THE TORUS–WRAP MAPPING FOR DENSE MATRIX CALCULATIONS ON MASSIVELY PARALLEL COMPUTERS*

BRUCE A. HENDRICKSON‡ AND DAVID E. WOMBLE‡

Abstract.
Dense linear systems of equations are quite common in science and engineering, arising in boundary element methods, least squares problems and other settings. Massively parallel computers will be necessary to solve the large systems required by scientists and engineers, and scalable parallel algorithms for the linear algebra applications must be devised for these machines. A critical step in these algorithms is the mapping of matrix elements to processors. In this paper, we study the use of the *torus–wrap mapping* in general dense matrix algorithms, from both theoretical and practical viewpoints. We prove that, under reasonable assumptions, this assignment scheme leads to dense matrix algorithms that achieve (to within a constant factor) the lower bound on interprocessor communication. We also show that the torus–wrap mapping allows algorithms to exhibit less idle time, better load balancing and less memory overhead than the more common row and column mappings. Finally, we discuss practical implementation issues, such as compatibility with BLAS levels 1, 2, and 3, and present the results of implementations of several dense matrix algorithms. These theoretical and experimental results are compared with those obtained from more traditional mappings.

Key words. LU factorization, QR factorization, Householder tridiagonalization, parallel computer, dense matrix, torus–wrap mapping

AMS subject classifications. 65Y05, 65F05

1. Introduction.

Dense linear systems of equations are quite common in science and engineering applications, appearing in boundary element methods, problems involving all–pairs interactions and least squares problems, among others. The kernel computation for these applications usually involves some factorization of the matrix to transform it to a more convenient form. The factored matrix can then be used to solve linear systems of equations, perform least squares calculations, determine eigenvectors and eigenvalues, or whatever other computation the application requires [19].

For a dense $n \times n$ matrix, most of these factorization algorithms require $\Theta(n^3)$ floating point operations and $\Theta(n^2)$ storage. Current sequential supercomputers can store and operate on systems with tens of thousands of unknowns. For example, a single processor CRAY Y–MP with 256 megawords of memory operating at its peak theoretical speed of 333 million floating point operations per second (Mflop/s) can compute the LU factorization of a $16{,}000 \times 16{,}000$ matrix in about 2.3 hours. Solving substantially larger problems or sequences of moderately sized problems on even the fastest single processor machines is prohibitively time consuming. It is clear from the computational requirements that to solve such problems will require computers capable of billions of floating point operations per second. This, in turn, will require massively parallel computers based on scalable architectures. To effectively use these machines, algorithms must be devised that scale well to large numbers of processors.

Efficient use of massively parallel computers is a subject of much current research, and numerous papers have been published about dense linear algebra algorithms on these machines. Because of its importance for solving linear systems, the LU factorization (and the related triangular solve) has been the primary subject of this research

* This work was supported by the Applied Mathematical Sciences program, U.S. Department of Energy, Office of Energy Research, and was performed at Sandia National Laboratories, operated for the U.S. Department of Energy under contract number DE–AC04–76DP00789.
Appeared in *SIAM J. Sci. Stat. Comput.* 15(5):1201–1226, 1994.

‡ Sandia National Laboratories, Albuquerque, NM, 87185

[1, 2, 4, 5, 11, 12, 16, 17, 21, 26, 31, 32]. There have been far fewer papers concerned with the efficient computation of the QR factorization, Householder tridiagonalization, or the eigenvalue problem [7, 8, 22, 37]. Fewer still have tried to address dense matrix algorithms in general.

Early implementations of dense matrix algorithms, and in particular the LU factorization, mostly used row or column decompositions in which entire rows or columns of the matrix were assigned to individual processors [17, 18, 21, 26]. The columns or rows that a processor owned were usually "wrapped" or scattered throughout the matrix to obtain good load balancing. On computers with 64 to 128 processors, the efficiencies of these algorithms were usually between 50% and 75% for the largest problems that could be stored on the machines, but the algorithms did not scale particularly well as the number of processors increased [17, 21].

An alternative method for assigning matrix elements to processors is the torus–wrap mapping. Variants of this assignment scheme have been independently discovered by several researchers, and consequently given a number of different names including cyclic [23], scattered [15], grid [36], and subcube–grid [8], as well as torus–wrap [30]. The mapping was first described by O'Leary and Stewart in a data-flow context [29, 30], and the synergy between the torus–wrap mapping and the hypercube topology was observed by Fox [14, 15]. Variants of the torus–wrap mapping have been used in high performance LU factorization codes on a number of different machines [3, 4, 6, 9, 27, 35, 36]. Assuming each matrix element is stored on only a single processor, Ashcraft built on work by Saad to show that for LU factorization, the torus–wrap mapping exhibits communication properties within a constant factor of optimal [1, 32]. (Ashcraft has recently devised an algorithm with lower order communication that violates this nonduplication assumption, but it requires an impractical factor of $p^{1/3}$ additional storage, where p is the number of processors [2].) Various algorithms for QR factorization employing the torus–wrap mapping have been described that use Givens rotations [8], modified Gram–Schmidt [37] and Householder reflections [22, 27]. A torus-wrap mapping algorithm for Householder tridiagonalization is described in [7]. A triangular solve algorithm using this mapping that achieves asymptotically optimal performance is presented in [5, 28]. Because of its scaling properties, the torus–wrap mapping has been suggested as the basic decomposition for parallel dense linear algebra libraries [13].

Despite the evident recent popularity of the torus–wrap mapping for a number of different dense linear algebra implementations, a careful analysis of the strengths and weaknesses of the mapping has been lacking. One purpose of this paper is to provide such an analysis, including communication overhead, memory requirements and load balancing issues. Our approach is to identify the critical computation and communication components of dense matrix operations on distributed memory computers and then to analyze the impact of different mappings on the performance of these components. Thus, the results in this paper are more general than much of the current literature, and we anticipate that our analysis will provide a basis for future research in this area.

Another purpose of this paper is to explore the practical aspects of implementations of dense matrix algorithms using the torus–wrap mapping. Three algorithms are actually implemented for this end, LU factorization, QR factorization and Householder tridiagonalization. Using these implementations, we compare the performance of a range of torus–wrap mappings with that of the row–wrap and column–wrap mappings, and we examine the scalability of the torus–wrap mapping to large numbers of processors using numerical results obtained on a 1,024–processor nCUBE 2. We also present models of performance for these three implementations that allow us to examine such effects as communication/computation overlap and the effect of vector lengths on communications and on the BLAS operations. These models then allow us to predict the optimal torus–wrap decompositions.

In §2, we characterize the basic operations required for dense matrix algorithms and their implications in the parallel computing environment; deriving lower bounds on required inter-processor communication. We define the torus–wrap mapping in §3 and describe its relationship to more familiar decomposition schemes. In §4, we discuss in more detail the properties of the torus-wrap decomposition, including its communication requirements, scalability, and compatibility with the standard BLAS routines. We present data from implementations of several different dense matrix algorithms in §5. These results clearly show the advantages of the torus–wrap mapping. Conclusions are presented in §6.

2. Dense linear algebra operations and communication. Nearly all dense linear algebra algorithms consist of a sequence of fundamental operations that transform a matrix into some more desirable form. The two most important such operations are Gauss transformations and Householder reflections. These operations generally dominate the computational effort in a dense linear algebra algorithm, so their efficient execution is essential for good performance. On a message passing multiprocessor the execution time of an algorithm can depend greatly upon its communication patterns, so minimizing the communication for these fundamental operations is important for achieving good parallel performance [23, 24, 33].

To understand the communication required to perform Gauss and Householder transformations, consider Fig. 1, where A is an $m \times n$ matrix, u is an m–vector and v an n–vector. Under either a Gauss transformation or a Householder reflection (or a Gauss–Jordan transformation), each element of A is updated by the outer–product of u and v; that is $A_{ij} \leftarrow A_{ij} + u_i v_j$. The difference between the algorithms is in the construction of u and v, which is a lower order operation in both computation and communication. The outer–product update of an element of A depends upon the element of u directly to its left and the element of v above it. The processor that calculates the new value for A_{ij} must know the old A_{ij} as well as u_i and v_j, which may require some communication. We will establish a lower bound on the total communication volume for this operation, which is a subset of the communication required to perform a Gauss or Householder transformation.

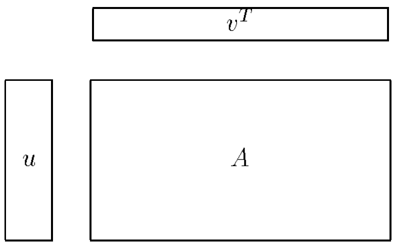

Fig. 1. Structure of Gauss and Householder transformations.

We denote by $N(q)$ the number of matrix elements owned by processor q, and let p be the total number of processors. We will assume that
(i) each element of A (and of u and v) is owned by a single processor, and
(ii) the matrix elements are balanced; that is, for each processor q, $N(q) \geq \alpha mn/p$

for some constant $\alpha > 0$.

We define the *communication volume*, V_C, of an algorithm to be the total length of all the messages the algorithm requires. For numerical algorithms, messages typically consist of floating point numbers, so the lengths are most naturally measured in terms of number of floating point values. The following theorem is a generalization of results found in [32].

THEOREM 2.1. *Under Assumptions* (i) *and* (ii) *above, the communication volume required to execute a Gauss, Householder or Gauss–Jordan transformation is at least* $2\sqrt{\alpha pmn} - (m + n)$.

Proof. Each element in the matrix needs the value of v above it and the value of u to its left. Let t_i^r denote the number of processors owning elements of row i and t_j^c be the number of processors owning elements of column j. To transmit u_i (or v_j) to all the processors in row i (or column j) requires at least $t_i^r - 1$ (or $t_j^c - 1$) messages of length 1, so the communication volume can be bounded by

$$V_C \geq \sum_{i=1}^{m}(t_i^r - 1) + \sum_{j=1}^{n}(t_j^c - 1). \tag{1}$$

Now we denote by s_q^r (and s_q^c) the number of rows (and columns) of which processor q owns at least one element. It follows from the definitions that $\sum_{q=1}^{p} s_q^r = \sum_{i=1}^{m} t_i^r$, and $\sum_{q=1}^{p} s_q^c = \sum_{j=1}^{n} t_j^c$. Substituting these identities into (1) yields

$$\begin{aligned} V_C &\geq -(m+n) + \sum_{q=1}^{p}(s_q^r + s_q^c) \\ &\geq -(m+n) + \sum_{q=1}^{p} 2\sqrt{s_q^r s_q^c}. \end{aligned}$$

Assumption (ii) ensures that for each q, $s_q^r s_q^c \geq N(q) \geq \alpha mn/p$, so

$$\begin{aligned} V_C &\geq -(m+n) + \sum_{q=1}^{p} 2\sqrt{\alpha mn/p} \\ &= 2\sqrt{\alpha pmn} - (m+n). \square \end{aligned}$$

COROLLARY 2.2. *Under Assumptions* (i) *and* (ii) *above, the communication volume required to execute a Gauss, Householder or Gauss–Jordan transformation on a square* $n \times n$ *matrix is at least* $2n(\sqrt{\alpha p} - 1)$.

Ashcraft has recently proposed an LU factorization algorithm that requires $\Theta(p^{1/3}n^2)$ communication volume, but it violates Assumption (i) [2]. This algorithm is impractical in its current form, requiring an extra factor of $p^{1/3}$ storage.

The lower bound expressed in Theorem 2.1 is attainable, up to a constant factor. For simplicity we let m and n be divisible by $\sqrt{p}$, and assume that $m+n \leq \gamma \min(m, n)$ for some constant γ. If we assign each processor a dense rectangular block of the matrix of size $(m/\sqrt{p}) \times (n/\sqrt{p})$, then $\alpha = 1$, and each row and each column will be owned by only $\sqrt{p}$ of the processors. The total communication volume involved in broadcasting a row will be $n(\sqrt{p} - 1)$, and for a column $m(\sqrt{p} - 1)$, implying a total of $(n + m)(\sqrt{p} - 1) \leq \gamma\sqrt{mnp} - (m + n)$, which is within a constant factor of the bound from Theorem 2.1.

We note that if each column (or row) of the matrix is owned by a single processor, then a Gauss or Householder transformation requires the broadcast of that column (or row) to all other processors. This involves a communication volume of $m(p-1)$ (or $n(p-1)$). Assuming again that $m+n \leq \gamma \min(m,n)$ for some constant γ, this volume is at least $(2/\gamma)\sqrt{mn}(p-1)$, which is larger than the lower bound by $\Theta(\sqrt{p})$.

Finally, we observe that the results in this section are a consequence of the fact that dense linear algebra operations can be formulated to require only a restricted form of communication. Values must be exchanged within each row of the matrix and within each column. Any operation that involves this communication pattern will be amenable to a similar analysis.

3. The torus–wrap mapping. Most of the previous work on parallel dense linear algebra has involved assigning elements of the $m \times n$ matrix A to processors using columns, rows or blocks. In a column (or row) scheme, entire columns (or rows) of the matrix are assigned to a single processor. One possibility is to have columns 1 through n/p assigned to processor zero, columns $n/p+1$ through $2n/p$ assigned to processor one and so on. Since most matrix factorizations work from left to right, decreasing the number of active columns, this scheme has the disadvantage that processor q has no work left to do after column $(q+1)n/p$ is processed. For this reason, it is preferable to assign columns $1, p+1, 2p+1, \ldots$ to processor zero, columns $2, p+2, 2p+2, \ldots$ to processor one, and so forth to form what is known as a *column–wrap* mapping. Column–wrap (and row–wrap) mappings have been the most widely used choice for dense linear algebra algorithms, but as noted in §2, row and column methods require $\Theta(\sqrt{p})$ more communication volume than necessary. For machines with a small number of processors, the simplicity of these mappings may outweigh the communication drawbacks, but for massively parallel machines this factor of $\sqrt{p}$ can be very important.

In block schemes, each processor is assigned a single dense rectangular submatrix. Many different block mappings are possible involving differently shaped rectangular submatrices and different assignments of blocks to processors. As mentioned in §2, block schemes can come within a constant of achieving the lower bound on communication volume. However, they have the same problems with idle processors that the nonwrapped row and column methods have. Hybrid *block–column–wrap* or *block–row–wrap* mappings are also possible. In these mappings, instead of owning a scattered set of single columns (or rows), a processor owns a scattered set of several adjacent columns (rows).

An analogy with row and column methods suggests wrapping a block mapping in both rows and columns. The result is what we will call the *torus–wrap* mapping. As there are many different block mappings, there are correspondingly many torus–wrap mappings. If the number of processors, p, can be factored as a product of p_r and p_c, then we can construct a block mapping in which the blocks are of size $(m/p_r) \times (n/p_c)$. For any appropriate p_r and p_c values, we get a block mapping and its torus–wrap counterpart. We note that in the limiting cases the torus–wrap mapping reduces to a row–wrap mapping (when $p_r = p$ and $p_c = 1$), or a column–wrap mapping (when $p_c = p$ and $p_r = 1$). We call the special case in which $p_r = p_c$ a *square* torus–wrap mapping. If it is not the case that m is divisible by p_r and n by p_c then some processors will own one more row and/or column than others.

Another choice in a block mapping occurs when deciding which blocks get assigned to which processors. Because communication in dense linear algebra algorithms occurs predominantly within rows or within columns, it is convenient to assign blocks in such

0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
8	9	10	11	8	9	10	11
12	13	14	15	12	13	14	15
16	17	18	19	16	17	18	19
20	21	22	23	20	21	22	23
24	25	26	27	24	25	26	27
28	29	30	31	28	29	30	31
0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7

Fig. 2. Processors owning matrix elements in a natural torus–wrap.

a way that communication among the set of processors owning a row (or column) is efficient. With mesh architectures, this is achieved by constructing a set of blocks that reflect the shape of the processor array. If the mesh is constructed as a rectangle of $p_r \times p_c$ processors, then the blocks are of size $(m/p_r) \times (n/p_c)$, which are assigned to processors in the natural way. The rows and columns are then wrapped to generate the corresponding torus–wrap assignment. This constitutes what we will call a *natural* torus–wrap mapping, and ensures that row (or column) communication occurs entirely within rows (or columns) of the processor mesh. An example of the natural torus–wrap is depicted in Fig. 2, where the mesh is of size 8×4, and the value displayed at each location is the processor that owns the corresponding matrix entry.

For hypercubes, we can exploit the fact that a d–dimensional hypercube can be viewed as the product of two hypercubes of dimensions d_r and d_c, where $d_r + d_c = d$. This is accomplished by dividing the bits of the processor identifier into two sets, b_r and b_c of cardinality d_r and d_c respectively. The bits of b_r can be associated with the row numbering of the matrix elements, and those of b_c with the column numbering. That is, all the processors owning elements from a single row of A have the same b_r bits, and all processors with elements from a column have the same b_c bits. This assignment scheme ensures that communication within a row involves changing only bits of b_c, while communication within a column involves only bits of b_r. So each row of the matrix lies within a subcube of dimension d_c, and each column in a subcube of dimension d_r. In this way, a total of $p_c = 2^{d_c}$ processors are assigned elements of each row, and $p_r = 2^{d_r}$ processors are assigned elements of each column.

One method for assigning processor numbers on a hypercube is to take the row number of a matrix element, subtract one, express the result modulo p_r, and then take the Gray code of the resulting d_r bit number. (For a discussion of Gray codes, see [15].) Performing the analogous calculation on the column bits, and assigning matrix elements to the resulting processors generates the *Gray–coded* torus–wrap, which is discussed in detail by Chu and George in [8]. An example corresponding to Fig. 2 is depicted in Fig. 3. The advantage of the Gray–coded torus–wrap is that neighboring elements of the matrix are owned by neighboring processors. More formally, a Gray–coded torus–wrap embeds a p_r by p_c mesh into a hypercube.

Figures 2 and 3 indicate another way to define the torus–wrap mapping. The assignment pattern of the leading p_r by p_c submatrix defines a tile, and the entire matrix is covered by copies of this tile in the obvious way.

An important generalization of the torus–wrap mapping is the *block–torus–wrap*, in which the matrix is first decomposed into a collection of blocks of size $\delta_1 \times \delta_2$. Each block is assigned to a single processor, in such a way that the distribution of

0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6
12	13	15	14	12	13	15	14
8	9	11	10	8	9	11	10
24	25	27	26	24	25	27	26
28	29	31	30	28	29	31	30
20	21	23	22	20	21	23	22
16	17	19	18	16	17	19	18
0	1	3	2	0	1	3	2
4	5	7	6	4	5	7	6

Fig. 3. Processors owning matrix elements in a Gray–coded torus–wrap.

blocks mirrors the distribution of elements in a torus–wrap mapping. We note that the torus–wrap is a special case of the block–torus–wrap in which $\delta_1 = \delta_2 = 1$. The block–torus is a natural generalization of block–row and block–column methods. Using blocks instead of single elements has both advantages and disadvantages, as will be discussed in the next section.

It is convenient to notice that for all the mappings considered above, each processor is assigned precisely the matrix elements that lie in the intersection of a particular set of rows and columns. Bisseling and van de Vorst call mappings with this property *Cartesian* [4]. Merely specifying a set of row indices and column indices for each processor uniquely defines a Cartesian mapping. Under a block mapping, a processor is assigned a consecutive set of row indices and a consecutive set of column indices. In a torus–wrap mapping, the row indices assigned to a processor constitute a linear sequence separated by p_r, and the column indices a sequence with step size p_c. In a row (or column) mapping, each processor is assigned all of the column (or row) indices and a subset of the row (or column) indices.

We conclude this section by observing that a matrix distributed among processors in a torus–wrap format can be viewed as a permutation of a matrix distributed in a block scheme. Specifically, a matrix A_b distributed in a block–wrap format can be treated as $\Pi_1^T A_t \Pi_2$, where Π_1^T and Π_2 are permutation matrices and A_t is a matrix distributed in a torus–wrap format. This equivalence allows for a different interpretation of the torus–wrap mapping. As observed in [27], a standard factorization algorithm on A_t can be viewed as a factorization of A_b in which the rows and columns are eliminated in a permuted order. An important result of this observation is that is possible to take advantage of the attractive properties of a torus–wrap mapping without necessarily having to redistribute the matrix among processors. For example, a block system can be solved by a routine that assumes a torus–wrapped system, and the result is correct up to permutations. More precisely, $A_b^{-1}x = \Pi_2^T A_t^{-1} \Pi_1^T x$, so a block mapping can be made to perform like a torus–wrap mapping, without redistributing the matrix, by merely permuting the right hand side and the solution vectors.

4. Virtues of the torus–wrap mapping. The torus–wrap mapping has a number of distinct advantages over the more conventional assignment schemes. These virtues become increasingly important as the number of processors increases. Generally, the torus–wrap mapping requires less communication than row or column schemes and it has excellent load balancing properties. These advantages will be discussed in detail in the following subsections, and their impact on performance of a collection of linear algebra algorithms will be presented in §5. Many of these issues are familiar to researchers who have used the torus–wrap mapping and have been touched upon in a number of publications describing specific implementations.

4.1. Communication volume. The square torus–wrap mapping allows Gauss and Householder transformations to be executed with a total communication volume of $\Theta(\sqrt{mnp})$, which is within a constant factor of optimal. Row and column schemes require about a factor of $\sqrt{p}$ more communication. We note, however, that if the matrix or the number of processors is small, message startup time dominates the message transmission time so that this factor of $\sqrt{p}$ is not seen. Also, when p is small, the reduced communication volume may not compensate for the increased complexity of the torus–wrap. The direction of high performance computing is towards large problems and massive parallelism, so the factor of $\sqrt{p}$ will become increasingly important.

Most linear algebra algorithms involve a sequence of about $\bar{m}$ Gauss or Householder transformations, where $\bar{m} = \min(m, n)$. This implies an overall communication volume of $\Theta(\bar{m}\sqrt{pmn})$ for torus–wrap, and $\Theta(p\bar{m}\sqrt{mn})$ for row and column mappings. The number of floating point operations (flops) is typically $\Theta(\bar{m}mn)$, which, if balanced, implies $\Theta(\bar{m}mn/p)$ per processor.

If we wish to solve larger problems by increasing the number of processors without increasing the memory of each processor, then the largest problem we can solve has $mn = cp$ for some constant c. This implies that the number of flops per processor is $\Theta(\bar{m})$ and the torus–wrap communication volume is $\Theta(p\bar{m})$, while the row or column communication volume is $\Theta(p^{1.5}\bar{m})$. All contemplated interconnection networks for massively parallel machines have $\Omega(p)$ wires, so the torus–wrap communication requirements have the potential to scale well. However, no proposed network has $\Omega(p^{1.5})$ wires, so row and column schemes will eventually be limited by communication. Similar but more detailed analyses for LU factorization can be found in [1, 4].

This advantage is not specific to Gauss and Householder transformations. The proof of Theorem 2.1 can be applied to any operation that requires communication within rows and columns, implying that such operations can be performed with near optimal communication volume using a torus–wrap mapping.

4.2. Communication parallelism. Although helpful for scaling analyses, communication volume may not be a particularly useful metric for performance modeling because it ignores any overlap in the communication operations. It is often the case that several messages can be transmitted simultaneously on different communication channels. To correctly predict performance, the times required by these overlapping messages should not be added. We define the *effective communication volume* of an algorithm to be the total length of all the messages that are not overlapped. The effective communication volume is a good estimate of the time required for communication operations when most messages are long so message startup time is negligible. This is the case for most dense linear algebra algorithms on large matrices.

With a torus–wrap mapping, the transmission of a column involves p_r overlapping broadcasts to $p_c - 1$ other processors. Assuming a logarithmic broadcast and a message length of m/p_r, this results in an effective communication volume of $d_c m/p_r$. Similarly, the transmission of a row requires an effective volume of $d_r n/p_c$. When $d_c = d_r = d/2$, the combined effective volume is $d(m+n)/(2\sqrt{p})$, but for the limiting cases of row or column mappings, it is dn or dm respectively. If m and n are about equal, the square torus–wrap mapping has an effective communication volume of about a factor of $\sqrt{p}$ less than that of row or column mappings. When the matrix is not square, or the row and column communication loads are not exactly equal, a nonsquare torus–wrap mapping may be best. This will be the case for some of the examples we consider in §5.

Since it roughly approximates communication time, the effective communication volume allows us to investigate the proportion of execution time devoted to communication. For dense factorizations of square matrices the sequential operation count is usually $\Theta(n^3)$, so if the load is well balanced the time spent performing these operations should be $\Theta(n^3/p)$. A dense factorization typically requires $\Theta(n)$ Gauss or Householder transformations, so the ratio of effective communication volume to parallel operation time is $\Theta((d_r p_r + d_c p_c)/n)$. For row or column schemes, this implies that the ratio of communication time to compute time grows as $\Theta(dp/n)$, but for a square torus it only grows as $\Theta(d\sqrt{p}/n)$. Assuming that each processor has finite memory, n can only grow as $\sqrt{p}$. In this case, the relative cost of communication scales as $\Theta(d\sqrt{p})$ for row and column mappings, but as $\Theta(d)$ for a square torus. The proportion of time spent on communications grows much more slowly for a square torus than for row or column mappings, allowing scalability to much larger machines.

For hypercubes, there are sophisticated broadcast schemes that manage to overlap communications more effectively than simple logarithmic broadcasts [24]. These algorithms use all of the wires in a cube (or subcube) to perform a broadcast. Using these algorithms, the row and column transmissions involve a combined effective communication volume of $m/p_r + n/p_c$. When n and m are about equal, a square torus is still better than a row or column method by about a factor of $\sqrt{p}/2$.

4.3. Message queue overhead. Row or column schemes require broadcasts of entire rows or columns of the matrix. Torus–wrap methods only require broadcasts of subsets of rows and columns with lengths n/p_c and m/p_r. To exploit the advantages of asynchronous communications, a processor that expects to receive a message must have space reserved for it. The amount of reserved space is less for a square torus than for a row or column mapping by a factor of about $\sqrt{p}$. This leaves more space that can be devoted to other things, like storing matrix elements. Consequently, the torus–wrap mapping allows larger problems to be solved than row or column mappings.

With block–torus–wrap methods, sets of blocks are broadcast together. The total communication volume is unchanged, as is the effective volume, but since sets of columns (or rows) are broadcast together, the number of startups is decreased and the lengths of messages are increased by a factor of δ_2 (or δ_1). This reduces the total communication time, but increases the amount of memory required for communication.

4.4. BLAS compatibility. Much work has been done developing a standard set of basic linear algebra subprograms or BLAS, which are available as a high performance library on many machines [10, 25]. The BLAS were devised with dense vectors or matrices in mind, but with the torus–wrap mapping each row and column is scattered. Although processors do not own any consecutive portion of the matrix, the submatrix assigned to each processor is rectangular and can be stored in a dense matrix format. For all the operations required for Gauss, Householder and Gauss–Jordan transformations, this submatrix can be treated as dense, which allows the use of levels one, two and three BLAS operations.

4.5. Load imbalance. To achieve optimal performance from a parallel computer, it is important that each processor has nearly the same total amount of work to do; that is the computational load must be balanced. For most dense matrix algorithms, the total number of floating point operations that have to be performed to update matrix element $A(i,j)$ is proportional to $\min(i,j)$. We will define the load of element $A(i,j)$ as $\min(i,j)$ and the load on a processor as the total load of all the elements it owns. Under a torus–wrap mapping, if A is an $m \times n$ matrix, the most heavily loaded processor will be the one owning element $A(m,n)$, and the least loaded will be the processor that would own $A(m+1,n+1)$ if A were larger. We denote the former processor by q and the latter by s, and let their loads be $W(q)$ and $W(s)$ respectively. We are interested in $\Delta = W(q) - W(s)$.

We observe that except for entries in the last row or last column of A, each element owned by q has a neighbor down and to the right that is owned by s. Similarly, with the possible exception of elements in the first row or first column, each value owned by s has a neighbor owned by q to the upper left. Each such pair of elements contributes a value of -1 to Δ. If we denote the number of pairs by N_p, then

$$\Delta = -N_p + W_{m,n}(q) - W_{1,1}(s),$$

where $W_{i,j}(x)$ is the load on processor x from elements in the row i and elements in the column j of the matrix.

We denote the first row partially owned by processor q in the torus–wrap mapping as r_0, and the corresponding first column as c_0. We assume for concreteness that $m \geq n$; the other case is analogous. The values of N_p and $W_{1,1}(s)$ are easy to compute. To compute $W_{m,n}(q)$ we separately sum the contributions from elements in the last row, elements in the last column that are above the diagonal, and the remaining elements in the last column on or below the diagonal. Some algebra gives the following result.

$$\begin{aligned}
N_p &= (\frac{m-r_0}{p_r})(\frac{n-c_0}{p_c}) = (m-r_0)(n-c_0)/p, \\
W_{1,1}(s) &= \lfloor c_0/p_c \rfloor \lfloor m/p_r \rfloor + \lfloor r_0/p_r \rfloor \lfloor n/p_c \rfloor - \lfloor c_0/p_c \rfloor \lfloor r_0/p_r \rfloor, \\
W_{m,n}(q) &= (\frac{n+c_0}{2})(\frac{n-c_0}{p_c}+1) + \lceil \frac{n-r_0}{p_r} \rceil (r_0 + \frac{p_r}{2}(\lceil \frac{n-r_0}{p_r} \rceil - 1)) + n \lfloor \frac{m-n}{p_r} \rfloor .
\end{aligned}$$

In the limit as m, n, and p become large $W_{1,1}$ becomes negligible, as do r_0 and c_0, so Δ approaches $n^2/(2p_c) + n(m-n/2)/p_r - mn/p$. (If $n > m$ the corresponding result is $\Delta = m^2/(2p_r) + m(n-m/2)/p_c - mn/p$.) For a square matrix, this maximal load imbalance reduces to $n^2(p_r+p_c-2)/(2p)$. So for a square matrix, a square torus–wrap induces less load imbalance than a row–wrap or column–wrap by a about a factor of $\sqrt{p}/2$. A similar conclusion is reached for parallel LU factorization by Bisseling and van de Vorst in [4].

Assuming that each processor has finite memory, then for square matrices n can grow as $\sqrt{p}$. Since the number of flops grows as n^3, the run time should scale roughly as n^3/p, which is proportional to $\sqrt{p}$. The maximum load imbalance scales as $p_r + p_c$, which is p for row or column mappings, but $\sqrt{p}$ for a square torus–wrap. Consequently, the proportional load imbalance increases as $\sqrt{p}$ for row or column mappings, but stays constant for a square torus–wrap.

Similar results apply for a block–torus mapping. For simplicity in the analysis, we assume that $\delta_2 = \delta_1 = \delta$, and that n and m are both divisible by δ. In this case, we can generalize the previous analysis by counting blocks instead of single elements. Letting the N_p and $W_{i,j}$ notation now apply to entire blocks, N_p scales linearly with δ because the workload imbalance associated with a pair of diagonally adjacent blocks increases as δ^3, but the number of blocks decreases as δ^2. Similarly, $W_{m,n}$ increases by about a factor of δ, while $W_{1,1}$ increases by about a factor of $\delta^2/2$. Since the the contribution of $W_{1,1}$ to Δ is negative, the growth in the load imbalance with delta is at most linear. For large m, n and p, and small δ, the N_p and $W_{m,n}$ terms will dominate $W_{1,1}$, so the load imbalance from a block–torus scheme with square blocks will approach $\delta\{n^2/(2p_c) + n(m - n/2)/p_r - mn/p\}$, where $m \geq n$. The important conclusion is that if the block sizes are small, the maximal load imbalance is proportional to the linear dimension of the blocks.

4.6. Processor idle time. Even if all the processors have the same total amount of work to do, the overall calculation will be inefficient unless each processor always has something to work on. Algorithms using Gauss or Householder transformations usually begin by computing a function of the first column (or row) of the matrix; the norm for Householder and the largest element for Gauss. If this first column (row) is not distributed, as in a column (row) mapping, then the other processors need to wait until this calculation is complete. This problem is exacerbated by using a block–column (or block–row) approach to allow level three BLAS, since several columns (rows) must be manipulated before the other processors have any work to do. A torus–wrap mapping allows some parallelism in the processing of each column or row which reduces this potential idle time problem. With a block–torus–wrap mapping, the idle time at the beginning of the calculation grows linearly with δ_1 or δ_2.

Most factorizations eliminate the rows and/or columns of the matrix in order, from left to right and top to bottom. As the factorization nears completion, only processors that still own active matrix elements have work to do. With column (or row) schemes, processors begin to drop out when there are p columns (rows) remaining. Block methods are even worse. With the torus–wrap, however, all the processors stay active until there are only p_c columns (or p_r rows) left to eliminate. If the matrix has p more (fewer) rows than columns then a row–wrap (column–wrap) mapping avoids this problem, but for the important special case of square and nearly square matrices, the torus–wrap mapping allows greater parallelism near the end of the computation. As at the beginning, with a block–torus–wrap the idle time at the end of the calculation grows linearly with δ_1 or δ_2.

5. Numerical results. In this section, we discuss the implementations of three dense matrix algorithms, LU factorization, QR factorization, and Householder tridiagonalization. For each of these examples, we investigate the performance implications of using different mapping schemes, both analytically and experimentally. All numerical experiments were performed in double precision C on a 1,024 node nCUBE 2 hypercube at the Massively Parallel Computing Research Laboratory at DOE's Sandia National Laboratory.

Our algorithms require only very simple communication patterns consisting of broadcast, collect, and binary exchange operations. We implemented each of these functions in a simple, generic way so that the resulting code should run on any architecture. Although our timings given are for a specific computer, the broad conclusions should be appropriate for other machines. For hypercubes there are asymptotically more efficient communication algorithms as was alluded to in §4.2, but our implementations do not exploit them.

5.1. LU factorization. Our first example is LU factorization with partial pivoting of a dense $n \times n$ matrix A. This the most important factorization in linear algebra as it is a very efficient method for solving systems of linear equations. There are several variants of LU factorization, each requiring $2n^3/3 + O(n^2)$ flops. Our algorithm uses the column–oriented, kij version as described in [19]. The algorithm is summarized in Fig. 4, where Roman subscripts denote integers and Greek subscripts denote sets of integers.

Processor q owns row set α and column set β
For $j = 1$ to n
    (∗ Find pivot row ∗)
    If $j \in \beta$ **Then**
        (∗ Compute maximum of entries in column j ∗)
        $\gamma^q := \max_{i \in \alpha} |A_{i,j}|$
        Binary exchange to compute $\gamma = \max_q \gamma^q$
        $s :=$ index of row containing the entry γ

    (∗ Generate update vector, v, from column j of A ∗)
    If $j \in \beta$ **Then**
        $A_{\alpha,j} := A_{\alpha,j}/\gamma$
        $v_\alpha := A_{\alpha,j}$
        Broadcast column v_α and s to processors sharing rows α
    Else Receive v_α and s

    (∗ Exchange pivot row and diagonal row, and broadcast pivot row ∗)
    If $j \in \alpha$ **Then**
        Send $w_\beta = A_{j,\beta}$ to processor owning $A_{s,\beta}$
    If $s \in \alpha$ **Then**
        Receive w_β
        $u_\beta := A_{s,\beta}$
        Broadcast row u_β to processors sharing columns β
        $A_{s,\beta} := w_\beta$
    Else Receive u_β
    If $j \in \alpha$ **Then**
        $A_{j,\beta} := u_\beta$

    If $j \in \alpha$ and $j \in \beta$ **Then**
        $A(j,j) = A(j,j) * \gamma$ (∗ Restore diagonal ∗)

    $\alpha := \alpha \setminus \{j\}$ (∗ Remove j from active rows ∗)
    $\beta := \beta \setminus \{j\}$ (∗ Remove j from active columns ∗)

    $A_{\alpha,\beta} := A_{\alpha,\beta} - v_\alpha u_\beta$

Fig. 4. Parallel LU factorization for processor q.

Our implementation of this algorithm incorporates double precision arithmetic and uses a Gray–coded torus–wrap mapping. As presented, this algorithm can be used with any Cartesian mapping, but in practice, block algorithms would be modified to send fewer, larger messages. The algorithm can be easily modified for a block–torus–wrap mapping, but since nCUBE only supports level one BLAS, we did not investigate this possibility. Improvements in performance have been observed on other machines using block algorithms [35]. To minimize the time spent waiting for the determination and broadcast of pivot elements our algorithm employs a compute–ahead technique. The processors owning the next column in the matrix generate and send the pivot information before updating the remainder of their elements.

The nCUBE 2 has 1,024 processors arranged in a ten dimensional hypercube. Each processor has four Mbytes of memory, which must be divided between the operating system, code, data and communication buffer. Using the optimal distribution of data among the processors, we can allocate about 3.8 Mbytes for data storage in our code. This means that the largest double precision, dense matrix that can be factored in core is about 22,000 × 22,000. Factoring the Linpack benchmark matrix of this size requires 3612 seconds (1.96 Gflop/s), using a processor decomposition in which $p_c = 64$ and $p_r = 16$, which is the optimal decomposition of 1,024 processors for this size problem as will be seen later in this section.

To investigate the effect of the different torus–wrap decompositions on scaling we need to use a matrix that can be stored on fewer than 1,024 processors. We factored an $8,000 \times 8,000$ matrix with different numbers of processors and the results are presented in Table 1. We note that although we can store a matrix of size $n = 11,000$ on 256 processors, the communication buffer requirements of the nonoptimal row and column distributions require that we reduce the size of the matrix, as was discussed in §4.3.

Table 1. Run times on the nCUBE 2 for LU factorization of an 8,000 × 8,000 matrix.

256 processors			512 processors			1,024 processors		
p_c	p_r	seconds	p_c	p_r	seconds	p_c	p_r	seconds
1	256	2568	1	512	2406	1	1,024	2394
2	128	1335	2	256	1064	2	512	953
4	64	930	4	128	624	4	256	483
8	32	777	8	64	462	8	128	307
16	16	719	16	32	396	16	64	237
32	8	706	32	16	372	32	32	207
64	4	729	64	8	376	64	16	201
128	2	806	128	4	412	128	8	215
256	1	996	256	2	506	256	4	260
			512	1	712	512	2	364
						1,024	1	589

We can model the performance to gain insight into the scaling properties and the optimal balance between p_c and p_r. There are four major contributions to the run time of the LU factorization code: a pivot entry search and pivot column update, a row broadcast, a column broadcast, and an outer–product update of the unfactored submatrix. We will need the following values taken from the nCUBE 2 manuals for our model.

Variable	Description	Microseconds
$T_{c,a}$	message startup time	100.
$T_{c,b}$	transmission time per double precision number	4.00
$T_{p,a}$	startup time for pivot operations	16.3
$T_{p,b}$	computational time for pivot operations (per element)	1.955
$T_{d,a}$	startup time for a daxpy	11.0
$T_{d,b}$	computational time for a daxpy (per element)	0.964

With these variables, the total time spent computing the outer–product updates of the unfactored portion of the submatrix using the column–oriented daxpy can be modeled as

$$T_{update} = \frac{n^2}{2p_c} T_{d,a} + \frac{n^3}{3p} T_{d,b}.$$

The time spent on pivot entry searches, which includes a local search and a binary exchange by those processors containing part of a column, and pivot column updates can be approximated by

$$T_{pivot} = nT_{p,a} + \frac{n^2}{2p_r} T_{p,b} + d_r n(2T_{c,a} + 3T_{c,b}),$$

where we recall that d_r and d_c are the dimensions of the subcubes to which columns and rows are assigned, respectively. The time for a logarithmic broadcast of the pivot row is about

$$T_{row} = d_r \left(nT_{c,a} + \frac{n^2}{2p_c} T_{c,b} \right).$$

Finally, the time for a logarithmic broadcast of the pivot column is approximated by

$$T_{column} = d_c n T_{c,a} + \frac{n^2}{2p_r} T_{c,b}.$$

The total run time, T_{total}, is modeled by summing the four contributions above.

We note an important difference between the contributions to the overall run time of the row broadcasts and the column broadcasts. If the processor holding column $j+1$ is the first processor to complete its portion of the broadcast of the pivot column, then the search for the pivot and update of column $j+1$ can be overlapped with the remaining stages of the broadcast of column j. When a Gray–coded torus–wrap mapping is used on a hypercube, the processors holding column $j+1$ are neighbors of those owning column j. Consequently, a logarithmic broadcast can be structured in such a way that the processors owning column $j+1$ quickly receive column j and then have no further participation in the broadcast. Thus the factor d_c multiplies only the startup time in the contribution to the total run time. The row broadcast, on the other hand, cannot be overlapped with any computations (without destroying the load balance of the algorithm) so that d_r multiplies both the startup and transmission time contributions to the total run time.

The data from Table 1 is shown graphically in Fig. 5, where instead of the run time, the vertical axis is the Mflop/s rate achieved. To generate these values, we used the number of flops required by the sequential algorithm, about 3.41×10^{11}.

We observe that the optimal distribution is not achieved at $d_c = d_r$. Qualitatively, the major reason for this is the fact that column broadcasts can be overlapped with computations while row broadcasts cannot. T_{update} and T_{pivot} also contribute to this phenomenon since the startup time for column operations is reduced by increasing d_c, and the communication time required to find the pivot element (which is the dominant term in T_{pivot}) is reduced by decreasing d_r.

This observation can be examined quantitatively by looking at the model. Specifically, if we write T_{total} in terms of n, d, d_c and the timing constants, differentiate with respect to d_c and set the resulting expression equal to zero, we obtain the equation

$$2^{2d_c} \left(\frac{T_{p,b} + T_{c,b}}{2^d} \right) - 2^{d_c} \left(\frac{4T_{c,a} + 6T_{c,b})}{n \ln 2} \right) - \left(T_{d,a} + \left(d - d_c + \frac{1}{\ln 2} \right) T_{c,b} \right) = 0.$$

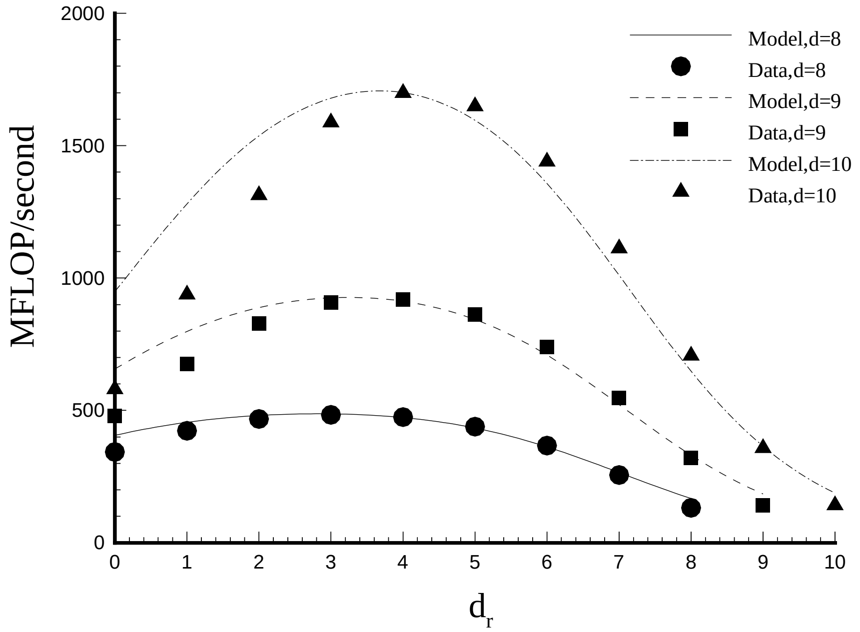

Fig. 5. Performance of the LU factorization code on the nCUBE 2.

We cannot solve this equation in closed form, but, in practice, simple numerical techniques can be used to compute the optimum value of d_c. Here, we derive an approximate expression for the optimum value of d_c. First, we observe that errors in the linear term involving d_c have only a small effect in the final solution and replace $(d - d_c + 1/\ln 2)T_{c,b}$ with $dT_{c,b}/2$. Second, we observe that for most cases of interest $2^{2d_c - d} \gg 2^{d_c}/n$ so that the second term can be dropped from the expression. Now, solving for d_c yields

$$d_c = \frac{d}{2} + \frac{1}{2}\log_2\left(\frac{T_{d,a} + dT_{c,b}/2}{T_{p,b} + T_{c,b}}\right). \tag{2}$$

We note that even though this value is approximate, the predictions are close to the observed optima as shown in Fig. 5. The predicted peak performances (for integer values of d_c) are 487 Mflop/s for $d = 8$ and $d_r = 3$, 925 Mflop/s for $d = 9$ and $d_r = 3$ and 1700 Mflop/s for $d = 10$ and $d_r = 4$. We see from (2) that the optimum value of d_c is shifted from the value $d/2$ by an amount that depends on the startup time for the daxpy operation, the computation time for the pivot search operation and the amount of row communication that cannot be overlapped with computation, but this shift is relatively small on the nCUBE 2 because the constants involved are of similar magnitudes.

One way to improve the performance of the code for large d_r would be to use the BLAS to update rows (in the outer–product update) rather than columns. This would have the effect of increasing the vector lengths for the BLAS calls and reducing the number of startups. We did not incorporate this improvement into the code because the optimum performance occurs for $d_r < d/2$ and would not be affected by the switch. This improvement will be discussed in more detail in §5.2 dealing with QR factorization.

As parallel machines are built with more and more processors, it becomes possible to solve larger and larger problems. An important metric of parallel algorithms is how they scale to larger problems on more processors [20]. We will assume the amount of memory available to a processor remains constant, so the largest dense matrix that can be stored on a machine has $n^2 = cp$ for some constant c. We define *scaled speedup* to be the ratio of computation rate to the single processor computation rate for problems in which the number of matrix elements per processor remains constant. We note that this definition of scaled speedup is based on holding the memory requirements on each processor constant, not the workload on each processor. Because the LU factorization of a dense matrix requires $O(n^3)$ flops but only $O(n^2)$ storage, holding the work load per processor constant would result in lower scaled speedup values.

The results of a set of runs to determine scaled speedups are presented in Table 2. The first column of the table gives the linear dimension of the square matrix, and the second the dimension of the hypercube. The third column contains the values of d_c and d_r that proved optimal. The fourth column shows the observed total number of Mflops per second of execution time, where a flop count of $2n^3/3$ is used. The fifth column divides the fourth by the number of processors. Scaled speedups are presented in the sixth column. The last column presents *efficiencies*, which we define to be the scaled speedup divided by the number of processors. We note that the efficiencies greater than one in the second and third rows of the table are due to the fact that the BLAS operations have greater efficiency with longer vectors.

Table 2. Times for LU factorizations of scaled matrices.

Matrix Size	Cube Dim	$d_r : d_c$	Mflop/s Observed	Mflop/s per Proc.	Scaled Speedup	Eff.
500	0	0 : 0	1.94	1.94	1.00	1.00
707	1	0 : 1	3.94	1.97	2.03	1.01
1,000	2	0 : 2	7.84	1.96	4.04	1.01
1,414	3	1 : 2	15.41	1.93	7.94	0.99
2,000	4	1 : 3	30.78	1.92	15.87	0.99
2,828	5	2 : 3	60.99	1.91	31.44	0.98
4,000	6	2 : 4	121.95	1.91	62.86	0.98
5,657	7	3 : 4	242.40	1.89	124.95	0.97
8,000	8	3 : 5	483.50	1.89	249.23	0.97
11,314	9	4 : 5	963.50	1.88	496.65	0.97
16,000	10	4 : 6	1917.19	1.87	988.24	0.96

In §4.1, we discussed the complexity of the communication volume and concluded that use of the torus–wrap mapping results in better scaling of an algorithm than either the row–wrap or the column–wrap mapping. This is born out by Fig. 6, where the performance of row–wrap and column–wrap mappings are compared to the optimal torus–wrap mapping for these scaled problems.

5.2. QR factorization. Our second example is Householder QR factorization without pivoting of an $m \times n$ matrix A. After LU, QR is probably the most important factorization in linear algebra, and is used in least squares problems, eigen–problems, basis generation and other settings. The total flop count for this algorithm is $2n^2(m - n/3) + O(n^2)$ for the usual situation in which $m \geq n$ [19]. Our parallel algorithm is outlined in Fig. 7 and is described in greater detail in [22]. Alternative QR algorithms that use the torus–wrap mapping can be found in [8, 37].

As with the LU implementation described in §5.1, to minimize time spent waiting for a Householder vector to be computed and broadcast, the processors that share the next column to be processed generate and broadcast their elements of the Householder

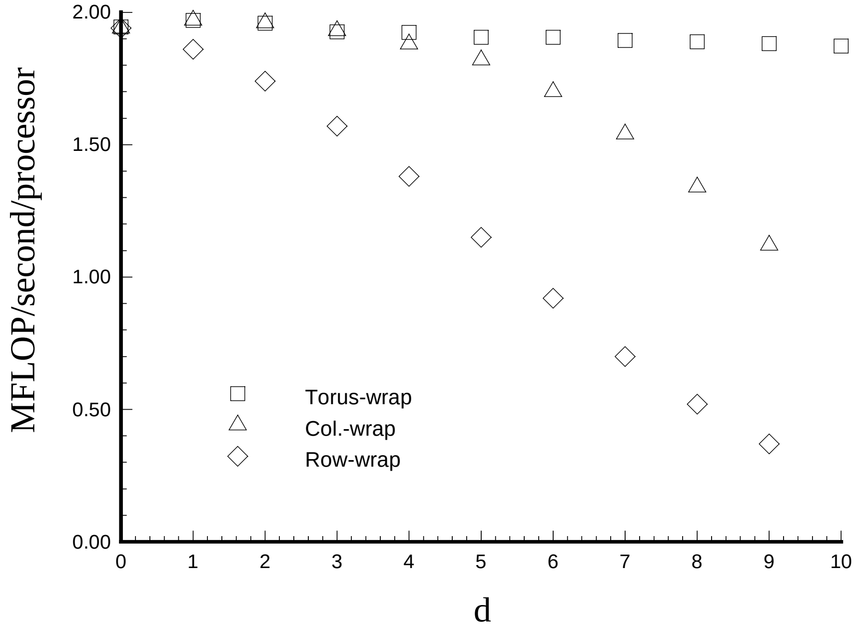

Fig. 6. Mflop/s/processor rates for LU factorization with different processor mappings.

vector before updating the rest of their matrix elements. Run times for the same set of problem sizes considered in §5.1 are presented in Table 3.

Table 3. Run times on the nCUBE 2 for QR factorization of an 8,000 × 8,000 matrix.

256 processors			512 processors			1,024 processors		
p_c	p_r	seconds	p_c	p_r	seconds	p_c	p_r	seconds
1	256	3105	1	512	2612	1	1,024	2515
2	128	2087	2	256	1532	2	512	1324
4	64	1643	4	128	1042	4	256	779
8	32	1463	8	64	834	8	128	537
16	16	1428	16	32	756	16	64	436
32	8	1381	32	16	729	32	32	408
64	4	1384	64	8	713	64	16	381
128	2	1437	128	4	736	128	8	383
256	1	1559	256	2	814	256	4	418
			512	1	996	512	2	507
						1,024	1	715

The total run time for this algorithm can be modeled as the sum of the times for computing Householder vectors, broadcasting Householder vectors, generating inner–products and performing outer–product updates. In addition to those parameters from §5.1, our model requires the following values from the nCUBE 2 manuals.

Variable	Description	Microseconds
$T_{n,a}$	startup time for a 2–norm	13.0
$T_{n,b}$	computational time for 2–norm (per element)	0.863

Processor q owns row set α and column set β
For $j = 1$ to n
    (* Generate Householder vector, v, from column j of A *)
    If $j \in \beta$ **Then**
        (* Compute contribution to norm of column j *)
        $\gamma^q := A^T_{\alpha,j} A_{\alpha,j}$
        Binary collapse $\gamma = \sum \gamma^q$ to processor owning $A_{j,j}$
        If $j \in \alpha$ **Then**
            $\tau = 2(\gamma + |A_{j,j}|\sqrt{\gamma})$
            $A_{j,j} := A_{j,j} + \mathrm{sign}(A_{j,j})\sqrt{\gamma}$
        $v_\alpha := A_{\alpha,j}$
        Broadcast v_α (and τ) to processors sharing rows α
    Else
        Receive v_α (and τ)

    $\beta := \beta \setminus \{j\}$ (* Remove j from active columns *)

    $r^q_\beta := v^T_\alpha A_{\alpha,\beta}$ (* Compute portion of dot–products *)
    Binary exchange (with appended τ) among processors sharing
        columns β to compute $r_\beta := \sum r^q_\beta$
    $A_{\alpha,\beta} := A_{\alpha,\beta} - (2/\tau) v_\alpha r_\beta$ (* Update the submatrix *)

    $\alpha := \alpha \setminus \{j\}$ (* Remove j from active rows *)

Fig. 7. Parallel Householder QR for processor q.

To generate a Householder vector the processor owning the top element must know the vector's 2–norm. The total time spent performing this task consists of the time for the local computations followed by the time for a binary collapse to combine the partial sums, which can be approximated by

$$T_{norm} = nT_{n,a} + \frac{n(2m-n)}{2p_r} T_{n,b} + d_r n(T_{c,a} + T_{c,b}).$$

The total time spent broadcasting Householder vectors can be modeled as

$$T_{bcast} = d_c n T_{c,a} + \frac{n(2m-n)}{2p_r} T_{c,b}.$$

As with the LU factorization column broadcast, the generation of the next Householder vector overlaps with all but the first stage of the broadcast of the current one. Thus the factor d_c does not appear in the per element term of the broadcast.

The time for computing all the dot–products is the sum of the time required for the numerics, $T_{dot-calc}$, and the time for combining all the partial sums within each column $T_{dot-comm}$. Although for hypercubes there are asymptotically more efficient alternatives [15, 34], we perform this communication using a binary exchange which requires redundant numerical operations. The gains from the more sophisticated algorithms are of a low order and so insignificant for large problems, and by not exploiting hypercube specifics we can reach broader conclusions about the performance of torus–wrap algorithms.

The numerical operations associated with the dot–products can be performed as a daxpy within rows, or as a ddot within columns (which has about the same startup and per–element cost as a daxpy). Our implementation dynamically chooses whichever of these operations involves fewer startups. For simplicity of analysis, we ignore the fact that the optimal choice can change during a run for nonsquare matrices, in which case the dot–product terms are about

$$\begin{aligned} T_{dot-calc} &= \frac{n}{2}\min(\frac{n}{p_c}, \frac{2m-n}{p_r})T_{d,a} + \frac{n^2(m-n/3)}{2p}T_{d,b}, \\ T_{dot-comm} &= d_r n T_{c,a} + \frac{d_r n^2}{2p_c}T_{c,b}. \end{aligned}$$

Finally, the outer–product updates on submatrices can be performed using daxpys within either columns or rows. We again choose whichever leads to fewer startups, so the total update time can be approximated as

$$T_{update} = \frac{n}{2}\min(\frac{n}{p_c}, \frac{2m-n}{p_r})T_{d,a} + \frac{n^2(m-n/3)}{2p}T_{d,b}.$$

The data from Table 3 is shown graphically in Fig. 8, where the vertical axis is the Mflop rate achieved. To compute rates, we used the sequential flop count, which for this problem is about 6.83×10^{11} flops.

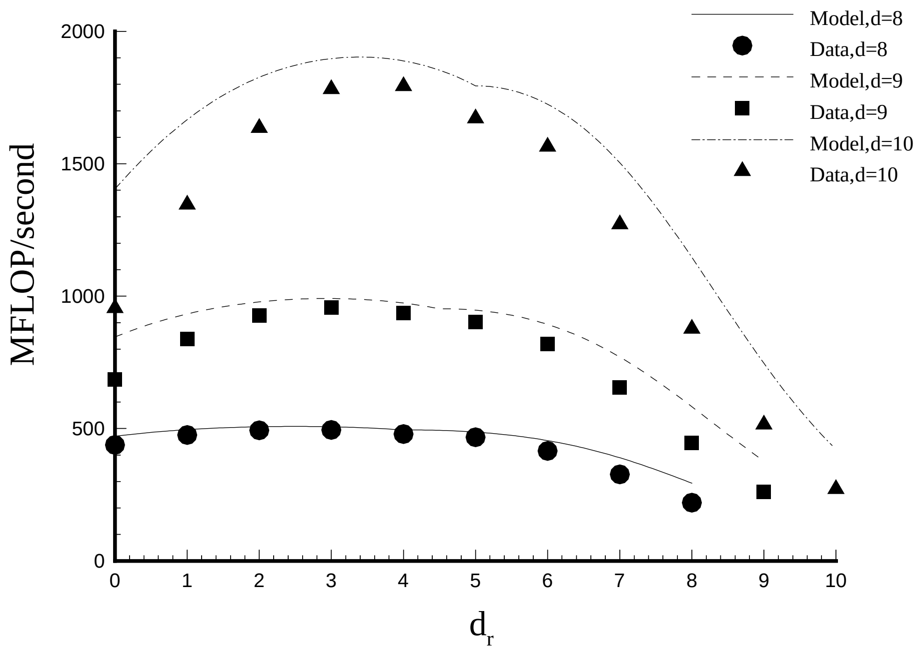

Fig. 8. Performance of the QR factorization code on the nCUBE 2.

The cusp in the curves is due to switching between a row oriented and a column oriented application of the level one BLAS. We expect the model to overpredict performance since by only including communication and BLAS it neglects any overheads. In the extreme case of column–wrapping, the model clearly overestimates the overlap of computation with communication. The model can be used to predict the optimal decomposition of processors among rows and columns by setting the derivative of the expression for run time to zero. However, as in §5.1, no closed form expression results. The peak performance is achieved when d_r is somewhat less than d_c due primarily to the greater communication within columns than within rows. A nonsquare torus also allows for fewer BLAS startups.

As in §5.1, we ran a sequence of factorizations in which the memory required per processor remained constant, allowing us to compute scaled speedups. The results are presented in Table 4. As before, the efficiencies greater than one are due to longer vectors in the BLAS routines.

Table 4. Times for QR factorizations of square matrices.

Matrix Size	Cube Dim	$d_r : d_c$	Mflop/s Observed	Mflop/s per Proc.	Scaled Speedup	Eff.
500	0	0 : 0	1.97	1.97	1.00	1.00
707	1	0 : 1	3.98	1.99	2.02	1.01
1,000	2	0 : 2	7.95	1.99	4.04	1.01
1,414	3	0 : 3	15.83	1.98	8.05	1.00
2,000	4	1 : 3	31.42	1.96	15.97	1.00
2,828	5	1 : 4	62.76	1.96	31.90	1.00
4,000	6	2 : 4	124.56	1.95	63.31	0.99
5,657	7	2 : 5	249.06	1.95	126.59	0.99
8,000	8	3 : 5	494.32	1.93	251.24	0.98
11,314	9	3 : 6	989.71	1.93	503.03	0.98
16,000	10	3 : 7	1963.91	1.92	998.01	0.97

Computation rates per processor for this set of problems are plotted in Fig. 9, comparing the optimal torus–wrap to row– and column–wrap mappings. As with LU, the torus–wrap mapping allows for much greater scalability than row or column schemes.

5.3. Householder tridiagonalization. Our third example, Householder tridiagonalization of a symmetric $n \times n$ matrix A, is used in eigenvector calculations of symmetric or Hermitian matrices. The best sequential algorithm requires $4n^3/3 + O(n^2)$ flops, and $n^2/2 + O(n)$ storage [19]. An algorithm that fails to exploit the symmetry of the matrix will require additional computation and memory, so it is better to store and manipulate only a lower (or upper) triangular part of the matrix. This causes problems for row or column oriented mappings since a particular row (or column) of the matrix is now stored partially as a row and partially as a column. The same issues arise with other algorithms for symmetric matrices, like Cholesky decomposition. However, since a square torus–wrap mapping treats rows and columns symmetrically, it is well suited to deal with this problem.

Our algorithm for square tori is outlined in Fig. 10. Each processor stores its portion of the lower triangular part of A in a column major format. One property of torus–wrap mappings as we have defined them is that for square tori the processors owning diagonal matrix elements are assigned a set of row indices equal to their set of column indices. This is also true for block torus–wrap mappings with square blocks, but it is not generally true of Cartesian mappings. This property makes transposing vectors easy, since these diagonal processors have the appropriate elements of both a vector and its transpose. We exploit this convenience in our implementation, which limits us to square tori. A similar algorithm is described in [7].

As evidenced by Fig. 10, exploiting symmetry makes this algorithm more complicated than those for LU and QR. First a Householder vector, v, must be generated and broadcast. The appropriate elements of the transpose of v (and later w) are then

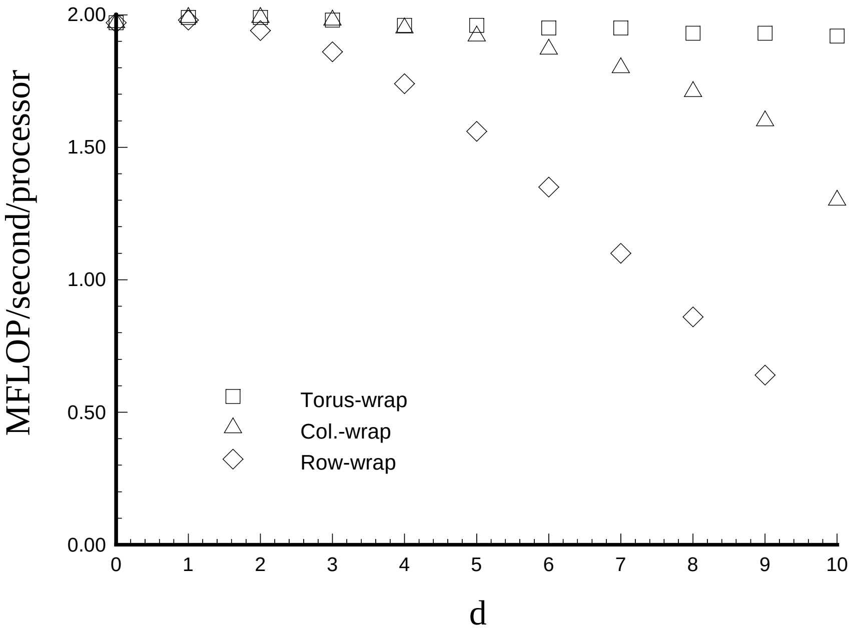

Fig. 9. Mflop/s/processor rates for QR factorization with different processor mappings.

communicated to each processor. Next comes the calculation of $s = \bar{A}v$, where $\bar{A}$ is the remaining submatrix. This is followed by the computation of $w = 2(s - (v^T s)v/\tau)/\tau$, where $\tau = ||v||^2$. Finally, an outer–product update of the matrix elements is performed.

We implemented this algorithm on the nCUBE 2 and used the code to factor an $8,000 \times 8,000$, double precision matrix, which required 1653 and 556 seconds on cubes of dimension eight and ten respectively.

The algorithm in Fig. 10 can be generalized to apply to non-square torus mappings, but this involves substantial complexity in performing the transpose operations and the matrix–vector multiplications. Although we did not implement this more general algorithm, we can develop a performance model to investigate the tradeoffs associated with different mappings. Our model will require the parameters introduced in the previous two sections and the following values from the nCUBE 2 manuals.

Variable	Description	Microseconds
$T_{s,a}$	startup time for a dscal	7.8
$T_{s,b}$	computational time for dscal (per element)	0.554
$T_{t,a}$	startup time for a ddot	10.0
$T_{t,b}$	computational time for ddot (per element)	0.984

Computing and broadcasting the Householder vectors is very similar to the operation required in Householder QR as described in §5.2.

$$T_{norm} = nT_{n,a} + \frac{n^2}{2p_r}T_{n,b} + d_r n(T_{c,a} + T_{c,b})$$

Processor q owns row set α and column set β of lower triangular A

For $j = 1$ to $n-1$

 $\alpha := \alpha \setminus \{j\}$ $(*$ Remove j from active rows $*)$

 $(*$ Generate Householder vector, v, from column j of A $*)$

 If $j \in \beta$ **Then**

  $\gamma^q := A^T_{\alpha,j} A_{\alpha,j}$

  Binary collapse $\gamma = \sum \gamma^q$ to processor owning $A_{j+1,j}$

  If $j+1 \in \alpha$ **Then**

   $\tau = 2(\gamma + |A_{j+1,j}|\sqrt{\gamma})$

   $A_{j+1,j} := A_{j+1,j} + \mathrm{sign}(A_{j+1,j})\sqrt{\gamma}$

  $v_\alpha := A_{\alpha,j}$

  Broadcast v_α (and τ) to processors sharing rows α

 Else Receive v_α (and τ)

 $\beta := \beta \setminus \{j\}$ $(*$ Remove j from active columns $*)$

 $(*$ Get elements of v^T to the correct processors $*)$

 If $\alpha = \beta$ **Then**

  $v_\beta := v_\alpha$

  Broadcast v_β to processors sharing columns β

 Else Receive v_β

 $(*$ Compute Av $*)$

 $r^q_\beta := A^T_{\alpha,\beta} v_\alpha$

 Binary collapse among processors sharing columns β

  to diagonal processor to form $r_\beta := \sum r^q_\beta$

 If $\alpha = \beta$ **Then**

  $s^q_\alpha := r_\alpha + A_{\alpha,\beta} v_\beta$ $(*$ excluding diagonal contribution $*)$

 Else $s^q_\alpha := A_{\alpha,\beta} v_\beta$

 Binary exchange among processors sharing rows α

  to form $s_\alpha := \sum s^q_\alpha$

 $(*$ Compute $v^T s$ and generate w $*)$

 $\eta^q := \sum_{i \in \alpha} v_i s_i$

 Binary exchange among processors sharing columns β

  to form $\eta := \sum \eta^q$ (and append τ)

 $w_\alpha := \frac{2}{\tau}(s_\alpha - \frac{\eta}{\tau} v_\alpha)$

 $(*$ Get elements of w^T to the correct processors $*)$

 If $\alpha = \beta$ **Then**

  $w_\beta := w_\alpha$

  Broadcast w_β to processors sharing columns β

 Else Receive w_β

 $A_{\alpha,\beta} := A_{\alpha,\beta} - v_\alpha w^T_\beta - w_\alpha v^T_\beta$ $(*$ Update the submatrix $*)$

Fig. 10. Parallel Householder tridiagonalization for processor q.

$$T_{bcast} = d_c n T_{c,a} + \frac{d_c n^2}{2p_r} T_{c,b}.$$

Unlike the models for LU and QR, we include the time for each stage of the broadcast. This is because the binary exchanges in the algorithm keep the processors tightly synchronized, which reduces the potential to overlap computation with communication.

For square tori, transposing v and w just requires broadcasts from the diagonal processors, but for non-square tori it is more complicated. We denote $p_{max} = \max(p_r, p_c)$, and $d_{max} = \max(d_r, d_c)$, with the obvious p_{min} and d_{min} counterparts. Transposition can be accomplished with d_{min} stages of a broadcast with message length about n/p_{max}, followed by $d_{max} - d_c$ stages of a binary exchange in which the message length doubles after each stage. We note that for non-square tori, some copying of data is also required. The total time spent performing these operations is about

$$T_{trans} = 2d_r n T_{c,a} + \frac{d_{min} n^2}{p_{max}} T_{c,b} + \frac{n^2}{p_{max}}(\frac{p_{max}}{p_c} - 1) T_{c,b}.$$

This formula is assymetric in rows and columns because the recursive doubling stage in the transpose need only occur if $d_r > d_c$. Otherwise, the last term in the expression reduces to zero.

Computing $\bar{A}v$ is somewhat problematic since only the lower triangular portion of the matrix is stored. We denote this triangular portion as L_1, and the portion of L_1 below the diagonal as L_2. We observe that $\bar{A}v = L_1^T v + L_2 v$, which requires communication in both rows and columns. Our algorithm first performs a ddot to determine the contribution from $L_1^T v$. These values are combined and sent to the processors owning the diagonal matrix elements using precisely the opposite of the communication pattern used above for transposition. Next, the contribution from $L_2 v$ is computed using a daxpy, and these values are combined across rows using a binary exchange. As a side effect, the binary exchange synchronizes the processors within each row and a total of about $(p_c - 1)n^2/2$ redundant flops are performed. As with our implementation of QR factorization, for hypercubes there are asymptotically more efficient alternatives to the binary exchange [15, 34], but our implementation does not use them. The calculation and communication time for this operation can be approximated as

$$\begin{aligned} T_{\bar{A}v-calc} &= \frac{n^2}{2p_c}(T_{t,a} + T_{d,a}) + \frac{n^3}{6p}(T_{t,b} + T_{d,b}), \\ T_{\bar{A}v-comm} &= (d_r + d_c) n T_{c,a} + \frac{n^2}{2p_{max}}(d_{min} + \frac{p_{max}}{p_c} - 1) T_{c,b} + \frac{d_c n^2}{2p_r} T_{c,b}. \end{aligned}$$

Forming $v^T s$ involves local computation, followed by a binary exchange among the processors sharing a set of column indices. This serves to synchronize the processors within each column, and requires a total of about $(p_r - 1)n^2/2$ redundant flops. Also, the same local computations are repeated by each column of processors, implying an overall additional $(p_c - 1)n^2$ flops beyond those in the sequential algorithm. The time for this computation and communication can be modeled as

$$\begin{aligned} T_{dot-calc} &= n T_{t,a} + \frac{n^2}{2p_r} T_{t,b} \\ T_{dot-comm} &= d_r n (T_{c,a} + T_{c,b}). \end{aligned}$$

Each processor can now generate its own elements of w, using a dscal followed by a daxpy. As above, this calculation is duplicated p_c times, resulting in about $2(p_c - 1)n^2$ extra flops. The time spent in this step is about

$$T_{genw} = n(T_{d,a} + T_{s,a}) + \frac{n^2}{2p_r}(T_{d,b} + T_{s,b}).$$

Updating $\bar{A}$ by $-vw^T - wv^T$ is now a local operation performed by each processor on its own data. Each column of $\bar{A}$ can be updated with two daxpys, so the time for this operation can be modeled as

$$T_{update} = \frac{n^2}{p_c}T_{d,a} + \frac{n^3}{3p}T_{d,b}.$$

The total time is modeled as the sum of the terms above. The predictions of the model for an $8,000 \times 8,000$ matrix are plotted in Fig. 11, with the observed values for square tori included for comparison. In computing rates, we use the sequential flop count, which is about 6.83×10^{11} for this problem.

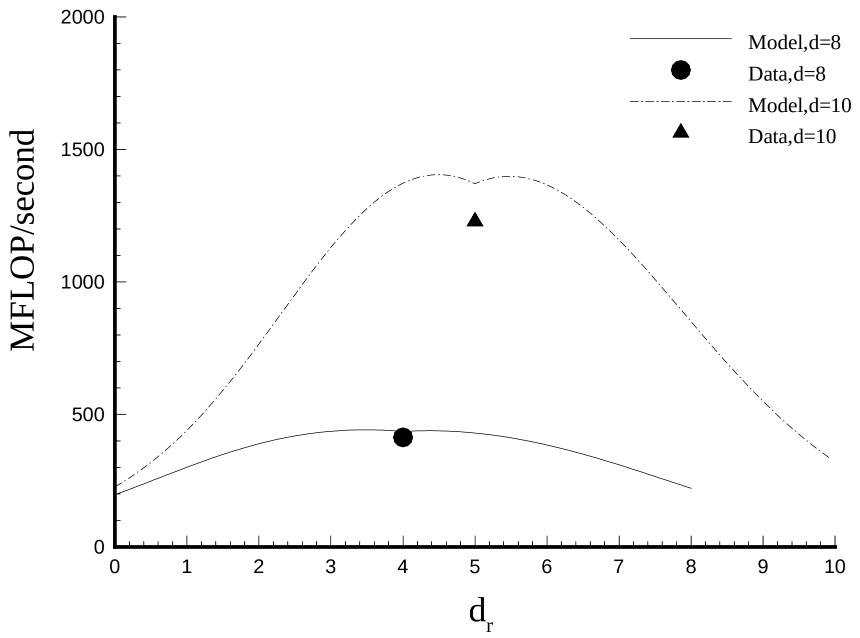

Fig. 11. Performance of the tridiagonalization code on the nCUBE 2.

As expected, the model indicates that a square torus is better than row or column methods for this problem. The cusp in the model is due to the different communication patterns that apply depending on the relative sizes of p_r and p_c. We note that for these problems, the model predicts a slight improvement in performance when $d_r = (d/2) - 1$, but the model neglects the additional copying required for non-square tori. We can use the model to predict the optimal tradeoff between p_r and p_c in general, but as in §5.1 and §5.2 no closed form expression exists.

As in §5.1 and §5.2, to investigate scaled speedup we ran the code on problems in which the local memory requirements remained constant. The results are presented in Table 5.

Table 5. Times for Householder tridiagonalization.

Matrix Size	Cube Dim	Mflop/s Observed	Mflop/s per Proc.	Scaled Speedup	Eff.
500	0	1.87	1.87	1.00	1.00
1,000	2	7.25	1.81	3.88	0.97
2,000	4	27.84	1.74	14.89	0.93
4,000	6	107.30	1.68	57.40	0.90
8,000	8	413.01	1.61	220.95	0.86
16,000	10	1596.33	1.56	853.97	0.83

On a single processor, the performance of the tridiagonalization code is about 5% less than that for LU or QR. This is a consequence of exploiting the symmetry of the matrix, and can be expected in other algorithms that work on symmetric matrices like Cholesky factorization. The short columns in the rightmost portion of the lower triangular matrix, and the short rows at the top result in many short vectors in the BLAS. Also, index calculations are more complex with a triangular matrix, adding some overhead to the calculation.

In addition, the performance of the tridiagonalization code scales less well than either LU or QR. The efficiency on 1,024 processors is about 83%, while for QR and LU it was in the upper 90's. This is a consequence of three factors. First, the repeated computations add about $n^2(7p_c + p_r)/2$ flops to the sequential algorithm. Second, exploiting symmetry requires a greater amount of communication. And third, the two binary exchanges effectively synchronize the processors, which reduces the potential for hiding communication with computation. The second factor will also influence other algorithms on symmetric matrices. Having said this, it is still true that the tridiagonalization code performs well, achieving greater than 75% of the peak BLAS performance on 1024 processors.

6. Conclusions. We have presented analytical and empirical evidence that for many dense linear algebra algorithms, the torus–wrap mapping is better than row or column mappings. The primary advantage of the torus–wrap is that it requires less communication, leading to better scalability, but there are a number of additional advantages including better load balancing, reduced processor idle time, and shorter message queues.

After factoring a matrix, one typically wishes to use it, for example, to solve linear systems or least–squares problems. This requires using the factored products to modify one or more vectors. If only a few vectors are involved, then the flop count is $\Theta(n^2)$, an order of magnitude less than the factorization. In this case, since the cost of the factorization dominates, a torus–wrap mapping for the factorization is likely to give the best overall performance. In addition, algorithms exist for doing a single triangular solve using the torus–wrap mapping that run in the asymptotically optimal time of $n^2/p + O(n)$ [5, 28].

If many vectors must be modified, then they can be combined to form a matrix which can be assigned to processors in a torus–wrap fashion. The same techniques that were employed in the factorization can now be used in the triangular solves, and good performance should result. For instance, the LU factorization code described in §5.1 has been used to invert a matrix by solving n linear equations on the nCUBE 2 at an overall computational rate of 1.96 Gflop/s.

Acknowledgements. We are indebted to Cleve Ashcraft and Andy Cleary for bringing the torus–wrap mapping to our attention, to Courtenay Vaughan for help

with the implementations and to Ernie Brickell for useful discussions. We also wish to acknowledge the extremely conscientious effort on the part of an anonymous referee whose suggestions significantly improved this paper.

REFERENCES

[1] C. C. Ashcraft, *The distributed solution of linear systems using the torus wrap data mapping*, Tech. Report ECA-TR-147, Boeing Computer Services, October 1990.

[2] ———, *A taxonomy of distributed dense LU factorization methods*, Tech. Report ECA-TR-161, Boeing Computer Services, March 1991.

[3] R. H. Bisseling and L. D. J. C. Loyens, *Towards peak parallel LINPACK performance on 400 transputers*, Supercomputer, 45 (1991), pp. 20–27.

[4] R. H. Bisseling and J. G. G. van de Vorst, *Parallel LU decomposition on a transputer network*, in Lecture Notes in Computer Science, Number 384, G. A. van Zee and J. G. G. van de Vorst, eds., Springer-Verlag, 1989, pp. 61–77.

[5] ———, *Parallel triangular system solving on a mesh network of transputers*, SIAM J. Sci. Stat. Comput., 12 (1991), pp. 787–799.

[6] R. P. Brent, *The LINPACK benchmark on the AP 1000: Preliminary report*, in Proc. 2nd CAP Workshop, November 1991.

[7] H. Y. Chang, S. Utku, M. Salama, and D. Rapp, *A parallel Householder tridiagonalization stratagem using scattered square decomposition*, Parallel Comput., 6 (1988), pp. 297–311.

[8] E. Chu and A. George, *QR factorization of a dense matrix on a hypercube multiprocessor*, SIAM J. Sci. Stat. Comput., 11 (1990), pp. 990–1028.

[9] J. J. Dongarra, *Performance of various computers using standard linear equations software*, Supercomputing Review, 4 (1991), pp. 45–54.

[10] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, *A set of level 3 basic linear algebra subprograms*, TOMS, 16 (1990), pp. 1–17.

[11] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia, PA, 1991.

[12] J. J. Dongarra, A. H. Sameh, and D. C. Sorensen, *Implementation of some concurrent algorithms for matrix factorization*, Parallel Comput., 3 (1986), pp. 25–34.

[13] J. J. Dongarra, D. Walker, and R. van de Geijn, *A look at scalable dense linear algebra libraries*, in Proc. Scalable High Performance Computing Conf., April 1992, pp. 372–379.

[14] G. C. Fox, *Square matrix decomposition – symmetric, local, scattered*, Tech. Report CalTech Publication Hm-97, California Institute of Technology, 1985.

[15] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving problems on concurrent processors: Volume 1*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[16] K. A. Gallivan, M. T. Heath, E. Ng, J. M. Ortega, B. W. Peyton, R. J. Plemmons, C. H. Romine, A. H. Sameh, and R. G. Voigt, *Parallel Algorithms for Matrix Computations*, SIAM, Philadelphia, PA, 1990.

[17] G. A. Geist and M. T. Heath, *Matrix factorization on a hypercube multiprocessor*, in Hypercube Processors, 1986, M. T. Heath, ed., SIAM, 1986, pp. 161–180.

[18] G. A. Geist and C. H. Romine, *LU factorization algorithms on distributed-memory multiprocessor architectures*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 639–649.

[19] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 1989. Second edition.

[20] J. L. Gustafson, G. R. Montry, and R. E. Benner, *Development of parallel methods for a 1024-processor hypercube*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 609–638.

[21] M. T. Heath and C. H. Romine, *Parallel solution of triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 558–588.

[22] B. Hendrickson, *Parallel QR factorization on a hypercube using the torus wrap mapping*, Tech. Report SAND91-0874, Sandia National Laboratories, October 1991.

[23] S. L. Johnsson, *Communication efficient basic linear algebra computations on hypercube architectures*, J. Par. Distr. Comput., 4 (1987), pp. 133–172.

[24] S. L. Johnsson and C. T. Ho, *Optimum broadcasting and personalized communication in hypercubes*, IEEE Trans. Computers, 38 (1989), pp. 1249–1268.

[25] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic linear algebra subprograms for fortran usage*, TOMS, 5 (1979), pp. 308–323.

[26] G. Li and T. F. Coleman, *A parallel triangular solver for a distributed-memory multiprocessor*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 485–502.

[27] W. Lichtenstein and S. L. Johnsson, *Block cyclic dense linear algebra*, SIAM J. Sci. Comput., 14 (1993), pp. 1257–1286.

[28] L. D. J. C. Loyens and R. H. Bisseling, *The formal construction of a parallel triangular system solver*, in Lecture Notes in Computer Science, Number 375, J. van de Snepscheut, ed., Springer-Verlag, 1989, pp. 325–334.

[29] D. P. O'Leary and G. W. Stewart, *Data-flow algorithms for parallel matrix computations*, Communications of the ACM, 28 (1985), pp. 840–853.

[30] ———, *Assignment and scheduling in parallel matrix factorization*, Lin. Alg. Appl., 77 (1986), pp. 275–299.

[31] J. M. Ortega and C. H. Romine, *The ijk forms of factorization methods ii. parallel systems*, Parallel Comput. 7 (1988), pp. 149–162.

[32] Y. Saad, *Communication complexity of the Gaussian elimination algorithm on multiprocessors*, Lin. Alg. Appl., 77 (1986), pp. 315–340.

[33] G. W. Stewart, *Communication and matrix computations on large message passing systems*, Parallel Comput., 16 (1990), pp. 27–40.

[34] R. van de Geijn, *Efficient global combine operations*, in Proc. 6th Distributed Memory Computing Conf., IEEE Computer Society Press, 1991, pp. 291–294.

[35] ———, *Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems*, Tech. Report Computer Science report TR–91–28, University of Texas, 1991.

[36] J. G. G. van de Vorst, *The formal development of a parallel program performing LU-decomposition*, Acta Inform., 26 (1988), pp. 1–17.

[37] E. L. Zapata, J. A. Lamas, F. F. Rivera, and O. G. Plata, *Modified Gram-Schmidt QR factorization on hypercube SIMD computers*, J. Par. Distr. Comput., 12 (1991), pp. 60–69.