Parallel Algorithms and Subcube Embedding on a Hypercube

Eleanor Chu
Alan George

Department of Computer Science

Research Report CS-89-59
December 1989

Parallel Algorithms and Subcube Embedding on a Hypercube [*]

Eleanor Chu
Alan George

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

Research Report CS-89-59

December 1989

Abstract

It is well known that the connection in a hypercube multiprocessor is rich enough to allow the embedding of a variety of topologies within it. For a given problem, the best choice of topology is naturally the one which incurs the least amount of communication and allows parallel execution of as many tasks as possible. Efficient parallel algorithms for performing QR factorization on a hypercube multiprocessor are proposed in [E. Chu and A. George, *SIAM J. Sci. Statist. Comput*, 11:4 (July 1990)], where the hypercube network is configured as a two-dimensional subcube-grid with an aspect ratio optimally chosen for each problem. In view of the very substantial *net* saving in execution time and storage usage obtained in performing QR factorization on an optimally configured subcube-grid, similar strategies are developed in this work to provide highly efficient implementations for three fundamental numerical algorithms: Gaussian elimination with partial pivoting, QR factorization with column pivoting, and multiple last squares updating.

[*]This work was supported in part by Canadian Natural Sciences and Engineering Research Council grant OGP0008111, by NASA Grant NAGW-1457, and by a research grant from the University of Waterloo.

Contents

1 Introduction

It is well known that the connection in a hypercube multiprocessor is rich enough to allow the embedding of a variety of topologies within it. Some commonly known embedded topologies are linear array, mesh, toroid, ring and tree as demonstrated in Fig. 1 for a hypercube of dimension three. A two-dimensional mesh can be distinguished from a two-dimensional torus in a hypercube of a dimension higher than three as illustrated in Fig. 2. Since each of the

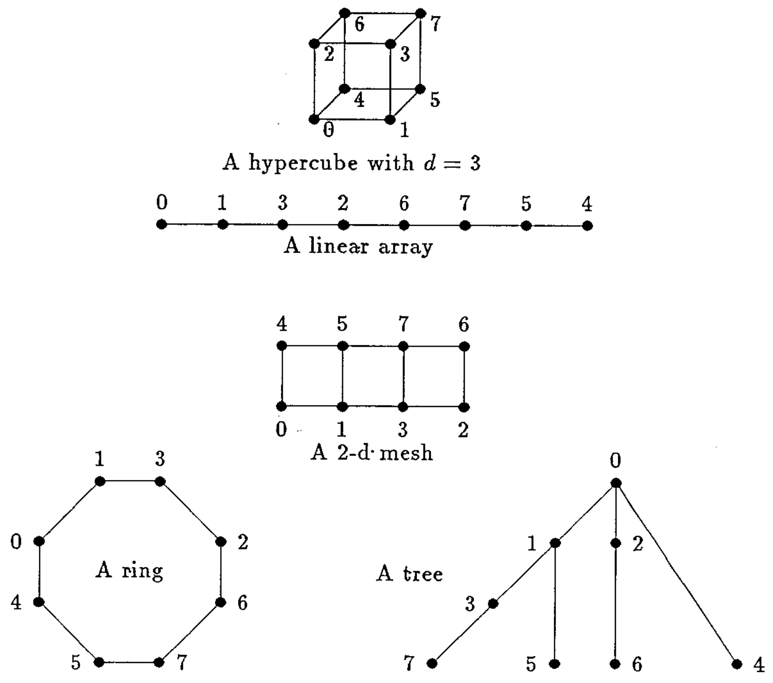


Figure 1: Topologies embedded in a hypercube of dimension 3.

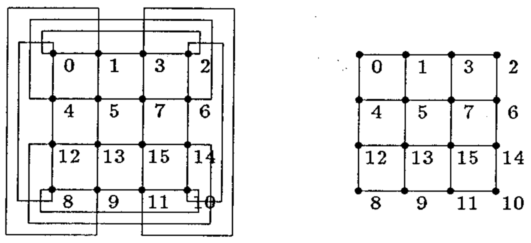


Figure 2: A 4×4 torus versus a 4×4 mesh-connected processor array.

topologies above employs a *different* subset of the communication channels in a hypercube network, the communication algorithm is necessarily different for each topology in order to minimize the latency in message transmission. Consequently one important decision to be made in the process of developing parallel algorithms for a hypercube multiprocessor is the embedded topology. The topology of choice and the precedence relationship between the computing tasks will normally dictate how the data should be distributed in order to achieve high parallelism. With the topology and the data distribution fixed, the communication algorithm can be tuned to reduce the latency in message transmission.

For a given problem, the best choice of topology is naturally the one which incurs the least amount of communication and allows parallel execution of as many tasks as possible. However, it may not be easy to evaluate the tradeoff between different topologies beforehand. In particular, even for the same problem one topology may be better than the other for only a specific range of problem sizes. This phenomenon is evident in the literature on parallel numerical algorithms.

In this paper, we advocate a topology which is very closely related to the two-dimensional mesh and torus, but enjoys a number of distinct features important in designing efficient parallel algorithms. We obtain this topology by configuring the hypercube network as a two-dimensional *subcube-grid*. A subcube-grid is not a mesh-connected processor array. Although a $\gamma_1 \times \gamma_2$ subcube-grid has γ_1 processors in each column and γ_2 processors in each row, the neighboring processors in the grid may or may not be physically connected; instead, each row and each column of processors are required to form a hypercube of smaller dimension, which is a subcube. The apparent difference between the two is illustrated by an example in Fig. 3, where each processor is denoted by a "•" and each physical communication channel is represented by a solid line between two processors. The diagram on the left in Fig. 3 shows a 4×4 subcube-grid and the diagram on the right shows a 4×4 mesh-connected processor array.

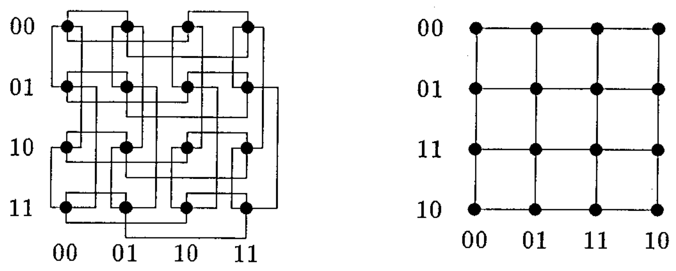
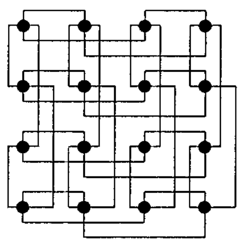


Figure 3: A 4×4 subcube-grid versus a 4×4 mesh-connected processor array.

As shown in Fig. 4, the processors of a hypercube of dimension $d = (d_1 + d_2)$ can be easily configured as a $\gamma_1 \times \gamma_2$ subcube-grid, where $\gamma_1 = 2^{d_1}$ and $\gamma_2 = 2^{d_2}$, by mapping the $p = 2^d$ processors to the subcube-grid row-by-row following the natural order of their processor *id*.

0000	0001	0010	0011
0100	0101	0110	0111
1000	1001	1010	1011
1100	1101	1110	1111



P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}
P_{16}	P_{17}	P_{18}	P_{19}	P_{20}	P_{21}	P_{22}	P_{23}
P_{24}	P_{25}	P_{26}	P_{27}	P_{28}	P_{29}	P_{30}	P_{31}

P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}
P_{16}	P_{17}	P_{18}	P_{19}
P_{20}	P_{21}	P_{22}	P_{23}
P_{24}	P_{25}	P_{26}	P_{27}
P_{28}	P_{29}	P_{30}	P_{31}

Figure 4: The configuration of a 4×4, a 4×8 and an 8×4 subcube-grids.

Since each column and each row of processors in a $\gamma_1 \times \gamma_2$ subcube-grid are hypercubes of dimensions d_1 and d_2 respectively, the basic subcube-doubling communication algorithm described below may be simultaneously employed by all γ_1 subcube rows, each row consisting of γ_2 processors with *id* varied only in the rightmost d_2 bits. Similarly, it can also be simultaneously employed by the γ_2 subcube columns, each column consisting of γ_1 processors with *id* varied only in the leftmost d_1 bits.

```
$\ell \leftarrow d$
while $\ell > 0$ do
    send (my message) to processor with id different
        from my id in bit $b_{\ell-1}$.
    receive a message
    update (my message) as instructed by each different algorithm
    $\ell \leftarrow \ell - 1$
```

In Fig. 5, we show the d *synchronous* communication steps in the algorithm above when $d = 3$. In Fig. 6, we show that the four subcube rows concurrently perform the $d_2 = 2$ communication steps and that the four subcube columns concurrently perform the $d_1 = 2$ communication steps in a 4×4 subcube-grid.

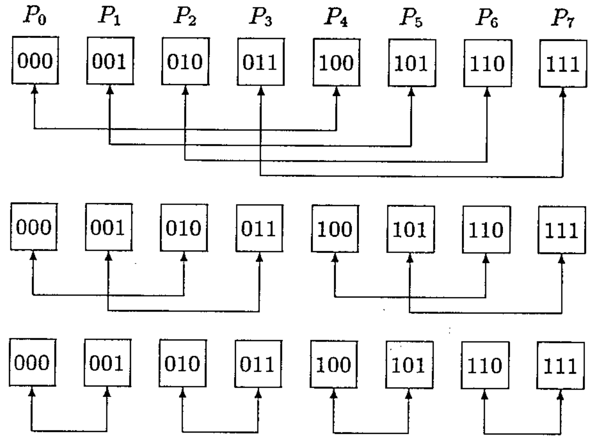


Figure 5: The d communication steps in the subcube-doubling algorithm ($d = 3$).

In our article [4] on implementing the QR factorization on a hypercube multiprocessor, we observed that different communication algorithms may be built on top of the basic subcube-doubling scheme by employing a variety of strategies in *updating* the message to be forwarded to the next neighbor. In particular, we show how *redundant update* can maintain data proximity so that exactly the same synchronous communication steps may be followed by all processors throughout the computation for all possible choices of the dimensions γ_1 and γ_2. The communication scheme we proposed in [4] allows us to employ the optimal aspect ratio "γ_1/γ_2", which is problem-dependent and is chosen at run time according to the particular dimensions of an input matrix. Since all $(p/2)$ disjoint pairs of processors exchange one message per communication step, the total number of messages is independent of the choice of the aspect ratio. The quantity that the optimal aspect ratio aims at reducing is the message length and hence the total communication volume. Since the price for minimizing the communication volume is paid in the form of *computing* the redundant update by otherwise *idle* processors, the net gain is very significant.

The actual enhancement in the performance of the parallel algorithm is demonstrated by the following timing results from [4]. Since the aspect ratio is simply a parameter to the parallel QR factorization program, the reduction from 1557 seconds, for example, in the

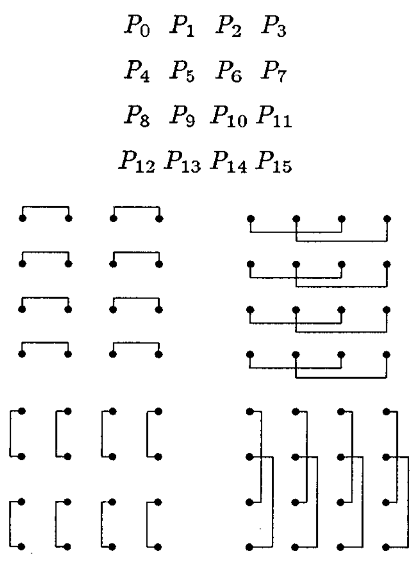

P_0 P_1 P_2 P_3
P_4 P_5 P_6 P_7
P_8 P_9 P_{10} P_{11}
P_{12} P_{13} P_{14} P_{15}

Figure 6: The channels concurrently employed by subcubes in $d = 4$ communication steps.

case $m = n = 1000$, to 1011 seconds by changing the subcube grid from 64×1 to 8×8 is indeed *free*. The reduction is even more impressive when $m \ll n$ or $m \gg n$ as seen in the bottom two entries in Table 1. In addition, we show in [4] that the optimal aspect ratio chosen in minimizing the communication volume also minimizes the storage requirement.

Table 1: Single-Precision Execution Times (sec) on Intel Hypercube iPSC.

QR Factorization of an $m \times n$ matrix using $\gamma_1 \times \gamma_2 = 64$ processors								
m	n	64×1	32×2	16×4	8×8	4×16	2×32	1×64
1000	1000	1557	1215	1060	1011*	1017	1084	1257
1200	800	1231	1012	922	905*	930	1030	1265
800	1200	1618	1183	986	907	891*	924	1030
1980	100	41.7	39.8*	42.0	48.5	64.4	99.6	175.0
100	1980	221.5	125.1	75.7	50.8	41.0	36.6	36.1*

*The minimum execution time

In view of the very substantial *net* saving in execution time and storage usage obtained in our earlier work [4], a natural question to ask is whether the same topology can be adapted to parallelize other numerical algorithms efficiently. In this paper, we propose new subcube-grid algorithms for the efficient parallel implementation of the following numerical algorithms.

1. Gaussian elimination with partial pivoting - In §2, we show not only that its subcube-

grid implementation enjoys the same saving as QR factorization, but also that partial pivoting can be incorporated without incurring extra computing or communications.

2. QR factorization with column pivoting - In §3, we show that column pivoting can be incorporated into our parallel algorithms in [4] without additional communications. We also show that the increase in computing cost is very small.

3. Multiple least squares updating - While the idea we use to develop the algorithms in [4] can be immediately applied to multiple least squares updating, it is equally important to have an efficient scheme to dynamically relocate the data of the computed Cholesky factor as the aspect ratio of the subcube grid changes. In §4, we show that with an appropriate mapping scheme, we can adapt the subcube doubling technique to obtain a *dynamic* data relocation algorithm for the Cholesky factor, and have thus obtained a feasible parallel algorithm for the continued process of multiple least squares updating.

2 Gaussian elimination with partial pivoting

Due to its fundamental and practical role in solving linear systems of equations, Gaussian elimination is often the first numerical algorithm considered when experimenting on a new computer architecture. Since 1985, the year in which hypercube multiprocessors became commercially available, much work has been done on the efficient implementation of this algorithm and its variants on hypercubes [1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16]. The topologies employed include the embedded ring, spanning-tree and two dimensional mesh-connected processor array. The communication algorithms devised for these embedded topologies often employ pipelining techniques to reduce communication delay in message passing. The data mapping strategies considered include column and row-oriented block or wrap mapping as well as submatrix block mapping. In [16], the theoretical lower bound for communication complexity of the Gaussian elimination algorithm on a nearest neighbor grid network was established to be $O\left(n^2/\sqrt{p}\right) + O\left(n\sqrt{p}\right)$ for a lock-step (synchronous communication) Gaussian elimination algorithm, and $O\left(n^2/\sqrt{p}\right) + O\left(\sqrt{p}\right)$ for any pipelined Gaussian elimination algorithm, where p is the total number of processors. Optimal concurrent algorithms for Gaussian elimination with and without pivoting are also proposed in [7]. However, since the algorithms proposed in [7] achieve optimality only on large hypercube machine with little latency and high communication bandwidth, they are not suitable for INTEL iPSC or NCUBE which have substantial message-passing latency. While the partial pivoting step is recognized to be crucial in maintaining stable computation in practice, it often causes extra communication and unwelcome disturbance in an otherwise well-pipelined flow of messages. Thus, the reduction of the latency caused by pivoting has been the subject of several studies. The parallel algorithm we propose in this paper offers a new implementation which incurs *no* extra communication while enjoying the significant saving in total communication cost by embedding a *subcube-grid* in the hypercube network.

2.1 A new parallel implementation

Since the parallel Gaussian elimination algorithm can be easily adapted from the subcube-grid algorithm we proposed in [4], we refer readers there for the algorithms originally deveoloped for the dense QR factorization and describe only the main features and modifications here.

1. In [4], the aspect ratio of the subcube-grid varies with the dimension of the rectangular matrices. Since we are dealing with only square matrices in Gaussian elimination with partial pivoting, the dimensions of the subcube along each column and each row, namely d_1 and d_2, should differ at most by one. Note that $d_1 = d_2$ if $d = d_1 + d_2$ is an even number. Following [4], we use γ_1 and γ_2 to denote the aspects of the subcube-grid, i.e. $\gamma_1 = 2^{d_1}$ and $\gamma_2 = 2^{d_2}$.

2. In [4], the rows of the matrix are wrapped around the γ_1 subcube-rows. Within each subcube-row, the elements are wrapped around the γ_2 processors (which form the subcube) according to their column subscripts. We employ the same data mapping strategy here for parallel Gaussian elimination with partial pivoting

3. Following [4], we proceed to reduce an $n \times n$ matrix to its upper triangular form in $(n-1)$ reduction steps. During each reduction step, $(n-i)$ nonzeros from the i^{th} column are eliminated, where $1 \leq i \leq n-1$.

4. During each reduction step, the $\lceil \frac{n-i}{\gamma_1} \rceil$ elements in the i^{th} column are broadcast to the γ_2 processors within each subcube-row by the subcube doubling technique exactly as proposed in [4]. We note that for Gaussian elimination *without* pivoting, the $\lceil \frac{n-i}{\gamma_2} \rceil$ elements in the i^{th} row can now be broadcast to the γ_1 processors within each subcube-column. After synchronous exchanges of d_2 messages of $\lceil \frac{n-i}{\gamma_1} \rceil$ reals and d_1 messages of $\lceil \frac{n-i}{\gamma_2} \rceil$ reals, all 2^d processors can now update their local data *concurrently.*

2.2 Parallel partial pivoting step

The implementation we propose here employs "explicit permutation" of pivot rows. As noted in our earlier work [3], the explicit permutation of pivot rows balances the work load as well as maintains the wrap mapping for the next stage of computation. *The real issue is how to do it without incurring extra communication.* We show below that by employing redundant but *free* permutations in the subcube doubling technique we can maintain data proximity and accomplish the explicit permutation using exactly the same synchronous communication steps as before. While there is no increase in communication cost at all, we emphasize that there is virtually *no* extra computing cost either. To help explain the idea of "free redundant permutation", we demonstrate in Table 2 how the first reduction step works on a 4×4 subcube-grid while factoring an 8×8 matrix. Considering the 4×4 subcube-grid as configured in Fig. 4, the processors shown in Table 2 are those in the second subcube-column, namely P_1, P_5, P_9 and P_{13}.

Table 2: Parallel partial pivoting step.

P_i	Pivot column & Local data	Message 1	Explicit permutation	Message 2	The last permutation	Local data
P_1	a_{11} a_{12} a_{16}	a_{11} a_{12} a_{16}		a_{61} a_{62} a_{66}	a_{31} a_{32} a_{36}	a_{32} a_{36}
	a_{51} a_{52} a_{56}				a_{51} a_{52} a_{56}	a_{52} a_{56}
P_5	a_{21} a_{22} a_{26}		a_{21} a_{22} a_{26}			a_{22} a_{26}
	a_{61} a_{62} a_{66}	a_{61} a_{62} a_{66}	a_{11} a_{12} a_{16}	a_{61} a_{62} a_{66}		a_{12} a_{16}
P_9	a_{31} a_{32} a_{36}	a_{31} a_{32} a_{36}		a_{31} a_{32} a_{36}	a_{61} a_{62} a_{66}	a_{62} a_{66}
	a_{71} a_{72} a_{76}				a_{71} a_{72} a_{76}	a_{72} a_{76}
P_{13}	a_{41} a_{42} a_{46}	a_{41} a_{42} a_{46}		a_{31} a_{32} a_{36}		a_{42} a_{46}
	a_{81} a_{82} a_{86}					a_{82} a_{86}

Recall that the γ_2 processors in each of the γ_1 subcube-rows have the same segment of the pivot column. Each processor can thus select the local pivot and the corresponding segment of the pivot row. We shall assume that in case of ties, the lower numbered row will be chosen as the local pivot row. In the i^{th} reduction step, our algorithm requires the processors who were assigned the i^{th} row to permute its segment with the local pivot row segment so that the initial local winner is from row i. The reason will be clear later. Thus, each of the γ_1 processors in each subcube-column holds one segment from the γ_1 different local pivot rows initially. Using our example in Table 2 assuming $|a_{1,1}| > |a_{5,1}|$, $|a_{2,1}| < |a_{6,1}|$, $|a_{3,1}| > |a_{7,1}|$ and $|a_{4,1}| > |a_{8,1}|$, the four local pivot rows are row 1, row 6, row 3 and row 4. Note that $|a_{1,1}| > |a_{5,1}|$ is ensured by the initially permutation in P_1.

In the proposed subcube-grid communication scheme, the γ_2 subcube-columns perform d_1 exchanges of messages *independently* and *concurrently*. The message is initially composed of the local pivot row segment. The initial messages are shown as Message 1 in Table 2 for our example. After each exchange, each processor updates its message by the segment with "larger" local pivot; in addition, γ_2 processors will *explicitly permute* their segments if the winner originates from them and the segment with "smaller" pivot is from row "i" in the i^{th} stage of computation. Note that after each message exchange, the processors involved in the explicit permutation have had both segments they need, so there is *no* extra communication cost. Furthermore, the permutation is absolutely free of any computing cost because the processors who need to overwrite their data by the current (losing) row i are those who own the current winner and they do not need to update the message itself. On the other hand, the processors who own the (losing) row i do not need to update their data until after the last exchange because their message always contains the most recent row i chosen from the permutation. This step can be seen from the "Explicit permutation" column in

Table 2, where we assume $|a_{6,1}| > |a_{1,1}|$ and therefore processor P_5 overwrites its local data by the segment from row 1, while processor P_1 simply updates its message content to be the segment from row 6. Therefore, only the updated local data of processor P_5 is shown in the "Explicit permutation" column. The same explicit permutation is performed by processors P_4, P_6 and P_7 which, together with P_5, consist of the second subcube-row.

Consequently, before each message exchange, there is always one segment from the i^{th} row during the i^{th} reduction step. After d_1 exchanges, all processors will have the corresponding segments coming from the same two rows – one of them is the final row i. The last explicit permutation can thus be performed by the processors who were originally assigned row i and those who were assigned the other row simultaneously. Referring to our example in Table 2, after the exchange of Message 2 between processors P_1 and P_9, and between P_5 and P_{13}, all four processors have the same two segments from row 3 and row 6. Assuming $|a_{3,1}| > |a_{6,1}|$, and recall that row 6 has taken the role of updated row 1, our algorithm thus requires processor P_1 to overwrite its segment from row 1 by the final winner from row 3, and processor P_9, which was originally assigned data from row 3, overwrites its segment from row 3 by the segment $\{a_{6,1}, a_{6,2}, a_{6,6}\}$ from row 6, which is the most up-to-date row 1. We display on the left in Fig. 7 the matrix resulting from *multiple* redundant permutations. For easy comparison, we show on the right in Fig. 7 the matrix resulting from one permutation as done in the sequential Gaussian elimination process. In both cases, only the rows of coefficients which have been relocated are shown explicitly, the other coefficients are marked by "$\times$" for simplicity.

$$\begin{pmatrix} a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ \times & \times & \times & \times & \times & \times & \times & \times \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} & a_{67} & a_{68} \\ \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times \\ a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times \end{pmatrix} \quad \begin{pmatrix} a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} & a_{38} \\ \times & \times & \times & \times & \times & \times & \times & \times \\ a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} & a_{18} \\ \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times \end{pmatrix}$$

Figure 7: The matrices resulting from multiple and single permutations.

Finally, we note that *no* explicit permutation will be performed if the matrix is positive definite. This is the immediate result from the observation that when the winner is row i in the i^{th} reduction step, no permutation will be made using the algorithm proposed above.

2.3 Communication complexity result

The communication complexity can be immediately obtained from [4] by setting $m = n$ and $\gamma_1 = \gamma_2 = \sqrt{p}$, assuming $p = 2^d$ and d is an even number. We thus obtain the total communication cost as

$$T(n,p,\lambda,\beta) \quad = \quad 2n\log_2 p\beta + \frac{n^2\log_2 p}{\sqrt{p}}\lambda + O\left(n\log_2 p\right) \ , \qquad (1)$$

where β is the start-up time for sending a message and λ is the time needed to transmit a floating-point number across one link between adjacent processors in the hypercube. Each exchange is counted as sending two messages sequentially in our analysis. The communication cost can thus be further halved if the communication hardware supports concurrent bidirectional transmission.

3 Parallel QR factorization with column pivoting

To incorporate column pivoting into the subcube-grid algorithm in [4], the communication scheme is exactly the same as we proposed in §2 for Gaussian elimination with partial pivoting except for applying the redundant (free) permutations to select the pivot column segment instead of the pivot row segment. However we must note that the criterion for choosing the local pivot column is now different and it uses the "column norm". To avoid communication in updating the column norms after each reduction step, we make use of the important property that the two-norms of vectors are *invariant* under orthogonal transformation. The modification to our original algorithm [4] is quite minor as described below.

First, we compute the initial n column norms and distribute them to the appropriate processors along with the data; i.e., the processor which is assigned data from the i^{th} column will be sent the initial norm of the i^{th} column. Therefore, all of the γ_1 processors in each subcube-column have the initial norms of the same $\lceil \frac{n}{\gamma_2} \rfloor$ columns. This amounts to distributing one extra row segment to each processor, and the cost is not significant if $\frac{m}{\gamma_1} \gg 1$, where m is the row dimension of the input matrix.

Secondly, we observe that since the norm is invariant under orthogonal transformation, the norm of the remaining $(n-i)$ elements in each column can be computed by simply subtracting the norm of the single i^{th} row element from the current norm. This can be accomplished without communication because every processor receives the corresponding segment from the i^{th} row after the last message exchange during the i^{th} reduction step in the original algorithm [4].

We can thus conclude that the communication complexity remains the same as our results in [4], and that the extra computing cost incurred in updating the norms is evidently small.

4 Multiple least squares updating

The implementation of the *multiple* least squares updating algorithm on the hypercube was first studied by Kim et.al. in [14]. The proposed algorithms in [14] wrap-mapped the Cholesky factor and the incoming rows to the p processors either following the processor *id*s or the processor order of an embedded linear array. Thus the communication cost in updating an $n \times n$ Cholesky factor by annihilating the input $m \times n$ matrix is $O\left(n^2 \log_2 p\right)$ in either case. We obtain the same result in [4] when employing the hypercube as a $p \times 1$ subcube-grid in reducing an $m \times n$ matrix to upper triangular form via orthogonal transformation. Although the communication cost is *independent* of the row dimension m of the input matrix and it would be the optimal choice if $m \gg n$, it is not the desirable algorithm otherwise. Due to the usually stringent time constraint in engineering applications, it is rare for $m \gg n$ when the Cholesky factor must be updated by the newly available data. Thus, it is very important that the parallel implementation be flexible and efficient for all cases. Since the algorithm we proposed in [4] was designed to work equally well for matrices of all possible dimensions, it is certainly a good candidate for multiple least squares updating on the hypercube. We show here that by using a different data mapping strategy for the computed Cholesky factor, we obtain a parallel algorithm which not only always chooses the optimal aspect ratio for the embedded subcube-grid according to the row and column dimensions of the newly available data but also allows *dynamic* relocation of the Cholesky factor if the aspect ratio changes. We also show that the new data mapping strategy maintains the work load as balanced as before.

Let us express the multiple least squares updating problem by

$$\begin{pmatrix} R_{n\times n} \\ A_{m\times n} \end{pmatrix} \longrightarrow \begin{pmatrix} \tilde{R}_{n\times n} \\ 0 \end{pmatrix},$$

where $\tilde{R}$ results from modifying the upper triangular factor R by zeroing out all elements in the $m \times n$ matrix A via orthogonal transformations. We describe the new data mapping strategy and the dynamic data relocation scheme in the following sections.

4.1 New data mapping strategy

Instead of wrap mapping the rows and columns of $R_{n\times n}$ and $A_{m\times n}$ to the $\gamma_1 \times \gamma_2$ subcube-grid, we propose *block* mapping for the rows while maintaining wrap mapping for the columns. That is, each subcube-row of processors contains a block of $\lceil \frac{n}{\gamma_1} \rceil$ consecutive rows from $R_{n\times n}$ and a block of $\lceil \frac{m}{\gamma_1} \rceil$ rows from $A_{m\times n}$. With each subcube-row, the data from each block are wrap mapped to the γ_2 processors in the subcube according to their column subscripts exactly as proposed in [4].

We want to emphasize here that the wrap mapping of rows was necessary to maintain balanced work load in reducing a full $m \times n$ matrix to its upper triangular form, but it is no longer necessary if the entire matrix A is zeroed out by updating an available triangular R. The reasons are two-fold:

1. In the latter case, all elements in the same column of $A_{m\times n}$ are modified the same number of times regardless of the row number.

2. The subcube-doubling communication algorithm as adapted in [4] has the unique feature that all processors always perform $\log_2 p$ synchronous exchanges of messages regardless of the location of the pivot row. Consequently, the fact that each processor owns consecutive pivot rows from R will not affect the work load distribution at all.

4.2 A dynamic parallel data relocation scheme

Since the row dimension of the data matrix A varies with the amount of data available at different times, the subcube-grid is to be dynamically configured to achieve the optimal aspect ratio and the new input matrix will be distributed to the processors accordingly.

When the aspect ratio differs from the one used in the previous update, it becomes necessary to *relocate* the previously distributed Cholesky factor R. The mapping strategy we proposed above allows us to relocate R to the new $\tilde{\gamma_1} \times \tilde{\gamma_2}$ subcube-grid efficiently. The following observations are the keys to the design of the parallel algorithm.

1. $\tilde{\gamma_1} = 2^{\tilde{d_1}}$, and $\tilde{\gamma_2} = 2^{\tilde{d_2}}$, where $\tilde{d_1} + \tilde{d_2} = d$, and d is the dimension of the hypercube.

2. $\tilde{\gamma_1} = \gamma_1 \times 2^i$, and $\tilde{\gamma_2} = \gamma_2 \times 2^{-i}$, for $-d_1 \le i \le d_2$. By this observation, we simply need an algorithm for $i = 1$, because the same algorithm can be executed $i > 1$ times to get the resulting distribution. This observation contributes to the simplicity of the algorithm.

3. The sequential ordering of processors on the subcube-grid following the processor *id* row by row plays an important role in the parallel algorithm.

4.2.1 The parallel algorithm

Since the row dimension of the input matrix is initially known to the host, we require the host to compute the new aspect $\tilde{\gamma_1}$ and send it to all p node processors in the hypercube together with the data from input matrix A. Therefore, there is *no* extra communication involved in broadcasting the value $\tilde{\gamma_1}$ to all processors in the subcube-grid, and $\tilde{\gamma_2} = p/\tilde{\gamma_1}$ can then be computed by everyone. The parallel algorithm can be best explained by an example. We consider the case of changing a 4×4 subcube-grid to an 2×8 subcube-grid as shown below.

$$\begin{array}{cccc} P_0 & P_1 & P_2 & P_3 \\ P_4 & P_5 & P_6 & P_7 \\ P_8 & P_9 & P_{10} & P_{11} \\ P_{12} & P_{13} & P_{14} & P_{15} \end{array} \longrightarrow \begin{array}{cccccccc} P_0 & P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 \\ P_8 & P_9 & P_{10} & P_{11} & P_{12} & P_{13} & P_{14} & P_{15} \end{array}$$

Since the columns of factor R were wrap mapped to the processors in each subcube row, processors P_0, P_1, P_2 and P_3 can simultaneously send half of the data (corresponding to appropriate columns) to P_4, P_5, P_6 and P_7. Since a block of consecutive rows are stored in each processor, processor P_4, P_5, P_6 and P_7 will send one half of their data (corresponding to appropriate columns) to processors P_0, P_1, P_2 and P_3 at the same time. The processors P_8, P_9, P_{10} and P_{11} will understandably exchange same amount of data with processors P_{12}, P_{13}, P_{14}, and P_{15} at the same time. Again because of the subcube-grid connectivity, this amounts to "one" synchronous message exchange between a pair of directly connected processors. Each message consists of half of the data the processor previously has and thus data distribution remains balanced.

4.2.2 Communication complexity results

We can thus conclude that for $i = 1$, the communication cost is one near-neighbor exchange of one message of $(n^2/2p)$ floating-point numbers. Since $i \leq d$, the communication volume for relocating R is $((n^2 \log_2 p)/2p)$ in the worst case.

References

[1] A. Gerasoulis, N. Missirlis, I. Nelken and R. Peskin. Implementing Gauss Jordan on a hypercube multiprocessor. In *The third conference on hypercube concurrent computers and applications*, 1569–1576. The Association for Computing Machinery, 1988.

[2] R. M. Chamberlain. An algorithm for LU factorization with partial pivoting on the hypercube. In Michael T. Heath, editor, *Hypercube Multiprocessors Proceedings 1987.* The Society for Industrial and Applied Mathematics, Philadelphia, 1987.

[3] E. C. H. Chu and J. A. George. Gaussian elimination with partial pivoting and load balancing on a multiprocessor. *Parallel Computing*, 5:65–74, 1987.

[4] E. C. H. Chu and J. A. George. QR factorization of a dense matrix on a hypercube multiprocessor. *SIAM J. Sci. Stat. Comput.*, 11, July 1990 (to appear).

[5] D. W. Walker, T. Aldcroft, A. Cisneros, G. C. Fox, and W. Furmanski. LU decomposition of banded matrices and the solution of linear systems on hypercubes. In *The*

third conference on hypercube concurrent computers and applications, 1635–1655. The Association for Computing Machinery, 1988.

[6] G. J. Davis. Column LU factorization with pivoting on a hypercube multiprocessor. Technical Report 6219, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, 1985.

[7] G. Fox, W. Furmanski, and D. Walker. Optimal Matrix Algorithm on Homogeneous Hypercubes. In *The third conference on hypercube concurrent computers and applications*, 1656–1673. The Association for Computing Machinery, 1988.

[8] G. A. Geist. Efficient parallel LU factorization with pivoting on a hypercube multiprocessor. Technical Report ORNL-6290, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, 1985.

[9] G. A. Geist and M. T. Heath. Parallel Cholesky factorization on a hypercube multiprocessor. Technical Report ORNL-6211, Oak Ridge National Laboratory, Oak Ridge, TN, 1985.

[10] G. A. Geist and M. T. Heath. Matrix factorization on a hypercube multiprocessor. In M. T. Heath, editor, *Hypercube Multiprocessors*, Philadephia, PA., 1986. SIAM Publications.

[11] G. A. Geist and C. H. Romine. LU factorization algorithms on distributed-memory architectures. In G. Rodrigue, editor, *Parallel processing for scientific computing*, pages 15–18. The Society for Industrial and Applied Mathematics, 1989.

[12] M. T. Heath. Parallel Cholesky factorization in message passing multiprocessor environments. Technical Report ORNL-6150, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, 1985.

[13] S. L. Johnsson. Communication efficient basic linear algebra computations on a hypercube architecture. Technical Report YALEU/DCS/RR-361, Yale University, Department of Computer Science, November 1988.

[14] S. Kim, D. P. Agrawal, and R. J. Plemmons. Recursive least squares filtering for signal processing on distributed memory multiprocessors. Technical Report Manuscript draft, Department of Electrical and Computer Engineering and Department of Computer Science and Mathematics, March 1988.

[15] C. Moler. Matrix computation on distributed memory multiprocessors. In M. T. Heath, editor, *Hypercube multirpocessors 1986.* The Society for Industrial and Applied Mathematics, 1986.

[16] Youcef Saad. Communication complexity of the Gaussian elimination algorithm on multiprocessors. *Linear Algebra and its Appl.*, 77:315–340, 1986.