed a suitable topology in the hypercube network.
2. how to partition the data among the p processors.
3. how to reduce the communication volume.

We address these three issues in the next three sections.

4.1 Topology Embedding

The choice of a particular network topology cannot be made without considering the data mapping strategy and the communication algorithm at the same time. So are the choices for the other two. Since we cannot describe all three in parallel, we present the chosen topology first and offer some justification after we describe the data mapping strategy and the communication algorithm. We have chosen to embed a linear array in the hypercube in order to facilitate the near-neighbour communication algorithm to be employed in Step III. The embedding consists of a unique mapping from the d-bit *reflected binary Gray code* [6] to the 2^d processor *id*'s of the given hypercube network. Given in Fig. 12 is an example for a hypercube of dimension 3. Note that the embedding is recursive – a hypercube of dimension d can be viewed as a linear array of two subcubes of dimension $(d-1)$. We shall see later how this feature is useful in our design of an efficient communication algorithm for Step II.

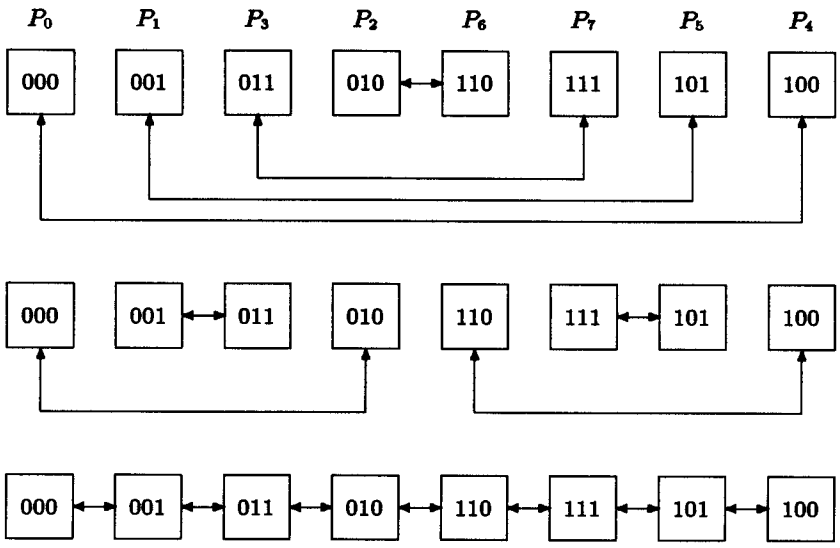

Figure 12: The communication network of a hypercube and the embedded linear array.

4.2 A Recursive Partitioning Algorithm

We first recall that in section 3 we observe that since the work involved in modifying a single pair of elements is the same regardless of the element location, the number of steps may be used to measure the work load. Let us further note that the total number of steps

to be performed by a particular processor is equal to the number of elements of R^{-T} the processor is allocated. Ideally, a data partitioning strategy should achieve the following objectives.

1. Every processor is assigned equal work.
2. All processors can work concurrently throughout the entire computation.

By treating the lower triangular matrix R^{-T} as a geometric object as in Fig. 13, the above objectives can be restated as below. As an example, we consider the simple case when there

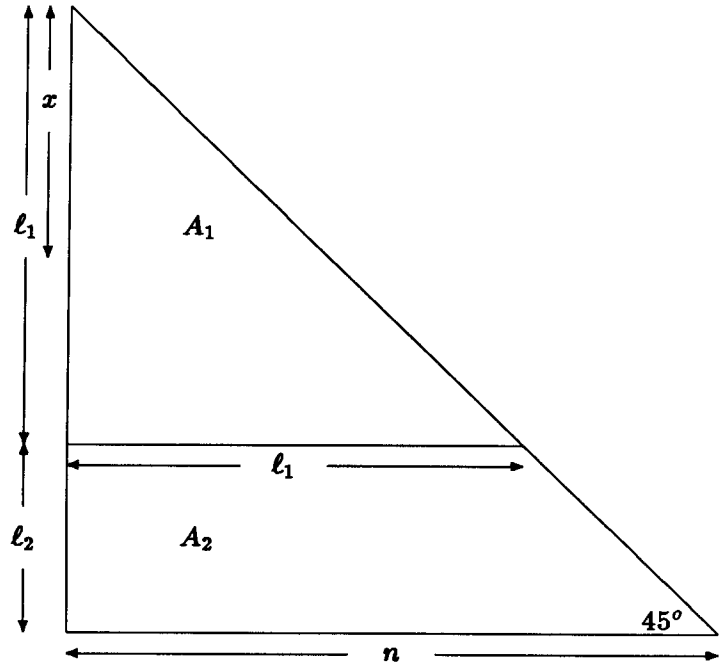

Figure 13: Bisecting a triangle so that the area $A_1 = A_2$.

are only two processors. The equal work requirement demands bisecting the triangle so that the "area" of the triangle and the trapezoid assigned to each processor is of the same size, i. e. $A_1 = A_2$ in Fig. 13. Using our notations in Fig. 13, we have $\ell_1 + \ell_2 = n$; letting $x = \ell_1 - \ell_2$, the condition $A_1 = A_2$ immediately yields

$$\frac{x^2}{2} = \frac{(n-x)^2}{4} . \tag{2}$$

Solving equation (2), we obtain

$$x = (\sqrt{2}-1)n . \tag{3}$$

Substituting x by $(\sqrt{2}-1)n$, we have

$$\ell_1 = \frac{n+x}{2} \approx 0.7n \tag{4}$$

and

$$\ell_2 = \frac{n-x}{2} \approx 0.3n\ . \tag{5}$$

The following lemma summarizes the results immediately obtained from equations (4) and (5).

Lemma 4 *We consider bisecting a given triangular matrix of dimension n as shown in Fig. 13. We denote the elements in A_1 and A_2 by U_1 and U_2 respectively. If $\ell_1 = \lfloor 0.7n \rfloor = 0.7n$, then we have*

(i) $U_2 - U_1 = 0.01n^2 - 0.2n$.

(ii) $U_1 = U_2$ *if and only if* $n = 20$.

(iii) $U_1 < U_2$ *if and only if* $n > 20$.

(iv) $U_1 > U_2$ *if and only if* $n < 20$.

Proof: By noting that

$$U_1 = \sum_{i=1}^{\ell_1} i = 0.245n^2 + 0.35n \tag{6}$$

and

$$U_2 = \frac{0.3n(0.7n+1+n)}{2} = 0.255n^2 + 0.15n\ , \tag{7}$$

we obtain the results in (i), (ii), (iii) and (iv) immediately. □

The parallel implication of the bisecting method is stated below as Theorem 5.

Theorem 5 *We consider bisecting a given lower triangular matrix of dimension n into two submatrices as indicated in Fig. 13. If U_1, the number of elements in A_1, is equal to or less than U_2, the number of elements in A_2, it can be shown that processor P_2 will not spend any time waiting until it processes the last column of data.*

Proof: We sketch the proof by illustrating the requirement of $U_1 = U_2$ in Fig. 14 for $n = 20$, where we show that for $U_1 = U_2$, the number of elements marked "×" in the top

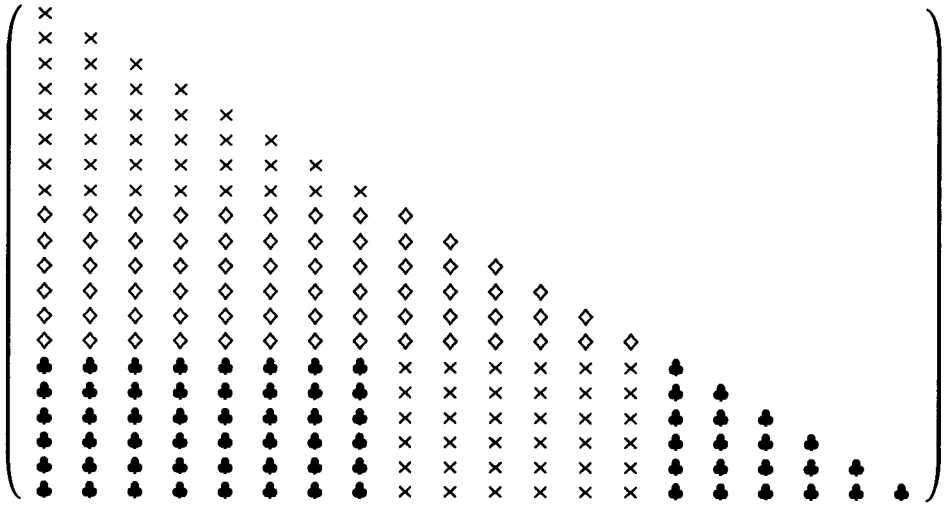

Figure 14: Illustration of the condition $U_1 = U_2$ when $n = 20$.

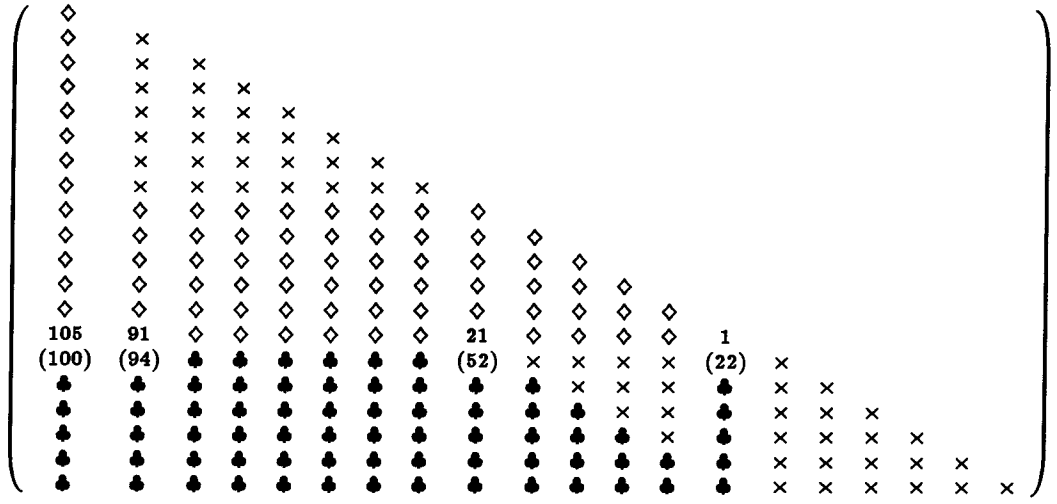

Figure 15: P_2 proceeds without waiting until the last column if the $\times$ elements in the top submatrix are less than the $\times$ elements in the bottom submatrix.

submatrix must be equal to the number of elements marked "×" in the bottom submatrix; i.e.,

$$\frac{x(x+1)}{2} = \frac{(n-x)^2}{4} . \tag{8}$$

For $n = 20$, $x = 8$ is indeed the solution to equation (8).

We then compare with the requirement imposed by allowing P_2 to proceed without waiting until the last column. As we demonstrate in Fig. 15, the latter condition can be transformed into the requirement that the number of elements marked "×" in the top submatrix must be less than the elements marked "×" in the bottom submatrix, i.e.,

$$\frac{x(x-1)}{2} < \frac{(n-x)^2}{4} - \frac{(n-x)}{2} + 1 . \tag{9}$$

To show that the inequality (9) holds, we observe that equation (8) implies

$$x > \frac{(n-x)}{2} . \tag{10}$$

We thus have

$$\begin{aligned} \frac{x(x-1)}{2} &= \frac{x(x+1)}{2} - x^2 \\ &= \frac{(n-x)^2}{4} - x^2 \\ &< \frac{(n-x)^2}{4} - \frac{(n-x)}{2} + 1 . \end{aligned} \tag{11}$$

□

We state the implication of Theorem 5 in Corollary 6.

Corollary 6 *If a given $n \times n$ ($n > 20$) lower triangular matrix is bisected according to $\ell_1 = \lfloor 0.7n \rfloor$ and $\ell_2 = n - \ell_1$, then the number of elements in A_1 is less than or equal to the elements in A_2 and P_2 will not spend any time waiting until it processes the last column of data. Furthermore, the number of elements in A_2 plus the number of elements in the last column of A_1 represent the upper bound of the total number of steps of the parallel algorithm.*

Clearly the bisecting method can be applied again to A_1 and each resulting top submatrix recursively. It is also clear that if we can bisect the bottom trapezoid in a similar fashion, then it can also be applied recursively to each resulting trapezoidal submatrix. At this point, let us also recall that we are interested in implementing this algorithm on a hypercube network of $p = 2^d$ processors. Note that a complete data partitioning algorithm consisting of d bisecting steps would generate 2^d submatrices, which are exactly what we

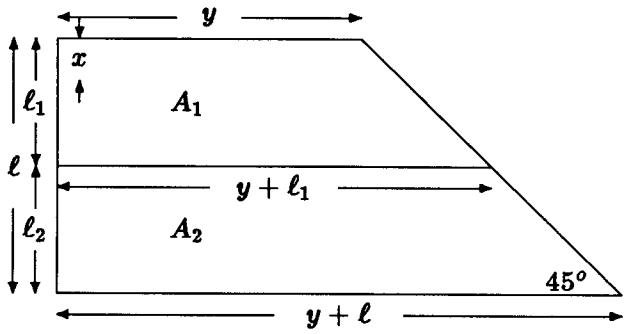

Figure 16: Bisecting a trapezoid so that the area $A_1 = A_2$.

need for the $p = 2^d$ processors consisting of the linear array. We shall next describe how to bisect a trapezoidal submatrix to meet our objectives. Let us treat the trapezoidal submatrix as a geometric object and fix our notations in Fig. 16. Referring to Fig. 16, given ℓ and y we need to find ℓ_1 ($\ell_2 = \ell - \ell_1$) so that area $A_1 = A_2$ for exactly the same reason as we bisect a triangle. We state our solution as Lemma 7.

Lemma 7 *The trapezoid in Fig. 16 can be bisected into two trapezoids of equal area if we choose* $\ell_1 = (\ell + x)/2$, *where* $x = \sqrt{(2y + \ell)^2 + \ell^2} - (2y + \ell)$.

Proof: Letting $x = \ell_1 - \ell_2$, we have

$$\ell_1 = \frac{\ell + x}{2}, \tag{12}$$

$$\ell_2 = \frac{\ell - x}{2}. \tag{13}$$

Since the areas A_1 and A_2 are each computed by

$$A_1 = \frac{y + (y + \ell_1)}{2}\ell_1 \tag{14}$$

and

$$A_2 = \frac{(y + \ell_1) + (y + \ell)}{2}\ell_2, \tag{15}$$

the condition $A_1 = A_2$ is satisfied if

$$\frac{y + (y + \ell_1)}{2}\ell_1 = \frac{(y + \ell_1) + (y + \ell)}{2}\ell_2. \tag{16}$$

Substituting $\ell_1 = (\ell + x)/2$ and $\ell_2 = (\ell - x)/2$ into equation (16), we obtain the following quadratic equation.

$$x^2 + (4y + 2\ell)x - \ell^2 = 0 \,. \tag{17}$$

Solving equation (17), we obtain

$$x = \frac{-(4y + 2\ell) \pm \sqrt{(4y + 2\ell)^2 + 4\ell^2}}{2} \,. \tag{18}$$

Since $x > 0$, our solution to equation (17) is

$$x = -(2y + \ell) + \sqrt{(2y + \ell)^2 + \ell^2} \,. \tag{19}$$

□

We can now apply Lemma 7 to a trapezoidal matrix and prove the following theorem.

Theorem 8 *We consider bisecting a given lower trapezoidal matrix into two submatrices as indicated in Fig. 16. If we choose* $\ell_1 = \lfloor (\ell + \lfloor x \rfloor)/2 \rfloor$, *where* $x = \sqrt{(2y + \ell)^2 + \ell^2} - (2y + \ell)$, *then the following results hold.*

1. *Suppose that the number of elements in* A_1 *is denoted by* U_1 *and the number of elements in* A_2 *is denoted by* U_2, *then* $U_1 < U_2$.
2. *If the top submatrix is processed by* P_1 *and the lower submatrix is processed by* P_2 *column by column from right to left, then processor* P_2 *will not spend any time waiting until it processes the last* k *columns, where* $k \geq 0$ *can be determined after* x *is chosen for a given problem.*

Proof: Since

$$\ell_1 = \left\lfloor \frac{\ell + \lfloor x \rfloor}{2} \right\rfloor \leq \frac{\ell + x}{2} \,, \tag{20}$$

and when $\ell_1 = (\ell + x)/2$, we have from Lemma 7 that

$$A_1 = A_2 \,, \tag{21}$$

$$U_1 = A_1 \tag{22}$$

and

$$U_2 > A_2 \,. \tag{23}$$

The inequality $U_1 < U_2$ is thus an immediate result from Lemma 7.

To prove the second result, we observe that the lead by P_1 is

$$\ell_2^2 - \ell_2 + 1 = (\ell - \ell_1)^2 - (\ell - \ell_1) + 1 \,, \tag{24}$$

and that we need to find the smallest k such that

$$zy + \frac{z(z-1)}{2} - kz \leq \ell_2^2 - \ell_2 + 1 \,, \quad z = \ell_1 - \ell_2 \,. \tag{25}$$

Since y is given, ℓ_1 and ℓ_2 are both known after x is determined, k can be easily computed. □

4.3 A Novel Communication Algorithm

Suppose that we have applied the recursive partitioning algorithm described in the previous section to divide R^{-T}, the transpose of a given $n \times n$ inverse Cholesky factor, into $p = 2^d$ submatrices, where d is the dimension of the hypercube network. Let us denote $R^{-T}y = a$ and assume that every processor has access to the new observation vector y. Referring to Fig. 11, every processor can compute a subset of $\{a_k | 0 \leq k \leq n\}$ independently and simultaneously. The size of the subset is the same as the number of rows of R^{-T} each processor is assigned. Let us use i_o and i_n to denote the first and the last row number of the block assigned to processor P_i. Clearly every processor can also compute the partial sum of a_k^2, namely

$$S_\gamma = \sum_{k=i_o}^{i_n} a_k^2 \,, \tag{26}$$

where γ $(1 \leq \gamma \leq p)$ represents the location of P_i in the linear array. Recall that we explain in detail earlier that for each processor to construct the Givens rotation matrices Q_{i_o} to Q_{i_n} concurrently, processor P_i needs α_{i_o-1} in order to compute $\{\alpha_{i_o}, \alpha_{i_o+1}, \cdots, \alpha_{i_n}\}$. Our goal in designing the communication algorithm is to have the data needed for computing α_{i_o-1} available to each P_i, $0 \leq i \leq p-1$, after $d = \log_2 p$ communication steps. The key observation underlying our communication algorithm is that

$$\alpha_{i_o-1} = \sqrt{1 + \sum_{m=1}^{\gamma-1} S_m} \,.$$

Our proposed algorithm employs the subcube doubling technique. Readers are referred to Fig. 12 for an example of the communication pattern when the processors are configured as a linear array. Using the subcube doubling technique, each processor is able to communicate with its d directly-connected neighbours after d communication steps without encountering traffic congestion. Let us denote the $d-$bit processor id by $b_0 b_1 \cdots b_{d-1}$, the

complete algorithm executed by processor P_i can be described as follows. Initially each processor P_i composes its own message to represent the index value γ and the partial sum S_γ computed by equation (26). Note again that γ here represents the location of P_i in the linear array. This point is made clear in Table 1 when we demonstrate how the algorithm works on a hypercube network consisting of 8 processors.

$S = 0$
$\ell \leftarrow d$
while $\ell > 0$ **do**
    send (my message) to processor with id different
        from my id in bit $b_{\ell-1}$.
    receive a message representing updated S_m and index m
    if $m < \gamma$ **then**
        $S \leftarrow S + S_m$
    $\ell \leftarrow \ell - 1$
    if $\ell > 0$ **then**
        update the index component of my message to be the maximum
            of γ and m.
        update the partial sum component of my message to be the sum
            of current S_γ and the received S_m.

At the end of the algorithm each processor P_i computes α_{i_o-1} from

$$\alpha_{i_o-1} = \sqrt{1+S} \tag{27}$$

To demonstrate how the algorithm works, we trace the proposed algorithm in Table 1 using the network in Fig. 12. From Table 1 it is straightforward to verify that the value of S available in the γ^{th} processor in the linear array is given by

$$S = \sum_{m=1}^{\gamma-1} S_m^{(0)} ,$$

where we use $S_m^{(0)}$ to indicate the initial value of S_m's as computed in column 4 of Table 1.

pid	P_i	γ	Initially	$\ell = 2$	$\ell = 1$	$\ell = 0$
000	P_0	1	compute $S_1^{(0)}$ set $S = 0$	$S_2^{(1)} = S_1^{(0)} + S_2^{(0)}$	$S_4^{(2)} = S_2^{(1)} + S_4^{(1)}$	
001	P_1	2	compute $S_2^{(0)}$ set $S = 0$	$S_2^{(1)} = S_1^{(0)} + S_2^{(0)}$ $S = S + S_1^{(0)}$	$S_4^{(2)} = S_2^{(1)} + S_4^{(1)}$	
011	P_3	3	compute $S_3^{(0)}$ set $S = 0$	$S_4^{(1)} = S_3^{(0)} + S_4^{(0)}$	$S_4^{(2)} = S_2^{(1)} + S_4^{(1)}$ $S = S + S_2^{(1)}$	
010	P_2	4	compute $S_4^{(0)}$ set $S = 0$	$S_4^{(1)} = S_3^{(0)} + S_4^{(0)}$ $S = S + S_3^{(0)}$	$S_4^{(2)} = S_2^{(1)} + S_4^{(1)}$ $S = S + S_2^{(1)}$	
110	P_6	5	compute $S_5^{(0)}$ set $S = 0$	$S_6^{(1)} = S_5^{(0)} + S_6^{(0)}$	$S_8^{(2)} = S_6^{(1)} + S_8^{(1)}$	$S = S + S_4^{(2)}$
111	P_7	6	compute $S_6^{(0)}$ set $S = 0$	$S_6^{(1)} = S_5^{(0)} + S_6^{(0)}$ $S = S + S_5^{(0)}$	$S_8^{(2)} = S_6^{(1)} + S_8^{(1)}$	$S = S + S_4^{(2)}$
101	P_5	7	compute $S_7^{(0)}$ set $S = 0$	$S_8^{(1)} = S_7^{(0)} + S_8^{(0)}$	$S_8^{(2)} = S_6^{(1)} + S_8^{(1)}$ $S = S + S_6^{(1)}$	$S = S + S_4^{(2)}$
100	P_4	8	compute $S_8^{(0)}$ set $S = 0$	$S_8^{(1)} = S_7^{(0)} + S_8^{(0)}$ $S = S + S_7^{(0)}$	$S_8^{(2)} = S_6^{(1)} + S_8^{(1)}$ $S = S + S_6^{(1)}$	$S = S + S_4^{(2)}$

Table 1: Demonstration of the proposed communication algorithm on a hypercube of dimension 3.

4.4 Computing the new estimator w'

Recall that in section 2 we describe Step IV of Algorithm LS-IU as computing the new estimator w' via

$$w' = w - \frac{\left(\sigma - y^T w\right)}{\delta} u \,. \tag{28}$$

We shall now proceed to describe how the data distribution dictated by the parallel implementation we propose for Step II and III supports the computation of w' by the last processor in the linear array.

We first recall that in our implementation of Step III we assume that the n-vector y representing the new observation is available in each processor. Referring to Table 1, it is clear that processor P_4, which is the last one in the 8-processor linear array ($\gamma = 8$) and is allocated the bottom submatrix, would have computed

$$S = \sum_{m=1}^{7} S_m^{(0)}$$

at the end of Step II and is ready to compute

$$\delta = \sqrt{1 + S + S_8^{(0)}}$$

$$= \sqrt{1 + a^T a}\,. \tag{29}$$

At the end of Step III, processor P_4 would have computed the entire u vector,too. Therefore, to update the estimator w via equation (28), processor P_4 only needs w and the right hand side of the new observation, σ. The latter is a scalar and can be distributed by the host processor together with vector y in Step I and incurs virtually no communication cost. Assuming that P_4 has w, we conclude that P_4 can compute the entire w' without communication cost and that the updated w remains in P_4 for use in future updating.

In Step V, P_4 would be the only node processor communicating the updated estimator vector w to the host. Finally observe that since only one node processor is involved in computing and reporting w, the distribution of newly available observation y and σ can overlap Steps IV and V.

5 Summary

In Section 2, we have presented Algorithm LS-IU in a manner which can be repeated when new observation becomes available. Since the inverse Cholesky factor R^{-1} and the estimator vector w are both updated *in-place* either in the column-oriented algorithm or in the proposed row-oriented algorithm, the distribution of R^{-1} and w is done only once before the Algorithm LS-IU is executed the first time. Therefore, the cost of the initial data mapping has little impact on the performance of the parallel algorithm over time. Comparing the column-oriented parallel implementation with the row-oriented one through Steps I to V of Algorithm LS-IU, we summarize our conclusions below.

1. In Step I, both implementations require the host to distribute the new observation represented by vector y and a scalar σ. In the column-oriented case, y is to be wrapped around the node processors while σ is needed by all of them; in the row-oriented case, y is needed by all nodes while σ is needed by a single processor. We do not expect significant difference in the communication cost of this step.

2. In Step II, the column-oriented implementation incurs a communication volume of $O(n \log_2 p)$, while the row-oriented algorithm reduces the communication volume to $O(\log_2 p)$.

3. In Step III, the column-oriented implementation incurs no communication cost. While the row-oriented implementation causes $O(n)$ communication among the nodes, we show how the communication can be masked by computation by inducing an appropriate precedence relationship on the data. The work load distribution is balanced in both cases.

4. In Step IV, the least square estimator w can be updated in either implementation without communication cost.

5. In Step V, while the host must collect the updated w_i's from all node processors in the column-oriented algorithm, only one node is responsible to communicate the updated vector w to the host in the row-oriented algorithm.

6. Since only one processor is involved in Steps IV and V in the row-oriented algorithm, it is possible to overlap Step I with Steps IV and V.

In a subsequent paper, we plan to implement our algorithm on a hypercube multiprocessor and compare our timing results with that of the column-oriented implementation recently reported in [4].

Acknowledgement

The authors thank Professor Robert J. Plemmons for suggesting the problem and providing us with references [4] and [5].

References

[1] J. Chun, T. Kailath, and H. Lev-Ari. Fast parallel algorithms for QR and triangular factorization. *SIAM J. Sci. Stat. Comput.*, 8:399–413, 1987.

[2] G. Cybenko. Fast toeplitz orthogonalization using inner products. *SIAM J. Sci. Stat. Comput.*, 8:734–740, 1987.

[3] C. S. Henkel, M. T. Heath, and R. J. Plemmons. Cholesky downdating on a hypercube. In *ACM Proc. Hypercube 3 Conf.*, pages 1592–1598, California Institute of Technology, 1988.

[4] C. S. Henkel and R. J. Plemmons. *Recursive Least Squares on a Hypercube Multiprocessor Using the Covariance Factorization.* Technical Report, Department of Nuclear Engineering and Department of Computer Science and Mathematics, October 1988. (submitted to *SIAM J. Sci. Stat. Comput.*).

[5] C. T. Pan and R. J. Plemmons. Parallel least squares modifications using inverse factorizations. *Computational and Applied Mathematics*, 1989. (to appear).

[6] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.