Block-Cyclic Dense Linear Algebra

Block–Cyclic Dense Linear Algebra

Woody Lichtenstein
S. Lennart Johnsson

TR-04-92

January 1992
Revised August 1992

Parallel Computing Research Group

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

Block–Cyclic Dense Linear Algebra

Woody Lichtenstein and S. Lennart Johnsson[1]
Thinking Machines Corporation
245 First Street
Cambridge MA 02142-1214
woody@think.com, Johnsson@think.com

Abstract

Block–cyclic order elimination algorithms for LU and QR factorization and solve routines are described for distributed memory architectures with processing nodes configured as two–dimensional arrays of arbitrary shape. The cyclic order elimination together with a consecutive data allocation yields good load–balance for both the factorization and solution phases for the solution of dense systems of equations by LU and QR decomposition. Blocking may offer a substantial performance enhancement on architectures for which the level–2 or level–3 BLAS are ideal for operations local to a node. High rank updates local to a node may have a performance that is a factor of four or more higher than a rank–1 update.

We show that in many parallel implementations, the $O(N^2)$ work in the factorization may be of the same significance as the $O(N^3)$ work, even for large matrices. The $O(N^2)$ work is poorly load–balanced in two–dimensional nodal arrays, which we show are optimal with respect to communication for consecutive data allocation, block–cyclic order elimination, and a simple, but fairly general, communications model.

In our Connection Machine system CM–200 implementation, the peak performance for LU factorization is about 9.4 Gflops/s in 64-bit precision and 16 Gflops/s in 32-bit precision. Blocking offers an overall performance enhancement of about a factor of two. The broadcast and reduce operations fully utilize the bandwidth available in the Boolean cube network interconnecting the nodes along each axis of the two–dimensional nodal array embedded in the cube network.

1 Introduction

The main contributions of this paper are: 1) empirical evidence that a block–cyclic order elimination can be used effectively on distributed memory architectures to achieve load–balance as an alternative to block–cyclic data allocation, 2) a discussion of the issues that arise when the block–cyclic orderings of rows and columns are different, which is the typical case when the number of processing nodes is not a square, and 3) a proof that within a wide class of regular data layouts, two–dimensional nodal arrays with consecutive (block) data allocation and cyclic elimination order are optimal for elimination based dense linear algebra routines. This last result applies to communication systems in which the communication time is a function only of the number of elements entering or leaving a node. The effectiveness of the block–cyclic order elimination demonstrates the utility of

[1]Also affiliated with the Division of Applied Sciences, Harvard University, Cambridge, MA 02138

equivalencing block–distributed and block–cyclic distributed arrays in Fortran D [7] and Vienna Fortran [31].

The programs described in this article were written for an implementation of the Connection Machine Scientific Software Library [29], CMSSL, on the Connection Machine system CM–200 [27]. This system is a distributed memory computer with up to 2048 nodes. Each node has hardware support for floating–point addition and multiplication in 32–bit and 64–bit precision. Each node has up to 4 Mbytes of local memory, a single 32–bit wide data path between the floating–point processor and the local memory, and separate communication circuitry. Data paths internal to the floating–point unit are 64–bits wide. The processing units are interconnected as an 11–dimensional Boolean cube, with a pair of channels between adjacent nodes. Data may be exchanged on all 22 (11×2) channels of every node concurrently. This property is exploited for data copying (spread) and data summation (reduction) in the algorithms described below.

We consider LU factorization, QR factorization (with and without partial pivoting), and solution routines for both LU factorization (triangular system solvers) and QR factorization. Our triangular solver encompasses both the routine _TRSV in the level–2 BLAS [6] (Basic Linear Algebra Subroutines) and the routine _TRSM in the level–3 BLAS [4, 5]. It is easy to show that a cyclic data allocation with consecutive order elimination, or a consecutive allocation and a cyclic order elimination, yields a factor of three higher processor utilization on two–dimensional nodal arrays, than consecutive allocation and consecutive elimination order [17]. Coleman [22, 23] reports some results from an implementation of triangular system solvers using consecutive order elimination and cyclic data allocation on multiprocessors with up to 128 nodes. Van de Geijn [30] uses consecutive order elimination and cyclic data allocation for an implementation of LU factorization and triangular system solvers. Since the Connection Machine system compilers by default use consecutive data allocation, we use a cyclic order elimination. We use a similar implementation strategy for routines for reduction of a symmetric or Hermitian matrix to real tridiagonal form (EISPACK _TRED and _HTRID [26], LAPACK _SYTRD and _HETRD [1]). Details concerning these routines will appear elsewhere.

We use blocking of row and column operations to increase the efficiency of operations in each node. The level–2 BLAS is used in each node to achieve maximum performance. The difference in peak performance between a rank–1 update and a higher rank update is about a factor of four on a Connection Machine system CM–200 node. The difference in peak performance is mostly determined by the difference in need for memory bandwidth between a rank–1 and a high rank update. In our CMSSL implementation of the BLAS local to each Connection Machine system CM–200 node [20], LBLAS, each node achieves a peak performance of about 9.3 Mflops/s in 64–bit precision on matrix multiplication (high rank updates). Our implementation of LU factorization of matrices distributed over all nodes achieves a peak performance, including communication, of 4.6 Mflops/s per node in 64–bit precision. As a comparison, our CMSSL implementation of dense matrix multiplication with operands distributed across all nodes achieves a peak performance of 4.8 Gflops/s in 64–bit precision [24].

This article provides sufficient insight into the details of the algorithms to account for the difference in performance between local matrix multiplication and global factorization and solve routines for dense matrices. We also describe how the performance scales with problem and machine size. In Section 2 we review the merits of higher level local BLAS for a class of common processor architectures. Section 3 discusses the performance of the level–1 and level–2 LBLAS used in CMSSL. Section 4 describes the layouts of data arrays created by the Connection Machine Run–Time System. Section 5 explains how we use block–cyclic ordering of the elimination steps to keep the work load–balanced across all nodes. Section 6 discusses the performance characteristics of the different parts of the algorithms. Finally, we show in Section 7 that a two–dimensional consecutive data layout is optimal for elimination algorithms using a cyclic order elimination and communication systems where the times for data copying and reduction are determined by the number of data items leaving or entering a node.

2 Blocking for improved performance of local BLAS

Memory bandwidth is the most critical resource in high performance architectures. Therefore, proper attention must be given to the primitives used in constructing linear algebra libraries. Nearly all floating–point computation in linear algebra occurs as multiply–add pairs. This fact is evident in the BLAS used to perform many operations in the solution of linear systems of equations, eigenanalysis, optimization, and the solution of partial differential equations. The first BLAS were vector routines (like DSCAL and DDOT) [21]. But, these routines require large memory bandwidth for peak floating–point performance, and algorithm designers turned to higher level BLAS, such as level-2 BLAS for matrix-vector operations, and level–3 BLAS for matrix-matrix operations. Next, we give a few examples to illustrate this fact.

Today, most high performance floating–point processors have the ability to perform concurrently one multiplication and one addition. The data for these operations is nearly always read from registers, and the results are written to registers. The peak performance can only be achieved when multiplication and addition can be performed concurrently, and the data they require can flow in and out of registers fast enough. The memory bandwidth required to match the computational bandwidth depends on the computation being performed.

For example, the DSCAL operation which multiplies a vector by a scalar with all operands in 64–bit precision, reads one scalar into a register, and then, for each component of the result, reads one element of the argument vector, multiplies it by the constant, and writes one result to memory. Thus, each multiplication requires 8 bytes to be loaded into a register and 8 bytes to be stored from a register, or 16 bytes of memory bandwidth per floating–point operation. As a contrast, the DDOT routine, which computes the inner product of 2 vectors in 64–bit precision, reads two 8 byte quantities for each multiply-add it performs, then stores one 8 byte result at the end. Thus, 16 bytes of memory bandwidth is required per 2 floating–point operations, or 8 bytes per floating–point operation. Similarly, it is easy to show that the rank–1 update routine DGER requires a memory bandwidth of 8

Operation	Mem. bandw. Bytes/flop
DSCAL	16
DAXPY	12
DDOT	8
DGER	8
DGEMV	4
DGEMM	8/b

Table 1: Memory bandwidth requirement for full utilization of a floating–point unit with one adder and one multiplier.

bytes per floating–point operation.

Matrix–vector multiplication, performed by the routine DGEMV, can be organized such that the vector is read into registers, and the matrix–vector multiplication computed through the accumulation of scaled vectors as $y \leftarrow \alpha x + y$. With α and y allocated to registers, and x representing a column of the matrix read from memory, 8 bytes of memory bandwidth is required for every pair of floating–point operations in 64–bit precision.

A level–3 BLAS routine such as DGEMM, allows for a further reduction in memory bandwidth requirement. It performs the operation $C \leftarrow A \times B$, which may be performed as a sequence of operations on b by b sub-blocks. If the blocks fit into the registers, then $2b^3$ floating–point operations may be computed using $3b^2$ input elements (b^2 elements per operand), producing b^2 results. If all contributions to a block of C are accumulated in registers, then it suffices to load $2b^2$ inputs for each set of $2b^3$ floating–point operations. All stores are delayed until all computations for a $b \times b$ block of C are completed. Therefore, $16b^2/2b^3$ bytes of memory bandwidth are required per floating–point operation in 64–bit precision, or $8/b$ bytes/flop. A high rank update of a matrix is equivalent to matrix multiplication.

Table 1 summarizes the memory bandwidth requirements for a subset of the BLAS. The significance of the difference in memory bandwidth requirement of the different routines depend upon the available memory bandwidth. For example, a computer with three 8 bytes wide data paths between each processor and the memory, i.e., a 12 bytes/flop computer such as the Cray-YMP, can perform DAXPY operations at peak rates. A 2 bytes/flop computer, such as an Intel i860 [10] and a Connection Machine system CM–200 node, may not have enough registers to achieve peak rates even for level–3 BLAS local to each node, such as for instance for the DGEMM routine. As a rule of thumb, the less memory bandwidth is available, the more blocking is desirable, but more blocking is only useful if there is a sufficient number of registers.

This simplified performance picture is in reality often complicated by pipeline delays and looping overheads. For short vectors and small register sets minimizing these quantities may be as important for performance as minimizing the demand for memory bandwidth.

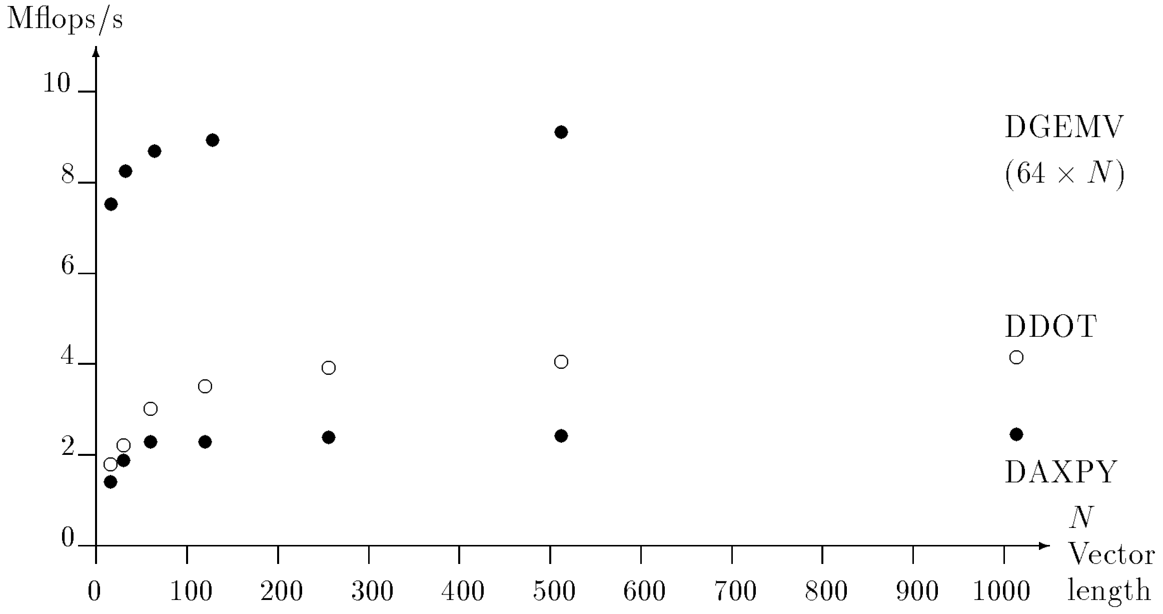

Figure 1: The execution rates in Mflops/s of the DAXPY, DDOT, and DGEMV LBLAS on *each* Connection Machine system CM–200 node.

3 Local BLAS on the Connection Machine system CM–200

In the following we refer to the BLAS local to each node as LBLAS to distinguish it from BLAS for data arrays distributed over several nodes, DBLAS. Each node of the Connection Machine system CM–200 is a 2 bytes/flop computer, based on the memory bandwidth. There is a single 32–bit wide data path between each floating–point processor and its local memory. The data paths internal to a processor are 8 bytes wide (i.e., internally each floating–point processor is a 4 bytes/flop computer). Each floating–point processor has 32 floating–point registers, and operates at 10 MHz. There is a one cycle delay for loading data from memory, and a three cycle arithmetic pipeline delay. Each vector operation incurs an overhead of at least 6 cycles. Moreover, the memory is organized into pages, and a page fault incurs a delay of one cycle. Stores are relatively more expensive and require close to two cycles. The floating–point processor can achieve close to peak performance in 32–bit precision for level–2 LBLAS with the operands local to a node, and at least half of the peak performance in 64–bit precision. The floating–point processor does not have enough registers to achieve its full peak rate on level–3 LBLAS. Table 2 gives the actual performance for the SAXPY, DAXPY, SDOT, and DDOT routines as function of the vector length for *each* Connection Machine system CM–200 node. Tables 3 and 4 give the performance for SGEMV and DGEMV as a function of matrix size.

Figure 1 shows the performance for the DAXPY and DDOT routines on each Connection Machine system CM–200 node as a function of the vector length. The figure also shows the performance of the routine DGEMV as a function of the number of columns for 64 matrix rows. The peak measured performance for the _AXPY routine is about 60% of the peak

V-length	SAXPY	DAXPY	SDOT	DDOT
1	0.21	0.16	0.21	0.16
2	0.41	0.29	0.35	0.35
3	0.58	0.42	0.52	0.51
4	0.80	0.54	0.68	0.66
5	0.97	0.65	0.83	0.79
8	1.40	0.95	1.27	1.15
10	1.65	1.13	1.53	1.36
15	2.15	1.41	2.13	1.79
20	2.55	1.61	2.65	2.13
30	3.11	1.87	3.03	2.21
40	3.07	1.89	3.71	2.70
50	3.37	2.02	4.23	2.96
60	3.53	2.09	4.41	3.02
70	3.56	2.11	4.75	3.19
80	3.71	2.18	5.06	3.33
100	3.77	2.22	5.32	3.45
120	3.93	2.28	5.51	3.53
140	4.00	2.31	5.86	3.68
256	4.20	2.39	6.52	3.92
512	4.32	2.43	6.97	4.07
1024	4.42	2.46	7.24	4.15
2048	4.46	2.48	7.38	4.20
4096	4.49	2.49	7.45	4.22
8192	4.50	2.49	7.49	4.24

Table 2: Level–1 LBLAS execution rates in Mflops/s on *each* Connection Machine system CM-200 floating–point processor.

SGEMV											
Number of	Number of columns										
Rows	2	3	4	5	8	16	32	64	128	512	2048
2	0.62	0.98	1.23	1.46	2.46	4.03	5.93	7.38	8.62	9.90	10.30
3	0.99	1.39	1.74	2.05	3.38	6.03	8.21	9.80	10.8	11.9	12.20
4	1.35	1.88	2.33	2.73	4.40	6.72	8.87	10.70	12.00	13.10	13.50
5	1.62	2.24	2.77	3.24	5.11	8.43	10.6	12.20	13.20	14.10	14.40
8	2.33	3.19	3.91	4.52	6.80	9.47	11.90	13.70	14.70	15.70	15.90
16	3.68	4.91	5.90	6.71	9.27	12.10	14.40	15.80	16.70	17.40	17.50
32	4.29	5.96	6.39	7.56	10.20	12.90	14.90	16.10	16.80	17.40	17.50
64	5.13	6.85	7.61	8.71	11.50	14.00	15.80	16.80	17.40	17.90	—
128	5.51	7.09	8.01	8.94	11.90	14.40	16.00	17.00	17.50	17.90	—
512	5.80	7.30	8.32	9.15	12.10	14.40	15.90	16.80	17.30	—	—
2048	5.92	7.40	8.45	9.25	12.10	14.30	15.70	—	—	—	—

Table 3: Execution rate in Mflops/s for matrix-vector multiplication on *each* Connection Machine system CM–200 floating–point processor.

DGEMV											
Number of	Number of columns										
Rows	2	3	4	5	8	16	32	64	128	512	2048
2	0.64	0.89	1.10	1.29	2.03	3.04	3.93	4.67	5.17	5.62	5.74
3	0.91	1.25	1.53	1.78	2.71	4.20	5.15	5.80	6.21	6.56	6.66
4	1.13	1.54	1.88	2.17	3.22	4.44	5.51	6.26	6.72	7.12	7.23
5	1.33	1.80	2.18	2.51	3.64	5.23	6.21	6.84	7.22	7.54	7.63
8	1.80	2.39	2.87	3.26	4.47	5.82	6.85	7.51	7.90	8.11	8.22
16	2.54	3.29	3.79	4.25	5.58	6.89	7.80	8.35	8.65	8.90	8.97
32	2.79	3.60	4.11	4.69	5.92	7.14	7.95	8.44	8.70	8.92	—
64	3.19	4.10	4.60	5.12	6.37	7.52	8.26	8.69	8.93	9.11	—
128	3.33	4.20	4.74	5.22	6.49	7.60	8.30	8.70	8.92	—	—
512	3.45	4.33	4.86	5.30	6.53	7.57	8.22	8.60	—	—	—
2048	3.49	4.89	4.91	5.34	6.57	7.60	—	—	—	—	—

Table 4: Execution rate in Mflops/s for matrix-vector multiplication on *each* Connection Machine system CM–200 floating–point processor.

performance of the _DOT routine. In 32–bit precision the performance is comparable up to a vector length of about 30, while for 64–bit precision the difference is measurable even for short vectors. Note that the _AXPY routine requires twice the memory bandwidth of the _DOT routine. The performance of the _GEMV routine is about twice that of the _DOT routine. This behavior is expected, since the memory bandwidth requirement of the _GEMV routine is about half of that of the _DOT routine.

Our implementation of the rank–1 update makes use of _AXPY operations, while the higher rank updates are based on matrix–vector or vector–matrix multiplication, depending upon the shape of the operands [20].

Given the performance characteristics of the LBLAS on the Connection Machine system CM–200, it is desirable to base linear algebra algorithms on the level–2 LBLAS. The potential performance gain from blocking the operations on b rows and columns is illustrated in Table 5. The Table shows the speedup of rank–b updates relative to a rank–1 update of a 32×32 matrix and a 512×512 matrix local to a Connection Machine system CM–200 node.

Blocking yields a larger relative performance gain for small matrices than for big matrices. For large submatrices per node a performance gain by a factor in excess of 3.5 is possible through the use of high rank updates. About 90% of this gain is achieved for a blocking factor of 16. For relatively small submatrices per node a performance gain in excess of a factor of 4.5 is possible through the use of high rank updates. About 85% of this gain is achieved by the use of a blocking factor of 16.

Rank	Matrix shape	
	32×32	512×512
1	1.00	1.00
2	1.49	1.42
4	2.20	2.00
8	3.17	2.69
16	3.82	3.12
32	4.25	3.38
64	4.51	3.54

Table 5: Speedup of rank–b updates over rank–1 updates for *each* Connection Machine system CM–200 node.

4 Standard data layouts

The routines described in this article are designed to operate in–place on arrays passed to the routines from high level language programs. The data motion requirements and the performance depend strongly upon the data allocation of the operands. The default allocation of data arrays to nodes determined by the Connection Machine Run–Time System is based entirely on the shape of the data arrays. Each array is by default distributed evenly over all nodes, i.e., the Connection Machine systems support a global address space. In the default array allocation mode, the nodes are configured for each array, such that the number of axes in the data array and in the nodal array is the same. The ordering of the axes is also the same. When there are more matrix elements than nodes, consecutive elements along each data array axis (a block) are assigned to a node. The ratios of the lengths of the axes of the nodal array are approximately equal to the ratios between the lengths of the axes of the data array [28]. In such a configuration the lengths of the local segments of all axes are approximately the same, and the communication needs minimized when references along the different axes are equally frequent. The default array layout is known as a *canonical layout.* In Section 7 we show that the canonical layout is optimum for LU and QR factorization for a simple, but realistic communications model. In [24] we show that the canonical layout is also optimal for matrix multiplication.

The canonical layout can be altered through compiler directives. An axis can be forced to be local to a node by the directive SERIAL, if there is sufficient local memory. The length of the local segment of an axis can also be changed by assigning *weights* to the axes. High weights are used for axes with frequent communication, and low weights for axes with infrequent communication. A relatively high weight for an axis increases the length of the local segment of that axis, at the expense of the lengths of the segments of the other axes. The total size of the subarray is independent of the assignment of weights. Only the shape of the subarray changes. The shape of the nodal array is important, since it affects the performance of global operations such as data copying, data summation, and pivot selection. The nodal array shape is also important for the performance of the LBLAS, since it affects the shape of the subarrays assigned to each node, and hence vector lengths and the relative importance of loop overhead.

In many computations more than one array is involved, and the relative layouts of the arrays may be important. For instance, in solving a liner system of equations there are two arrays involved when the computed solutions overwrite the right–hand sides. The required communication in the triangular systems solver depends in a significant way on how the triangular factors and the right–hand sides are allocated to the nodes. With the original matrix factored in–place, the triangular factors are stored in a two–dimensional data array, for which the default nodal array shape is a two–dimensional array. For a single right–hand side, the default nodal array shape is a one–dimensional array. Even if there are many right–hand sides, there is no guarantee that the shapes of the nodal arrays are the same for the data array to be factored and the data array of right–hand sides. The ALIGN compiler directive may be used to assure that different data arrays are assigned to nodes using the same nodal array shape for the allocation.

The consecutive allocation scheme [17] selects elements to be assigned to the same node. Compiler directives, such as axis weights, SERIAL and ALIGN, address the issue of choosing the nodal array shape. Another data layout issue is the assignment of data sets to nodes, where a set is made up of consecutive elements along each data array axis. The network topology and the data reference pattern are two important characteristics in this assignment. The nodes of the Connection Machine system CM–200 are interconnected as a Boolean cube with up to 11 dimensions. A Boolean cube network of n dimensions has 2^n nodes.

The nodes of a Boolean cube can be given addresses such that adjacent nodes differ in precisely one bit in their binary encoding. Assigning subarrays to nodes using the standard binary encoding of the subarray index along an axis, does not preserve adjacency along an axis. For instance, 3 and 4 differ in all three bits in the encoding of the addresses of 8 nodes, and are at a distance of three apart. In general, $2^{n-1} - 1$ and 2^{n-1} differ in n bits in their binary encoding, and are at a distance of n. The number of bits in which two indices differ translates directly into distance in a Boolean cube architecture.

Binary–reflected Gray codes [25] generate embeddings of arrays into Boolean cube networks that preserve adjacency [17]. Gray codes have the property that the encoding of successive integers differ in precisely one bit. In a Boolean cube network successive indices are assigned to adjacent nodes. The binary–reflected Gray code is efficient, both in preserving adjacency and in node utilization, when the length of the axes of the data array is a power of two [11]. For arbitrary data array axes' lengths, the Gray code may be combined with other techniques to generate efficient embeddings [3, 12].

The binary–reflected Gray code embedding is the default embedding on the Connection Machine system CM–200, and is enforced by the compiler directive NEWS for each axis. The standard binary encoding of each axis is obtained through the compiler directive SEND. The performance of global operations, such as pivot selection and broadcast required in the factorization and solve routines, is insensitive to the data distribution along each axis. Our factorization and solver routines also require permutations for rectangular nodal arrays. These permutations may exhibit some sensitivity to whether arrays are allocated in NEWS or SEND order.

5 Cyclic order factorization and triangular system solution

We consider the solution of a system of equations $AX = Y$ through factorization and forward and backward substitution. We consider both single and multiple right–hand sides. We present algorithms for load–balanced factorization and triangular systems solution on data arrays allocated to nodes by a consecutive data allocation scheme. The matrix is factored in–place, and the solutions overwrite the right–hand sides.

5.1 Factorization

5.1.1 Balanced work load

For an N by N matrix assigned to a $p \times q$ nodal array, each node is assigned a submatrix of shape $\frac{N}{p}$ by $\frac{N}{q}$. With the consecutive allocation scheme, the submatrix on node (i, j), $1 \leq i \leq p$, $1 \leq j \leq q$ consists of elements (a, b) with $(i-1) * \frac{N}{p} < a \leq i * \frac{N}{p}$ and $(j-1) * \frac{N}{q} < b \leq j * \frac{N}{q}$.

An LU factorization algorithm steps through the rows of a matrix, subtracting a multiple of a row, the pivot row, from all rows not yet selected as pivot rows. If the matrix is allocated to the nodes with a consecutive allocation scheme, and the pivot rows are selected in order, then after $\frac{N}{p}$ rows have been chosen, the first row of nodes will no longer have any work to do. One method for avoiding this imbalance is to redistribute the rows and columns of the matrix periodically, as the work load becomes unbalanced. This approach leads to substantial communications overhead. The characteristics of QR decomposition is similar, though instead of subtracting the pivot row from rows not yet selected as pivots, a normalized linear combination of those rows is subtracted from each of them.

A more elegant technique is to use a cyclic data allocation scheme [8, 17, 22, 23, 30]. In a cyclic data allocation scheme the first row of the matrix is placed on the first row of nodes, the second row of the matrix on the second row of nodes, and so on until one matrix row has been assigned to each of the p rows of nodes. Then, the $(p+1)^{st}$ matrix row is assigned to the first row of nodes, etc. With this method, matrix rows are consumed evenly from the different rows of nodes during elimination, so that no row of nodes ever has more than one more active matrix row than any other row of nodes. Columns are treated similarly.

The space (rows and columns) and time (pivot selection) dimensions are interchangeable [13, 14, 15, 16]. Hence, instead of distributing rows and columns cyclically over the two–dimensional nodal array, pivot rows and columns can be selected cyclically to achieve the desired load–balance. We have chosen this approach for our implementation. To allow for the use of level–2 LBLAS, blocking of rows and columns on each node is used. In LU factorization a blocking of the operations on b rows and columns means that b rows are eliminated at a time from all the other rows.

In summary, we factor a matrix in–place using a *block–cyclic elimination order* for both rows and columns. For block size b we first eliminate rows $1, \ldots, b$, then rows $(p+1), \ldots, (p+b)$, etc. We are eliminating b matrix rows at a time from each row of nodes. A block–cyclic elimination order was recommended in [15] for load–balanced solution of banded systems.

Below we give an example of a block–cyclic ordered elimination for a square array of nodes, and a block size of 2. Pivot rows and columns are indicated by numbers, eliminated elements by periods, and all other elements are shown as an asterisk.

Example I: $N = 16$, $p = q = 4$, $b = 2$

$$\left[\begin{array}{cccc@{\quad}cccc@{\quad}cccc@{\quad}cccc}
\multicolumn{16}{c}{\text{Step} \quad 1.} \\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\[4pt]
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\[4pt]
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\[4pt]
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
1&1&*&*&*&*&*&*&*&*&*&*&*&*&*&*
\end{array}\right]
\left[\begin{array}{cccc@{\quad}cccc@{\quad}cccc@{\quad}cccc}
\multicolumn{16}{c}{\text{Step} \quad 2.} \\
&&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
.&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&2&2&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&2&2&*&*&*&*&*&*&*&*&*&*\\[4pt]
.&.&2&2&2&2&2&2&2&2&2&2&2&2&2&2\\
.&.&2&2&2&2&2&2&2&2&2&2&2&2&2&2\\
.&.&*&*&2&2&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&2&2&*&*&*&*&*&*&*&*&*&*\\[4pt]
.&.&*&*&2&2&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&2&2&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&2&2&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&2&2&*&*&*&*&*&*&*&*&*&*\\[4pt]
.&.&*&*&2&2&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&2&2&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&2&2&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&2&2&*&*&*&*&*&*&*&*&*&*
\end{array}\right]$$

$$\left[\begin{array}{cccc@{\quad}cccc@{\quad}cccc@{\quad}cccc}
\multicolumn{16}{c}{\text{Step} \quad 3.} \\
&&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
.&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&.&.&*&*&3&3&*&*&*&*&*&*\\
.&.&*&*&.&.&*&*&3&3&*&*&*&*&*&*\\[4pt]
.&.&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&.&*&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&.&.&*&*&3&3&*&*&*&*&*&*\\
.&.&*&*&.&.&*&*&3&3&*&*&*&*&*&*\\[4pt]
.&.&3&3&.&.&3&3&3&3&3&3&3&3&3&3\\
.&.&3&3&.&.&3&3&3&3&3&3&3&3&3&3\\
.&.&*&*&.&.&*&*&3&3&*&*&*&*&*&*\\
.&.&*&*&.&.&*&*&3&3&*&*&*&*&*&*\\[4pt]
.&.&*&*&.&.&*&*&3&3&*&*&*&*&*&*\\
.&.&*&*&.&.&*&*&3&3&*&*&*&*&*&*\\
.&.&*&*&.&.&*&*&3&3&*&*&*&*&*&*\\
.&.&*&*&.&.&*&*&3&3&*&*&*&*&*&*
\end{array}\right]
\left[\begin{array}{cccc@{\quad}cccc@{\quad}cccc@{\quad}cccc}
\multicolumn{16}{c}{\text{Step} \quad 4.} \\
&&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
.&*&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&.&.&*&*&.&.&*&*&4&4&*&*\\
.&.&*&*&.&.&*&*&.&.&*&*&4&4&*&*\\[4pt]
.&.&*&*&*&*&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&.&*&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&.&.&*&*&.&.&*&*&4&4&*&*\\
.&.&*&*&.&.&*&*&.&.&*&*&4&4&*&*\\[4pt]
.&.&*&*&.&.&*&*&*&*&*&*&*&*&*&*\\
.&.&*&*&.&.&*&*&.&*&*&*&*&*&*&*\\
.&.&*&*&.&.&*&*&.&.&*&*&4&4&*&*\\
.&.&*&*&.&.&*&*&.&.&*&*&4&4&*&*\\[4pt]
.&.&4&4&.&.&4&4&.&.&4&4&4&4&4&4\\
.&.&4&4&.&.&4&4&.&.&4&4&4&4&4&4\\
.&.&*&*&.&.&*&*&.&.&*&*&4&4&*&*\\
.&.&*&*&.&.&*&*&.&.&*&*&4&4&*&*
\end{array}\right]$$

Step 5.

$$\left[\begin{array}{cccccccccccccccc}
* & * & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & * & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & . & 5 & 5 & . & . & 5 & 5 & . & . & 5 & 5 & . & . & 5 & 5 \\
. & . & 5 & 5 & . & . & 5 & 5 & . & . & 5 & 5 & . & . & 5 & 5 \\
. & . & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & * & * & * & * & * & * & * & * & * & * & * \\
. & . & 5 & 5 & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & 5 & 5 & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & * & * & . & . & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & * & * & * & * & * & * & * \\
. & . & 5 & 5 & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & 5 & 5 & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & * & * & . & . & * & * & . & . & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & . & * & * & . & * & * & * \\
. & . & 5 & 5 & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & 5 & 5 & . & . & * & * & . & . & * & * & . & . & * & *
\end{array}\right]$$

Step 6.

$$\left[\begin{array}{cccccccccccccccc}
* & * & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & * & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & . & * & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & * & * & * & * & * & * & * & * & * & * & * \\
. & . & . & . & . & . & 6 & 6 & . & . & 6 & 6 & . & . & 6 & 6 \\
. & . & . & . & . & . & 6 & 6 & . & . & 6 & 6 & . & . & 6 & 6 \\
. & . & * & * & . & . & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & * & * & * & * & * & * & * \\
. & . & . & . & . & . & 6 & 6 & . & . & * & * & . & . & * & * \\
. & . & . & . & . & . & 6 & 6 & . & . & * & * & . & . & * & * \\
. & . & * & * & . & . & * & * & . & . & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & . & * & * & . & * & * & * \\
. & . & . & . & . & . & 6 & 6 & . & . & * & * & . & . & * & * \\
. & . & . & . & . & . & 6 & 6 & . & . & * & * & . & . & * & *
\end{array}\right]$$

Step 7.

$$\left[\begin{array}{cccccccccccccccc}
* & * & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & * & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & . & * & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & * & * & * & * & * & * & * & * & * & * & * \\
. & . & . & . & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & . & . & . & . & . & * & . & . & * & * & . & . & * & * \\
. & . & * & * & . & . & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & * & * & * & * & * & * & * \\
. & . & . & . & . & . & . & . & . & . & 7 & 7 & . & . & 7 & 7 \\
. & . & . & . & . & . & . & . & . & . & 7 & 7 & . & . & 7 & 7 \\
. & . & * & * & . & . & * & * & . & . & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & . & * & * & . & * & * & * \\
. & . & . & . & . & . & . & . & . & . & 7 & 7 & . & . & * & * \\
. & . & . & . & . & . & . & . & . & . & 7 & 7 & . & . & * & *
\end{array}\right]$$

Step 8.

$$\left[\begin{array}{cccccccccccccccc}
* & * & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & * & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & . & * & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & * & * & * & * & * & * & * & * & * & * & * \\
. & . & . & . & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & . & . & . & . & . & * & . & . & * & * & . & . & * & * \\
. & . & * & * & . & . & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & * & * & * & * & * & * & * \\
. & . & . & . & . & . & . & . & . & . & * & * & . & . & * & * \\
. & . & . & . & . & . & . & . & . & . & . & * & . & . & * & * \\
. & . & * & * & . & . & * & * & . & . & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & . & * & * & . & * & * & * \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 8 & 8 \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 8 & 8
\end{array}\right]$$

As can be seen from Example I, the result of the factorization is not two block triangular matrices, but *block–cyclic triangles.* A block–cyclic triangle can be permuted to a block triangular matrix, as discussed in Section 5.2. However, it is not necessary to carry out this permutation for the solution of the block–cyclic triangular system of equations. Indeed, it is desirable to use the block–cyclic triangle for the forward and backsubstitutions, since the substitution process is load–balanced for the block–cyclic triangles. Using block triangular matrices stored in a square data array (A) allocated to nodes with a consecutive data allocation scheme would result in poor load–balance. Before discussing block–cyclic triangular solvers, and the relationship between pivoting strategies and the block–cyclic elimination order, we consider rectangular nodal arrays.

5.1.2 Rectangular arrays of processing nodes

Rectangular nodal arrays result in different row and column orderings, since the length of the cycle for the cyclic elimination order is different for rows and columns. Let $\hat{P}_1$ give the relationship between consecutive order and block–cyclic order elimination for rows, and $\hat{P}_2$ give the relationship between consecutive order and block–cyclic order elimination for columns. Then, $\hat{P}_1(i) = j$ means that the i^{th} row to be eliminated is row j. Specificly, for a matrix allocated to the p by q nodal array with the consecutive (block) allocation scheme, as described in Section 5.1, the block–cyclic ordered elimination with a block size

$b = 1$ yields

$$\begin{aligned}\hat{P}_1 &= 1, \frac{N}{p}+1, 2\frac{N}{p}+1, \ldots \\ \hat{P}_2 &= 1, \frac{N}{q}+1, 2\frac{N}{q}+1, \ldots\end{aligned}$$

Thus, $\hat{P}_1 \neq \hat{P}_2$ if $p \neq q$. Clearly, if the number of nodes in the entire machine is not a square (as is the case for a 2048 node Connection Machine system CM–200, and the Intel Delta [30]), and all nodes are used, then $p \neq q$ and $\hat{P}_1 \neq \hat{P}_2$. How to choose p and q for optimum performance under the constraint $pq = \text{const}$ is discussed in Section 7.

Note that even though $\hat{P}_1 \neq \hat{P}_2$ naturally occurs when $p \neq q$, choosing a different blocking factor for rows and columns yields the same result.

5.2 Solving block–cyclic triangular systems

In an algorithm eliminating rows and columns consecutively, transformations are applied to an original matrix A, until it has been reduced to an upper triangular matrix T. In block cyclic–ordered elimination, A is reduced to a block–cyclic triangular matrix $V = P_1 T P_2^{-1}$, where T is upper triangular. Block–cyclic triangles are just as easy to invert as standard block triangles, but they are load–balanced across the nodes of a distributed memory machine. The block–cyclic triangle V for the example above (step 8, Example I), and the corresponding permutation matrix P_1, are shown below. Example II shows a block–cyclic triangle for a rectangular nodal array ($P_1 \neq P_2$).

Example I: $N = 16$, $p = q = 4$, $b = 2$

$$V = P_1 T P_2^{-1} = \left[\begin{array}{cccc@{\qquad}cccc@{\qquad}cccc@{\qquad}cccc}
* & * & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & * & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & . & * & . & . & * & * & . & . & * & * & . & . & * & * \\[1em]
. & . & * & * & * & * & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & * & * & * & * & * & * & * & * & * & * & * \\
. & . & . & . & . & . & * & * & . & . & * & * & . & . & * & * \\
. & . & . & . & . & . & . & * & . & . & * & * & . & . & * & * \\[1em]
. & . & * & * & . & . & * & * & * & * & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & * & * & * & * & * & * & * \\
. & . & . & . & . & . & . & . & . & . & * & * & . & . & * & * \\
. & . & . & . & . & . & . & . & . & . & . & * & . & . & * & * \\[1em]
. & . & * & * & . & . & * & * & . & . & * & * & * & * & * & * \\
. & . & * & * & . & . & * & * & . & . & * & * & . & * & * & * \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & * & * \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & *
\end{array}\right]$$

where $$P_1 = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}$$

The permutation matrix P_2 is constructed analogously.

Example II: $N = 16$, $p = 2$, $q = 4$, $b = 2$

$$V = P_1 T P_2^{-1} \quad = \left[\begin{array}{cccc@{\quad}cccc@{\quad}cccc@{\quad}cccc}
\ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast \\
. & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast \\
. & . & \ast & \ast & . & . & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast \\
. & . & \ast & \ast & . & . & \ast & \ast & . & \ast & \ast & \ast & \ast & \ast & \ast & \ast \\
. & . & \ast & \ast & . & . & \ast & \ast & . & . & \ast & \ast & . & . & \ast & \ast \\
. & . & . & \ast & . & . & \ast & \ast & . & . & \ast & \ast & . & . & \ast & \ast \\
. & . & . & . & . & . & . & . & . & . & \ast & \ast & . & . & \ast & \ast \\
. & . & . & . & . & . & . & . & . & . & . & \ast & . & . & \ast & \ast \\
 & & & & & & & & & & & & & & & \\
. & . & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast \\
. & . & \ast & \ast & . & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast & \ast \\
. & . & \ast & \ast & . & . & \ast & \ast & . & . & \ast & \ast & \ast & \ast & \ast & \ast \\
. & . & \ast & \ast & . & . & \ast & \ast & . & . & \ast & \ast & . & \ast & \ast & \ast \\
. & . & . & . & . & . & \ast & \ast & . & . & \ast & \ast & . & . & \ast & \ast \\
. & . & . & . & . & . & . & \ast & . & . & \ast & \ast & . & . & \ast & \ast \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & \ast & \ast \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & \ast
\end{array}\right]$$

In our block–cyclic triangular solver, the solution matrix X overwrites the right–hand sides Y. The solver requires that the set of right–hand sides Y and the matrix A are *aligned.* The alignment of Y and A ensures that the shape of the nodal array is the same for A and Y. For a single right–hand side, the alignment implies that the components of Y are allocated to the first column of the nodal array using the consecutive allocation scheme. With several right–hand sides, the alignment implies that the consecutive allocation scheme is used to allocate right–hand sides and columns of the matrix A to the same number of node columns, with each right–hand side being allocated to a single node column. The alignment of A and Y can be accomplished without data motion by using the compiler directive ALIGN. Alignment at run–time in general will require data motion. It can be performed by the Connection Machine router. Our library routine validates that the arrays are aligned, and if the arrays are not prealigned, performs the alignment through a call to the router.

Our column–oriented algorithm for solving block–cyclic upper triangular systems of the form

$$P_1 T P_2^{-1}\ X = Y.$$

starts with the last block column of $P_1 T P_2^{-1}$ and progresses towards its first block column, in a way similar to a standard column oriented upper triangular system solver. A multiple of each block column of $P_1 T P_2^{-1}$ is subtracted from Y in each backward elimination step. The multiple of block column k, which is subtracted from Y at the k^{th} stage of this process is the k^{th} block component (row) of the solution vector X. If the k^{th} block column consists of columns $\hat{P}_2(j), \hat{P}_2(j+1), \ldots, \hat{P}_2(j+b-1)$, then the last b elements of each column in this block column are in rows $\hat{P}_1(j), \hat{P}_1(j+1), \ldots, \hat{P}_1(j+b-1)$. If X overwrites Y,

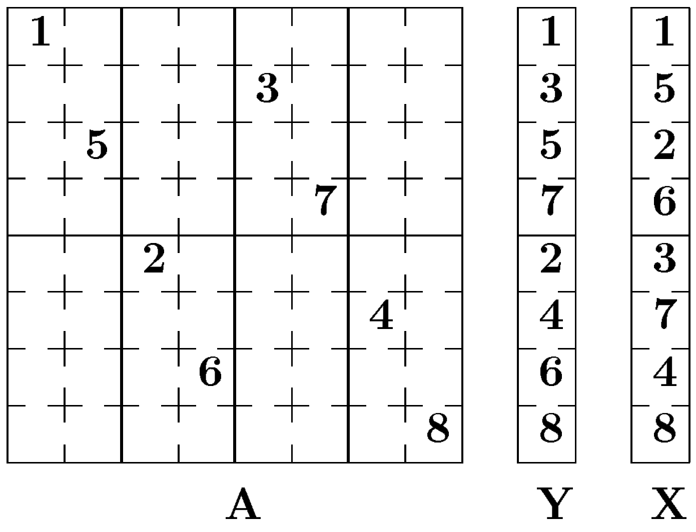

Figure 2: Block–cyclic triangular backsubstitution. In–place ordering (Y) and correct ordering (X) of solutions.

then it is natural to record the multiplier of the k^{th} block column in components (rows) $\hat{P}_1(j), \hat{P}_1(j+1), \ldots, \hat{P}_1(j+b-1)$ of Y. But, these components (rows) of the result belong in components (rows) $\hat{P}_2(j), \hat{P}_2(j+1), \ldots, \hat{P}_2(j+b-1)$ of Y. Hence, after completing the iterative process of block column subtractions, the permutation $P_2P_1^{-1}$ (a generalized shuffle [19]), must be applied to Y to obtain X.

The ordering of Y after the in–place substitution process, and the correct ordering of the solution are illustrated in Figure 2 for an 8×8 matrix mapped to a 2×4 nodal array. The numbers indicate the pivoting order for the factorization, and the placement of the solution components after the in–place backsubstitution, and after the postpermutation. In the example, the destination address in the postpermutation is obtained as by a right cyclic shift of the row index in the solution matrix. For instance, with indices labeled from 0, the content in location 1 of Y is send to location 4, the content of location 4 to location 2, and the content of location 2 to location 1.

Briefly put, the natural process for solving block–cyclic triangular systems evaluates $W = (P_1TP_1^{-1})^{-1}Y$. But, $X = (P_1TP_2^{-1})^{-1}Y = (P_2P_1^{-1})(P_1TP_1^{-1})^{-1}Y = (P_2P_1^{-1})W$. Thus, the solutions X, aligned with A and overwriting the aligned matrix Y, are obtained through a postpermutation $P_2P_1^{-1}$ applied to the result of the in–place block–cyclic triangular solver.

Note that in applying a block–cyclic elimination order to data allocated by a consecutive allocation scheme, the computations are load–balanced both for factorization and triangular system solution without permuting the matrix A, or the block–cyclic triangles. Whenever, $P_1 \neq P_2$, for instance for rectangular nodal arrays, a shuffle permutation of the solution matrix is required. But, for few right–hand sides, the amount of data permuted is considerably less than if the input matrix and the right–hand side matrix had been permuted from consecutive to cyclic allocation, and the solution matrix permuted from cyclic to consecutive allocation.

5.3 LU factorization and block–cyclic triangular system solution with partial pivoting

We have now presented all the elements of our algorithm for solving dense systems of equations with partial pivoting for a consecutive data allocation. The complete algorithm is as follows:

- Factor the matrix A in–place using a block–cyclic elimination order. Exchange each selected pivot row with the appropriate row defined by the block–cyclic elimination order. Record the pivot selection.
- Align the right–hand side matrix Y with the matrix A (the array storing the block–cyclic triangles). Y and A can be aligned at compile time through the use of the compiler directive ALIGN.
- Perform a forward block–cyclic triangular system solve in–place using the recorded pivoting information.
- Perform a backward block–cyclic triangular system solve in–place.
- Whenever the row and column permutation matrices P_1 and P_2 are different, as in the case of rectangular nodal arrays, perform a postpermutation (generalized shuffle) of the solution matrix (or prepermutation (generalized shuffle) of the matrix A as explained in the next subsection).
- Align the solutions with Y, assuming that the solutions overwrites the right–hand sides.

No permutation of the matrix A, or the block–cyclic triangular factors, is made for load–balance. A permutation of the solution matrix is only required when $P_1 \neq P_2$, as in the case of rectangular nodal arrays. For few right–hand sides, the size of the solution matrix is insignificant compared to the matrix A, and the time for permutation of the result of little influence on the performance. For a large number of right–hand sides, a prepermutation of the matrix A, as explained in the next section, may yield better performance. In our implementation, the permutation of the result, when required, is performed by the Connection Machine system router. Optimal algorithms [19] are known for the generalized shuffle permutations required in pre or postpermutation when $N/\max(p, q)$ is a multiple of the block size. However, no implementation of such algorithms is currently available on the Connection Machine systems.

Note that if the data arrays A and Y are not aligned at compile time, then the last two steps could be combined into a single routing operation. Note further, that if there are more right–hand sides than columns of A, then it may be preferable to align A with Y, if the optimal nodal array shapes for factorization and solution phases are different, or if alignment is made at run–time, since fewer data elements must be moved if the layout of Y is the reference. Optimal array shapes are discussed further in Section 7.

In our implementation, the location of the pivot block row is defined by the block–cyclic elimination order. Selected pivot rows are exchanged with rows of the pivot block row in order to assure load–balance.

The forward and backward solution in our implementation correspond to the routine _TRSM in the level–3 BLAS, but it is not identical since a block–cyclic ordered elimination is used.

5.4 LU factorization and triangular system solution without pivoting: pre vs. postpermutation

In Section 5.2 we showed that in case $P_1 \neq P_2$, backward elimination of a block–cyclic triangular system consists of two parts: a fairly standard backward elimination with reverse block–cyclic ordered block column subtractions from the right–hand side, and a postpermutation of the result when $P_1 \neq P_2$. For such block–cyclic elimination orders, successive blocks are *not* selected from the diagonal of the matrix.

LU factorization without pivoting selects pivots from the diagonal of a matrix, and works well for diagonally dominant systems precisely because the large diagonal elements are guaranteed to be the pivots in the factorization process. However, in block–cyclic ordered elimination with $\hat{P}_1 \neq \hat{P}_2$, the matrix entries used as pivots will be on the block–cyclic diagonal, i.e., in locations of the form $(\hat{P}_1(i), \hat{P}_2(i))$. Thus, for diagonally dominant matrices factored on, for instance, a rectangular nodal array with a no pivoting strategy, it is necessary to prepermute the original matrix in some way that maps the original diagonal to the block–cyclic diagonal.

One obvious prepermutation is to replace A with $B = P_1 A P_2^{-1}$. Then, given a factorization $L_B^{-1} B = P_1 T_B P_2^{-1}$, and routines to apply L_B^{-1} and $(P_1 T_B P_1^{-1})^{-1}$, $AX = Y$ can be solved as $X = P_1^{-1}\ (P_1 T_B P_1^{-1})^{-1}\ L_B^{-1}\ P_1\ Y$. But, a more efficient approach is to replace A with $C = A P_1 P_2^{-1}$. A block–cyclic elimination order applied to C yields $L_C^{-1} C = P_1 T_C P_2^{-1}$. Furthermore, $L_C^{-1} A = P_1 T_C P_1^{-1}$, and the solution of $AX = Y$ is obtained as $X = (P_1 T_C P_1^{-1})^{-1}\ L_C^{-1}\ Y$. In summary, a prepermutation of A required for numerical stability, can be used to cancel the postpermutation of the result.

Thus, in a block–cyclic elimination order on rectangular nodal arrays, LU factorization with pivoting on the diagonal is performed as follows

Apply $P_1 P_2^{-1}$ from the right.	$A \leftarrow A P_1 P_2^{-1}$
Factor C **in–place.**	L_C, $P_1 T_C P_2^{-1}$
Forward solve in–place.	$Y \leftarrow L_C^{-1} Y$
Backward solve in–place.	$X = (P_1 T_C P_1^{-1})^{-1} Y$

Note that if $P_1 = P_2$, then the first step has no effect on the data ordering, and should be omitted. In this case, the factorization with no pivoting should be faster than with pivoting, since no time is required for finding pivots, and for exchanging selected pivot rows with the rows of the block pivot row. However, when $P_1 \neq P_2$, then the no pivot option requires a prepermutation of the matrix A, while the pivot option requires postpermutation of the solution matrix X in addition to the data rearrangement required for pivoting. In our implementation we find that no pivoting is always faster than pivoting even when $P_1 \neq P_2$ and there is only one right hand side (see Figure 5).

Note further that the prepermutation $A \leftarrow AP_1P_2^{-1}$ can be used also for LU factorization with pivoting, thus removing the need for the postpermutation of the result. This technique gives improved performance when there are more columns in the right–hand side array Y, than in the original matrix A. Thus, prepermuting the matrix A when there are many right–hand sides, should result in better performance for the no pivoting option, than for the pivoting option.

5.5 QR factorization and system solution

QR factorization and the solution of the factored matrix equations can be performed in a manner analogous to the LU factorization and the solution of triangular systems. In the factorization, pivoting is replaced by an inner product of the current column with all other columns, i.e., a vector–matrix multiplication. This vector–matrix multiplication doubles the number of floating–point operations, and replace two copy–spread operations with one physical spread–with–add operation. The result is an operator Q^T in factored form, satisfying $Q^TA = R$, where $R = P_1TP_2^{-1}$ is a block–cyclic upper triangle. If A has m rows and n columns with $m \geq n$, then the QR factorization may be used to find the vector X that minimizes the residual 2-norm $\|AX - Y\|_2$. The procedure is

Apply Q^T.	$Y \leftarrow Q^TY$
Backward solve.	$X = \Pi(P_2P_1^{-1})(P_1TP_1^{-1})^{-1}Y$

where Π is projection onto the first n components. The backsolve $(P_1TP_1^{-1})^{-1}$ operates on the first n rows in block–cyclic order of Y. The final permutation $P_2P_1^{-1}$ moves the first n block–cyclic rows to the first n rows.

6 Performance

In this section we first report the performance achieved on a few matrix sizes. Then, we analyze the performance, and discuss factors that contribute to the discrepancy between

No. of rows/	LU		QR	
columns, N	32–bit	64–bit	32–bit	64–bit
1024	172	139	374	321
2048	620	494	1301	1053
4096	2040	1537	3902	2842
8192	5586	3846	8878	5704
16384	11573	7173	—	—
26624	16030	9398	—	—

Table 6: Execution rates in Mflops/s for LU and QR factorization on a 2048 node Connection Machine system CM-200. A blocking factor b of 16 was used for LU factorization. A blocking factor of 8 was used for QR factorization. Nodes in Gray code order.

the performance of the local matrix kernels and the performance of the global factor and solve routines. Communication and arithmetic are considered separately. All arrays are allocated with default compiler layouts. This implies that nodes are in Gray code order along each axis, and that Y is aligned with A only in the case that the number of right hand sides is equal to the number of equations. A 2048 node and a 512 node Connection Machine system CM–200 were used for the performance measurements. Both systems result in rectangular nodal arrays. Alignment of Y with A (when necessary) and postpermutation of the solution(s) were performed using the Connection Machine router. Times for these data motion steps are included in the solve timings. In all cases we used random data for the matrix A. In this section, LU factorization always means LU factorization with partial pivoting. In this section, QR factorization always means QR factorization with no pivoting.

6.1 Measurements

In 64–bit precision the peak performance for LU factorization is 9.4 Gflops/s, or close to 4.6 Mflops/s per node, which is about 50% of the peak performance of the level–2 LBLAS. Table 6 shows the performance of LU and QR factorization for a few matrix sizes on a 2048 node Connection Machine system CM–200. The execution rate for LU factorization is computed based on $\frac{2}{3}N^3$ floating–point operations, and the QR factorization rate based on $\frac{4}{3}N^3$ operations, regardless of the blocking factor. For the solvers the execution rates are based on $2RN^2$ operations for the LU solver, and $3RN^2$ for the QR solver.

Tables 7 and 8 give some performance data for the factorization and the forward and backward solution of dense systems. R denotes the number of right–hand sides in the systems of equations being solved. The improved efficiency with an increasing matrix size, increased number of right–hand sides, and increasing blocking factor is apparent. For LU factorization in 32–bit precision the performance increases by a factor of over 700 for an increase in matrix dimension N by a factor of 132 and a blocking factor of 1. With a blocking factor of 16, the same increase in matrix size yields a performance increase by a factor of over 1300. The performance increase in 64–bit precision for an increase in N by a factor of 104 is 425 and 840, respectively. For small values of N, the performance increases much faster than N. For large values of N, the performance increase is still substantial, and roughly proportional to N. This characteristic is true for both the factorization and the solution phases.

The execution rate for the forward and backward block–cyclic triangular system solvers increases significantly with the matrix size for a large number of right–hand sides. With the number of right–hand sides equal to the number of equations, the performance of the triangular solvers is about 50% higher than for LU factorization.

Comparing the execution rates of the LU and QR factorization routines, we observe that except for large matrices, the execution rates of the QR factorization routine is approximately twice that of the LU factorization routine. Hence, the execution times are approximately equal. However, for large matrices in 64–bit precision, the execution rate for QR factorization is only 15 – 20% higher than for LU-decomposition, and the execution time for QR factorization is significantly longer than for LU factorization.

Comparing the LU and QR solve routines, we observe that the QR solve routine benefits more from an increased blocking factor than the LU solve routine. The execution rate of the QR solve routine is about 20% higher than the LU solve routine for a large number of right–hand sides and a blocking factor of 1. But, for a blocking factor of 16, the QR solve routine has an execution rate that is about 40% higher than that of the LU solve routine.

The optimum blocking factors for LU factorization and triangular system solution are summarized in Figure 3. The optimum blocking factor increases with an increased matrix size. For LU factorization the sensitivity of the execution rate with respect to the blocking factor is small (on the order of 20%) for small matrices, while for large matrices an increase in the blocking factor from 1 to 16 may increase the performance by a factor of more than two. The behavior is similar for the LU solve routine. The optimum blocking factor increases with the matrix size. In our implementation the optimum blocking factor for LU factorization is 4 for matrices of size up to 1024, 8 for matrices of size between 1024 and 8096, and 16 for sizes 8096 to 16896, the maximum size used in our test. Note that the optimum blocking factor for the LU solve routine in general is higher than for the factorization routine. Figures 4 and 6 show the execution rate as a function of matrix size and blocking factor for LU factorization and solve, respectively. Figure 7 shows the execution rate of the LU solve routine as a function of matrix size and the number of right–hand sides.

The optimum blocking factor for the QR factorization routine increases somewhat slower with the matrix size than for the LU factorization routine. The optimum blocking factor for the QR solve routine is higher than for the QR factorization routine, just as in the case of LU factorization and solve. Figure 8 shows the execution rate as a function of matrix size and blocking factor for QR factorization.

In the next few subsections we analyze the performance behavior of the communication functions, and the local arithmetic as a function of matrix and machine size, and provide a model to predict the performance.

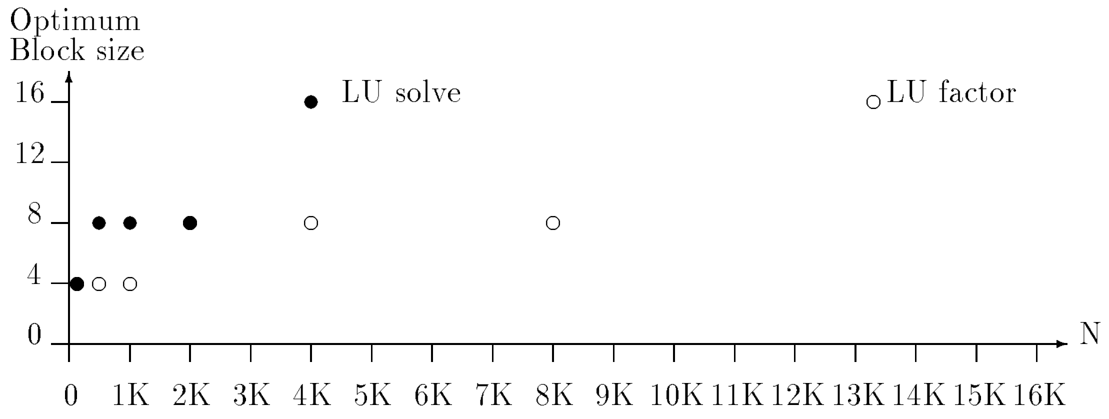

Figure 3: Optimal blocking factor for LU factorization and triangular system solution.

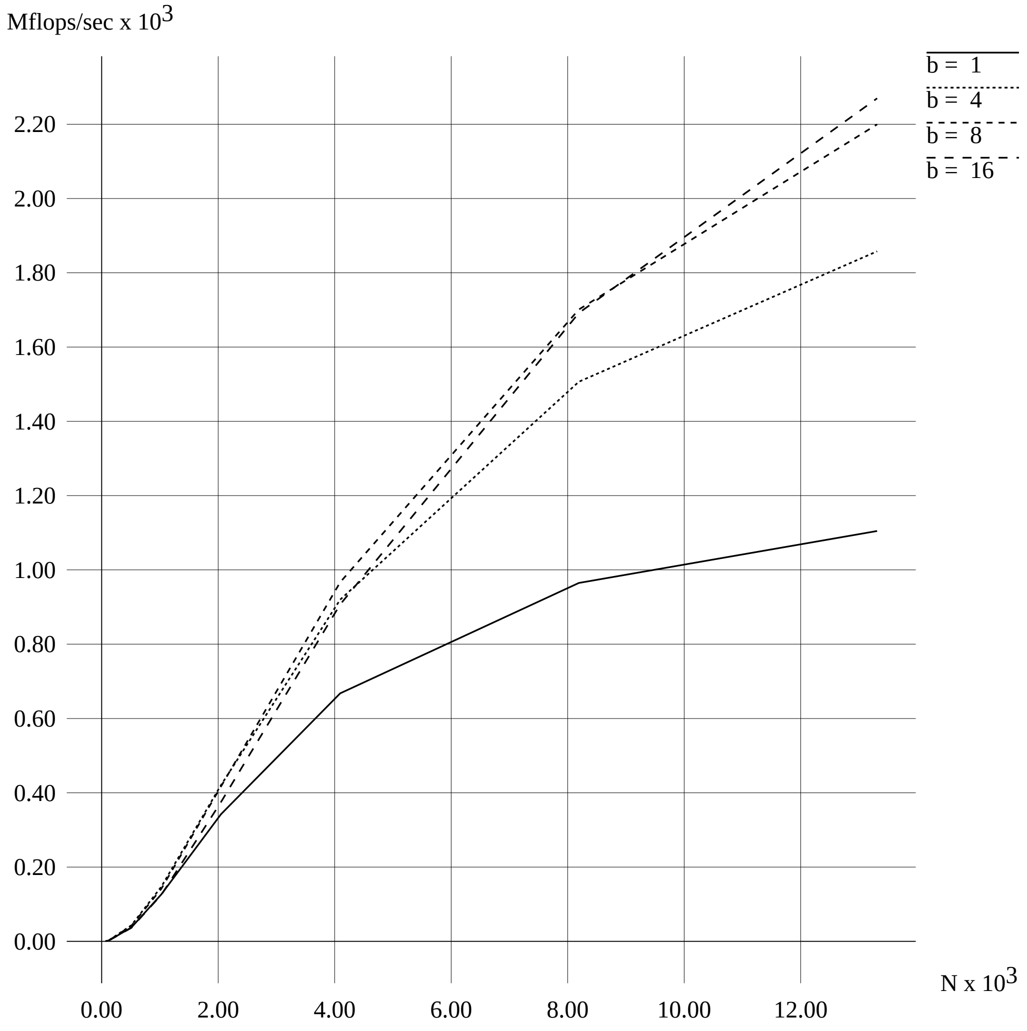

Figure 4: Execution rates for LU factorization on a 512 node Connection Machine system CM–200 with blocking factor b, 64–bit precision. Nodes in Gray code order.

Matrix size N	Block size b	32–bit precision Factor	32–bit Solve R = 1	32–bit Solve R =N/2	32–bit Solve R = N	64–bit precision Factor	64–bit Solve R = 1	64–bit Solve R = N/2	64–bit Solve R = N
128	1	2.7	.1	3.1	6.1	2.6	.1	3.1	6.0
512	1	39.9	.3	45.1	82.8	37.5	.2	42.0	72.8
1024	1	140.7	.5	153.0	252.0	126.4	.4	131.5	198.7
2048	1	420.1	.9	417.9	624.8	342.1	.7	320.7	434.8
4096	1	944.3	1.5	915.9	1127.9	667.7	1.1	615.1	713.1
8192	1	1510.1	2.2			964.7	1.5		
13312	1	1804.3	2.7			1104.8	1.8		
16896	1	1910.2	3.0						
128	4	3.3	.1	4.2	8.3	3.0	.1	4.1	8.0
512	4	48.8	.5	62.2	112.8	43.3	.4	55.9	96.7
1024	4	170.0	.9	213.1	348.8	146.7	.8	179.1	277.2
2048	4	517.8	1.7	614.9	892.7	420.3	1.5	470.8	639.3
4096	4	1237.5	3.1	1407.0	1726.8	920.0	2.6	970.9	1143.3
8192	4	2204.8	5.1			1506.6	3.9		
13312	4	2835.7	6.9			1857.9	4.8		
16896	4	3099.1	7.7						
128	8	3.2	.1	4.1	8.2	2.9	.1	3.9	7.6
512	8	48.6	.5	65.4	117.8	42.0	.5	57.4	98.8
1024	8	169.1	1.0	224.3	369.2	141.5	.9	184.4	286.5
2048	8	524.1	2.0	667.5	972.4	416.8	1.7	501.1	680.4
4096	8	1330.9	3.8	1603.3	2013.3	966.9	3.2	1075.8	1289.2
8192	8	2561.9	6.6			1701.1	5.1		
13312	8	3471.8	9.2			2200.0	6.7		
16896	8	3876.7	10.5						
128	16	3.1	.1	3.9	7.6	2.7	.1	3.4	6.7
512	16	43.8	.6	64.6	116.9	36.4	.5	55.2	95.6
1024	16	154.8	1.1	223.4	371.8	125.1	1.0	179.9	281.6
2048	16	485.5	2.2	680.9	979.2	374.6	1.9	498.1	674.0
4096	16	1268.3	4.2	1651.2	2085.3	907.1	3.5	1091.7	1317.4
8192	16	2574.8	7.5			1691.8	6.0		
13312	16	3628.9	10.8			2270.6	8.1		
16896	16	4116.9	12.6						

Table 7: Execution rates in Mflops/s for LU factorization and block–cyclic triangular system solution on a 512 node Connection Machine system CM–200 as a function of matrix size, blocking factor, and the number of right–hand sides. Nodes in Gray code order.

Matrix size N	Block size b	32–bit precision Factor	32–bit Solve R = 1	32–bit Solve R =N/2	32–bit Solve R = N	64–bit precision Factor	64–bit Solve R = 1	64–bit Solve R = N/2	64–bit Solve R = N
128	1	5.8	.1	6.2	11.9	5.3	.1	5.6	10.5
512	1	85.7	.5	83.3	142.0	74.2	.5	66.9	104.6
1024	1	299.3	1.0	261.1	390.7	237.3	.8	186.9	255.6
2048	1	865.1	1.8	639.8	876.5	605.3	1.3	412.0	514.4
4096	1	1827.3	2.8	1297.0	1510.4	1104.9	1.9	744.7	827.1
8192	1	2750.0	4.0			1516.7	2.4		
13312	1	3183.7	4.8			1701.0	2.7		
16896	1	3334.3	5.1						
128	4	7.2	.4	9.8	19.1	6.3	.3	8.4	16.2
512	4	106.6	1.6	134.6	235.5	88.0	1.2	108.0	179.6
1024	4	371.5	3.0	436.0	675.1	284.7	2.2	320.4	469.6
2048	4	1101.4	5.5	1126.1	1534.4	749.3	3.9	748.3	960.1
4096	4	2427.1	9.1	2235.1	2601.2	1471.4	5.8	1311.3	1476.8
8192	4	3902.2	12.9			2178.2	7.6		
13312	4	4694.3	15.3			2544.4	8.6		
16896	4	4986.5	16.3						
128	8	7.0	.4	9.4	18.2	6.0	.3	7.8	14.8
512	8	102.9	2.3	146.2	253.9	80.6	1.6	111.6	184.5
1024	8	359.8	4.4	469.1	724.7	262.3	2.9	334.5	496.6
2048	8	1088.8	8.1	1272.7	1698.4	711.9	5.3	818.4	1034.9
4096	8	2548.5	13.8	2557.4	3012.5	1475.9	8.5	1555.8	1793.4
8192	8	4361.5	20.4			2315.9	11.5		
13312	8	5424.8	24.2			2797.1	13.2		
16896	8	5838.8	25.8						
128	16	6.3	.4	8.4	16.0	4.9	.3	6.3	11.7
512	16	85.6	2.8	143.2	251.7	63.4	1.8	105.5	176.7
1024	16	300.9	5.4	470.6	744.4	206.9	3.4	324.3	484.7
2048	16	917.4	10.2	1295.5	1729.7	572.9	6.2	814.3	1048.4
4096	16	2227.2	17.7	2762.9	3297.1	1253.9	10.2	1577.7	1827.3
8192	16	4068.2	26.9			2113.7	14.7		
13312	16	5276.3	32.5			2669.0	17.4		
16896	16	5780.5	34.8						

Table 8: Execution rates in Mflops/s for QR factorization and block–cyclic triangular system solution on a 512 node Connection Machine system CM–200 as a function of matrix size, blocking factor, and the number of right–hand sides. Nodes in Gray code order.

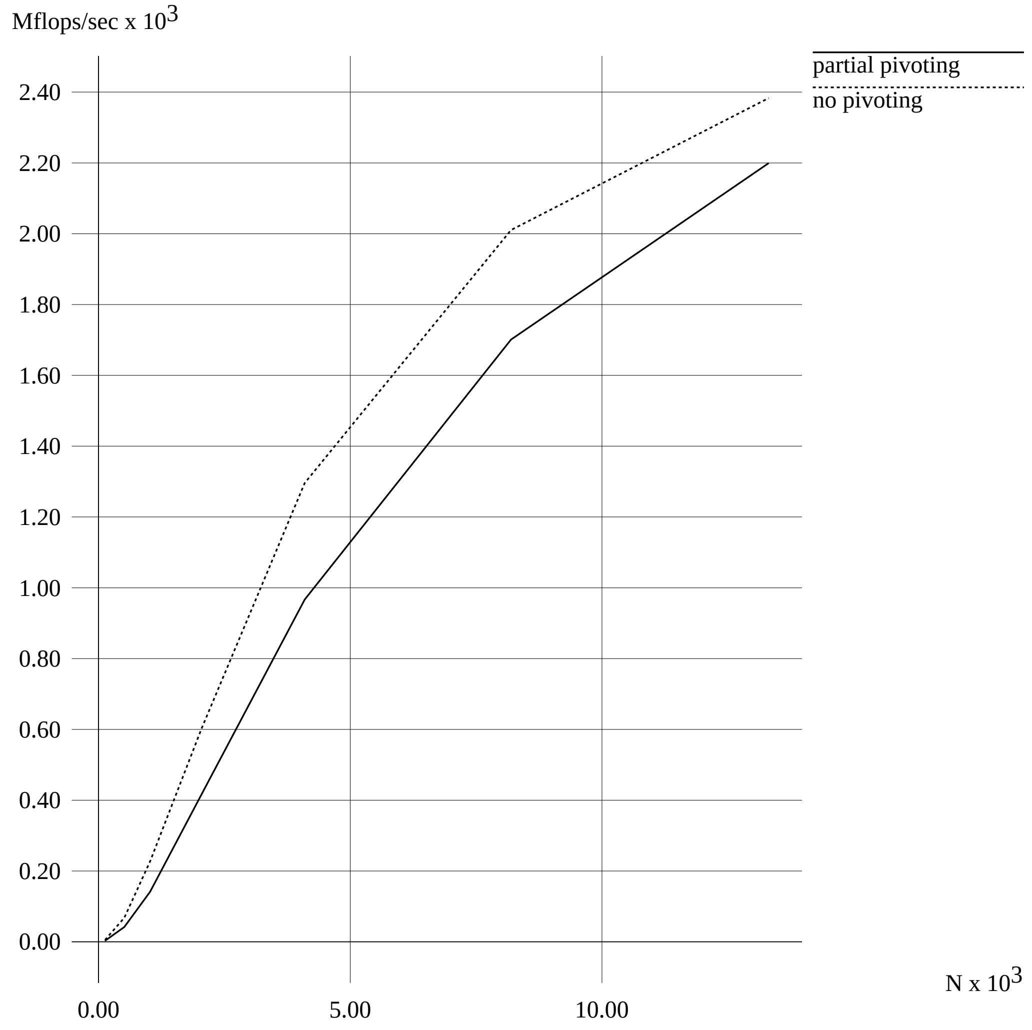

Figure 5: Execution rates for LU factorization with and without pivoting on a 512 node Connection Machine system CM–200 with blocking factor 8, 64–bit precision. Nodes in Gray code order.

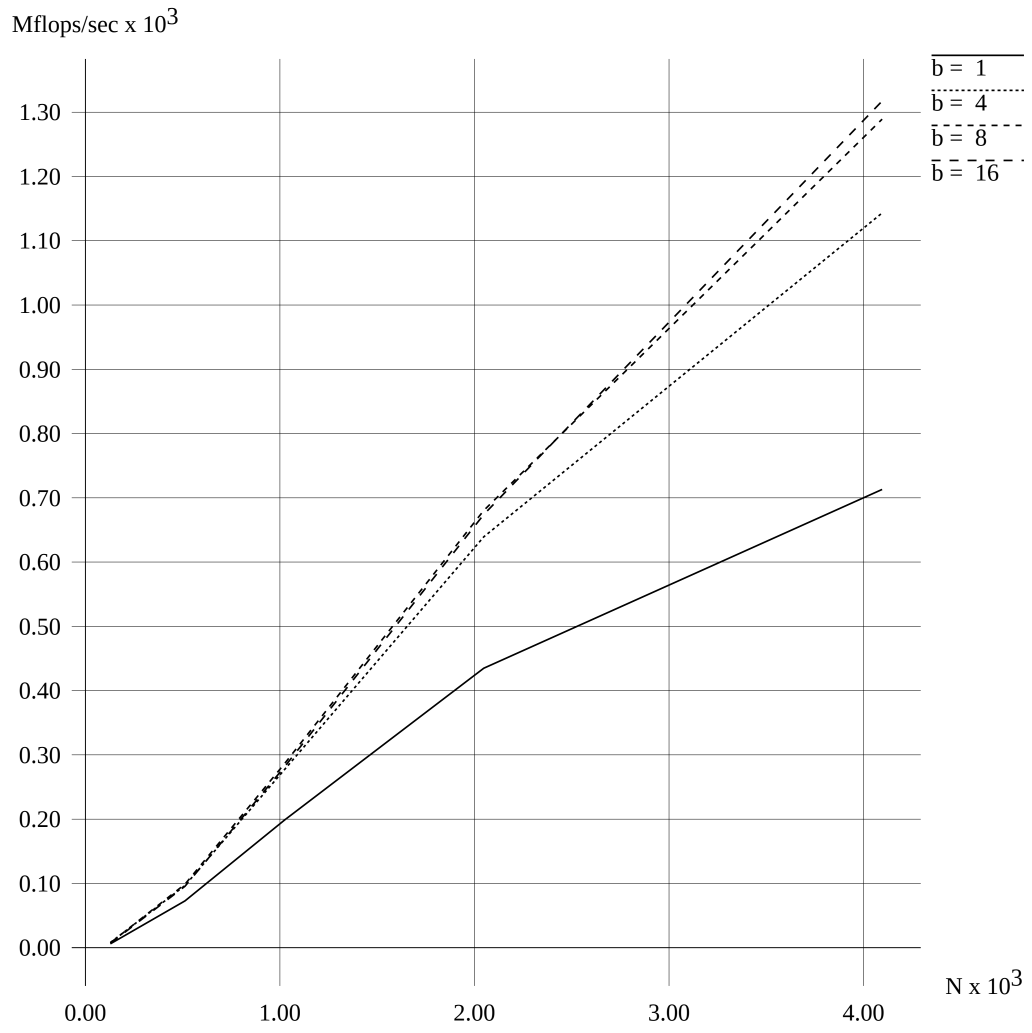

Figure 6: Execution rates for solution of block–cyclic triangular systems of equations with N right–hand sides for LU decomposition on a 512 node Connection Machine system CM–200 with a blocking factor b, 64–bit precision. Nodes in Gray code order.

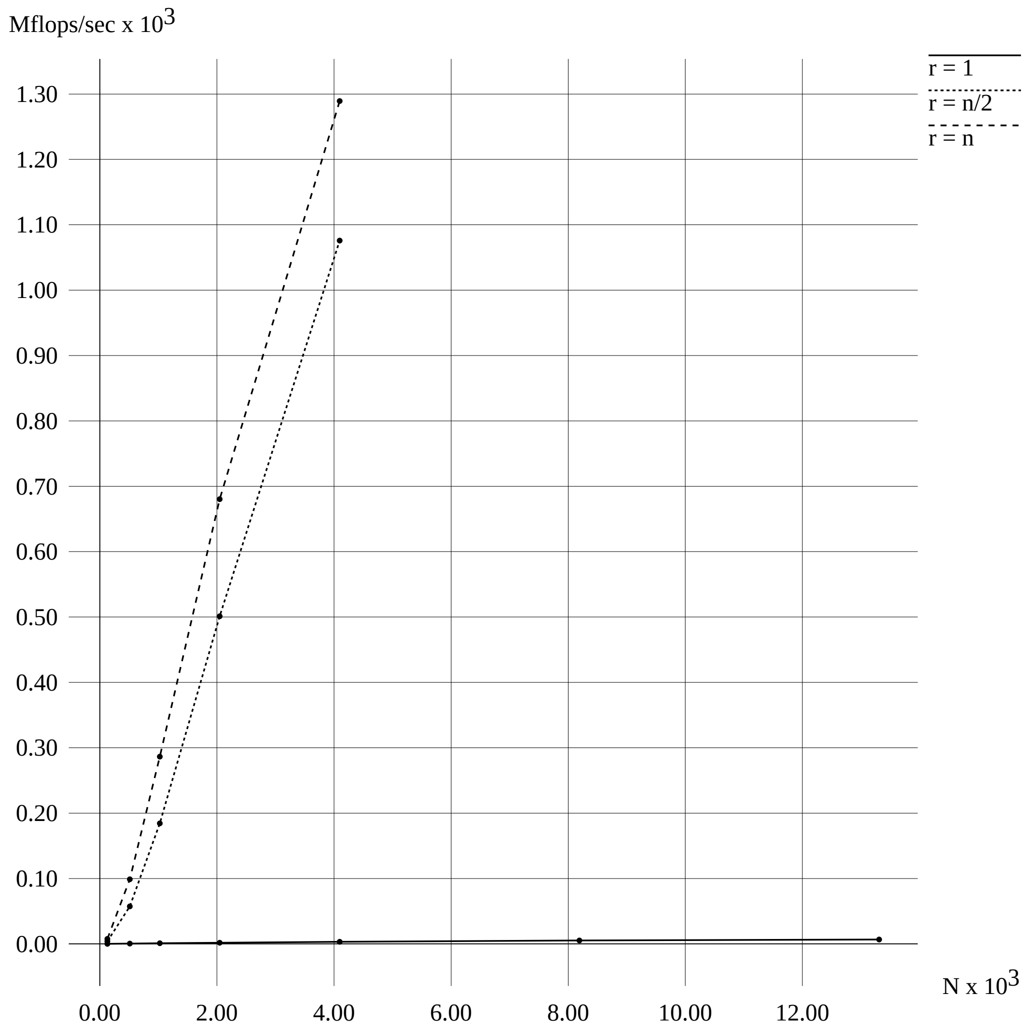

Figure 7: Execution rate for solution of block–cyclic triangular systems of equations with r right–hand sides for LU decomposition on a 512 node Connection Machine system CM–200 with a blocking factor 8, 64–bit precision. Nodes in Gray code order.

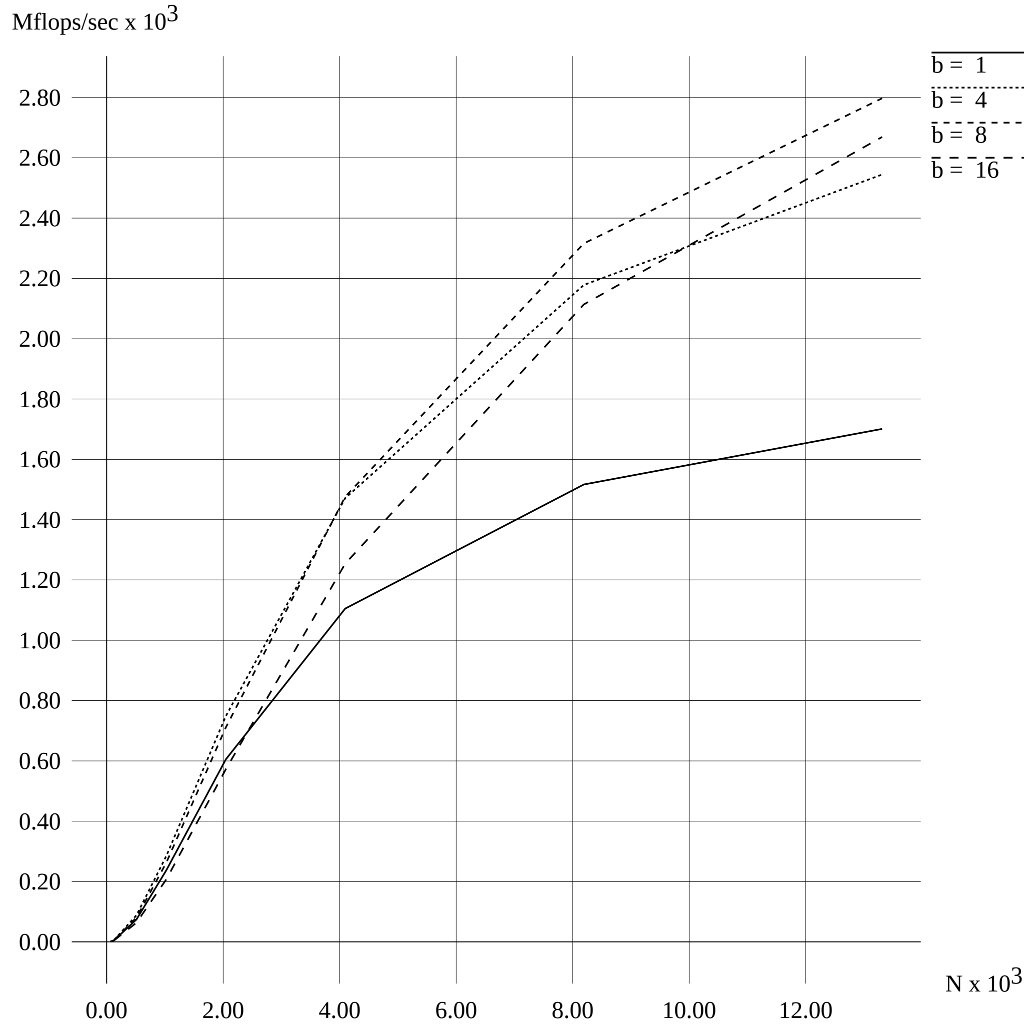

Figure 8: Execution rates for QR factorization on a 2048 node Connection Machine system CM–200 with a blocking factor b, 64–bit precision. Nodes in Gray code order.

6.2 Communication

Two main types of communication are required: 1) *Spreads* copy data from a single row (or column) of nodes to all other rows (or columns), 2) *Reductions* to select the pivot row in LU factorization with partial pivoting, and for column summations in QR factorization. In addition, general *sends* are used to align the right hand sides Y with the columns of A, to realign the solutions with the input array Y, and, when $P_1 \neq P_2$, to prepermute the matrix A, or to postpermute the solution(s). In a spread of a pivot column, $\frac{N}{p}$ elements per node must be communicated to every other node in a row for the consecutive data allocation described in Section 5.1. With a blocking factor of b rows and columns the number of elements to be spread for a block column is $\frac{Nb}{p}$. Similarly, a block row requires that $\frac{Nb}{q}$ elements be spread to every other node in a column.

On the Connection Machine system CM–200, where the nodes are interconnected as a Boolean cube, there exist several paths from the node that must spread the data to any of the nodes receiving the data. Indeed, since p and q are always chosen such that the nodes form subcubes of the Boolean cube, there exist $\log_2 p$ and $\log_2 q$ edge-disjoint paths between a node and every other node in a column and row subcube. Hence, the data set a node must spread can be divided up among the different paths to balance the communications load and maximize the effective use of the available communications bandwidth [18]. Spreads on the Connection Machine system CM–200 are implemented in this manner, and use the communications bandwidth optimally [18]. Note that for a fixed data set per node, the time for a spread decreases with an increased number of nodes. Table 9 gives some timings for spreads of different size data sets on Connection Machine systems CM–200 of various sizes. For a large data set per node, increasing the number of cube dimensions by a factor of 10 (from 1 to 10) reduces the time for a spread by a factor of 3.41. Increasing the number of dimensions by a factor of 5 yields a reduction in the time for a spread of a large data set by a factor of 2.06. The overhead is quite substantial, making the speedup significantly less than the increase in the number of cube dimensions.

The prepermutation $P_1 P_2^{-1}$, or the postpermutation $P_2 P_1^{-1}$, when required, is performed by the Connection Machine system CM–200 router in our implementation. These permutations are *generalized shuffle* permutations. Optimal algorithms for Boolean cube networks are known for these permutations when $N/\max(p,q)$ is a multiple of the block size [19], but not implemented on the Connection Machine system CM–200. In cases with only one right hand side, the cost of the alignment and postpermutation is only a small fraction of the total cost of the solve.

Note that, on the Connection Machine systems, the default layout of the matrix A and the right–hand sides Y is typically not the same, since the nodal array shape depends upon the data array shape. Aligning A and Y at compile time avoids data motion at run–time. With a default layout of Y and A, the alignment constitutes a shuffle permutation, which would be performed by the router. Similarly, with the solutions overwriting the right–hand sides, the default data allocation requires a reallocation of the result from being aligned with A, to the default layout. This reallocation could be combined with a postpermutation of the solution matrix X, when necessary, into a single permutation.

Number of	Number of nodes										
elements	2	4	8	16	32	64	128	256	512	1024	2048
2	0.0366	0.383	0.448	0.502	0.573	0.640	0.719	0.789	0.878	0.965	1.036
4	0.0366	0.383	0.448	0.502	0.573	0.640	0.719	0.789	0.878	0.965	1.036
8	0.0623	0.416	0.480	0.505	0.576	0.644	0.722	0.792	0.881	0.968	1.040
16	0.1136	0.470	0.515	0.542	0.615	0.684	0.763	0.798	0.888	0.975	1.046
32	0.2162	0.577	0.598	0.600	0.691	0.730	0.811	0.848	0.939	1.027	1.101
64	0.4214	0.793	0.741	0.720	0.773	0.856	0.907	0.947	1.042	1.133	1.168
128	0.8318	1.223	1.055	0.962	0.963	0.984	1.044	1.045	1.247	1.305	1.344
256	1.6527	2.084	1.654	1.445	1.377	1.352	1.360	1.338	1.410	1.434	1.503
512	3.3139	3.805	2.882	2.410	2.203	2.054	1.990	1.925	1.946	1.953	2.001
1024	6.7319	7.247	5.310	4.342	3.825	3.491	3.287	3.098	3.021	2.991	2.957
2048	13.4530	14.129	10.193	8.204	7.067	6.331	5.844	5.445	5.210	5.028	4.910
4096	26.9059	27.895	19.931	15.929	13.585	12.048	10.960	10.137	9.588	9.102	8.782
8192	53.8115	55.426	39.437	31.379	26.619	23.444	21.226	19.522	18.344	17.288	16.520
16384	107.6210	110.476	78.415	62.279	52.657	46.269	41.721	38.291	35.818	33.662	31.997
32768	215.2400	220.577	156.396	124.070	104.723	91.874	82.711	75.829	70.764	66.366	62.991

Table 9: Time in msec for spreads of different size data sets and Connection Machine systems CM–200 of various sizes. 32–bit precision.

Such a combined permutation would require the same time as either the postpermutation itself, or the reallocation of X, using optimal algorithms [19].

Note also, that if all arrays have a default layout, then for many right–hand sides it may be advantageous with respect to performance to align the matrix A with Y and, if a prepermutation of A is necessary, combine this permutation with the alignment operation.

6.3 Arithmetic efficiency

Not all the work is well load–balanced, even in the block–cyclic order elimination. Some work is applied only to a single row (or column) of the matrix at a time. For example, in LU factorization with partial pivoting, it is necessary to find the location of the largest element in the current column. Though this work is of order $O(N)$ for a single (block) row or column, compared to the $O(N^2)$ work for a rank–b update, it is magnified in importance because the grid–based layouts do not load–balance individual rows and columns across the whole machine. Despite this drawback, grid–based layouts are optimal for communication systems in which the communication time is determined by the data volume leaving or entering a node, as shown in Section 7.

6.4 Detailed performance analysis

For simplicity, in this section and the next, we consider only the case $p = q$.

Assuming that αN^3 floating–point operations are carried out on p^2 nodes running at rate r, while the time per unit of work spent on N^2 work (some arithmetic and communication) on p nodes is C, the total time is $\frac{\alpha N^3}{rp^2} + C\frac{N^2}{p}$. The achieved computational rate is

$$R = \frac{\alpha N^3}{\frac{\alpha N^3}{rp^2} + C\frac{N^2}{p}}$$

or

$$R = rp^2\left(\frac{\theta}{\theta + 1}\right)$$

where

$$\theta = (\frac{N}{p})(\frac{\alpha}{rC}).$$

For LU factorization on a 2048 node Connection Machine system CM–200, the compilers by default configure the nodes as a 32×64 array. The algorithm we use requires three spreads for each row and column that is eliminated; two to accomplish the pivoting row swap and one to spread the column of coefficients. From Table 9, we conclude that the time for a row spread is $570 + 3.1 *$ (number of 32–bit words per column of nodes) μsec, and the time for a column spread is $640 + 2.7 *$ (number of 32–bit words per row of nodes) μsec. For the largest matrices in Table 7, $\frac{N}{p} \approx 630$ (actually the submatrices assigned to each node are rectangular 448×896, and $\sqrt{(448 * 896)} \approx 630$). Some straightforward calculus shows that communication contributes about 15 μsec to C. By counting cycles for the $O(N^2)$ arithmetic part of the code, we estimate that it contributes another 12 μsec to C. For LU factorization $\alpha = 2/3$. We used a blocking factor of $b = 16$ for the timings reported in Table 7, i.e., all $O(N^3)$ work is performed by rank–16 updates. These run at a peak rate of about 7.6 Mflops/s per node. However, including the effect of DRAM page faults reduces this performance to a little over 7 Mflops/s. Hence, $\frac{\alpha}{rC} \approx \frac{2}{3 \cdot 7.0 \cdot 27} \approx \frac{1}{285}$, and $\theta \approx 2.2$, for which the performance loss factor is $\frac{\theta}{\theta+1} \approx 0.69$. Altogether, this analysis predicts a performance of about 4.85 Mflops/s per node, which is very close to the measured performance.

The important conclusion is that for $\theta \ll 1$, or $\frac{N}{p}$ small relative to $\frac{rC}{\alpha}$, the performance increases roughly linearly in $\frac{N}{p}$. Because rC is large, nearly all interesting cases fall into this range of submatrix sizes.

A minor point relevant for fine-tuning is that smaller problems should use smaller block sizes. This need arises because one of the contributions to C is work occurring entirely within block rows or block columns, which grows quadratically with the block size b.

6.5 Scalability

As the matrix size N increases for a fixed machine size, the number of rows and columns assigned to a node increases proportionally. Hence, if N is doubled, so is the amount of data per row *and* column per node. Hence, the time required for the $O(N^2)$ term increases in proportion to N^2. But, the amount of work that must be performed by each node for the factorization, increases in proportion to the cube of the local matrix size. Hence, the importance of the $O(N^2)$ term decreases with increased N for a fixed number of nodes. However, the significance does not decrease in proportion to N, since the efficiency of the level–2 LBLAS increases with increased N. The variation in the computational rate r spans a range of more than one order of magnitude, while the performance loss factor may span two orders of magnitude. Hence, the dramatic variation in performance as a function of the local submatrix size.

Scaling the problem size with the number of nodes such that size of the submatrix assigned to each node remains fixed increases the efficiency of our current Connection Machine system CM–200 implementation. The performance of the level–2 LBLAS is unaffected by this form of scaling, but the communication is sped up because more channels are available for spreads and reductions. Thus, increasing the number of nodes decreases the value of C.

7 Optimal layouts

Communication is inevitable for the solution of full rank factorization problems on a distributed memory computer [9]. Another source of inefficiency in our implementation is the poorly load–balanced $O(N^2)$ work. We now consider the impact on the data communication (for spreads and reductions) of some alternative data layouts intended to improve the load–balance for the $O(N^2)$ work.

Consider an arbitrary layout of an N by N matrix on p^2 nodes, which is regular in the following sense:

> Each column intersects exactly c nodes in N/c elements.
>
> Each row intersects exactly r nodes in N/r elements.
>
> There are exactly N^2/p^2 elements on each node.
>
> > Each node intersects exactly cN/p^2 columns.
> >
> > Each node intersects exactly rN/p^2 rows.

Figure 9 shows a two–dimensional data array assigned to nodes forming a one–dimensional array in a Boolean cube. A binary-reflected Gray code encoding [25] is used for the array embedding. Each square subarray of the data array is assigned to one of four nodes, as indicated by the number in the corresponding box. In effect, the data array is assigned to the nodes of the embedded one–dimensional array in a block–skewed way. This layout satisfies the regularity conditions with $c = 4$, $r = 4$, and $p = 2$.

It is clear that by making $rc > p^2$, the load–balance can be improved for row and column oriented operations. In order to evaluate the impact on the time for a spread, we assume that the time of the spread is proportional to $\max(inject, eject)$, where $inject$ is the maximum number of data elements injected into the communications system by any node,

0	1	3	2
1	0	2	3
3	2	0	1
2	3	1	0

Figure 9: Layout of a two–dimensional data array on a Boolean cube configured as a one–dimensional array.

and $eject$ is the maximum number of elements ejected from the communications system by any node. Hence, we assume that communication operations are limited by the bandwidth at the nodes. Then, the time of a column spread is $\max(N/c, rN/p^2)$ and the time of a row spread is $\max(N/r, cN/p^2)$. Now, consider an algorithm consisting of local computation, row spreads, and column spreads. Suppose that local computation consists of a perfectly load balanced part, a part which takes place on a single column at one time, and a part which takes place on a single row at one time. By making $rc > p^2$ we can reduce the cost of the single column and single row operations. In return, the cost of the row spreads and the column spreads will grow. In fact, it is likely that distributing rows and columns across more processors will also increase the cost of the perfectly load balanced part of the computation, since more distributed layouts will reduce local vector lengths. If we ignore this effect of reduced vector lengths, then an optimal layout should minimize $cost = \alpha * \max(N/c, rN/p^2) + \beta * \max(N/r, cN/p^2) + \gamma * N/r + \delta * N/c$, where α,β,γ, and δ, are parameters describing the relative importance of column spread, row spread, single row work, and single column work, respectively.

To find the values of c and r that minimize this expression, we first assume that $rc = K$ with $K \geq p^2$. Then, $cN/p^2 \geq N/r$ and $rN/p^2 \geq N/c$, and the minimization problem reduces to finding the minimum of $(\alpha/p^2) * r + (\beta/p^2) * c + \gamma/r + \delta/c$, subject to the constraint $rc = K$. Using the constraint to eliminate c, and solving for the critical value of r, gives $r^2 = \frac{(\beta/p^2)*K+\gamma}{(\alpha/p^2)+\delta/K}$. Here, $cost(K) = 2 * \sqrt{((\alpha/p^2) + \delta/K) * ((\beta/p^2) * K + \gamma)}$. Minimizing this minimum cost with respect to K gives $K_{min} = \sqrt{\frac{\gamma*\delta}{\alpha*\beta}} * p^2$. For $K > K_{min}$, $cost(K)$ is increasing. So if $\gamma\delta < \alpha\beta$, the minimum $cost$ for all $K \geq p^2$ occurs for $K = p^2$. If we take $K < p^2$, then the minimization problem reduces to finding the minimum of $(\alpha + \delta)/c + (\beta + \gamma)/r$ subject to the constraint $rc = K$. In this case, the minimum is a decreasing function of K, so the minimum cost for all $K < p^2$ occurs for $K = p^2$.

The above analysis assumes that the number of columns is equal to the number of rows, which is true for LU factorization on a matrix that fits in primary storage. For the triangular solve, the number of right hand sides R is often not equal to the number of rows N. With R right hand sides, the assumption that each column intersects c nodes in N/c elements is still valid. But, each row now intersects r nodes in R/r elements. With NR/p^2 elements on each node, each node intersects cR/p^2 columns (instead of cN/p^2 columns) and rN/p^2 rows, as before. It is easily verified that even for the triangular solve, $K = p^2$ is optimal.

The overall conclusion is that, provided $\gamma\delta < \alpha\beta$, i.e., provided the geometric mean of the single row and single column work is less than the geometric mean of the row and column spread work, the best choice is $K = p^2$. The condition $rc = p^2$ implies that the optimal layout is indeed based on two–dimensional subgrids for both factorization and triangular system solution. This conclusion is true, because any column intersects a node in N/c elements and $N/c = rN/p^2$, which is the number of rows that meet any node. Thus, the rows meeting any node must be exactly those N/c rows that meet any one of the columns meeting that node. In our implementations of LU and QR factorization, the condition $\gamma\delta < \alpha\beta$ is satisfied, (although not by a wide margin). The optimal value of r/c under the constraint $rc = p^2$ is $\frac{\beta+\gamma}{\alpha+\delta}\frac{R}{N}$. Thus, the ratio of the lengths of the axes of the two–dimensional nodal array is proportional to the ratio between the corresponding axes of the data array operated upon for both factorization and triangular system solution.

The above communications model applies to some, but not all architectures. For instance, for hypercubes with concurrent communication on all channels of every node, such as the Connection Machine system CM-200, the above communications model must be modified. In the example given by the diagram above, a column spread and a row spread are actually all–to–all broadcasts [2, 18]. The time for such an operation is proportional to the number of elements entering a node divided by the number of communication channels per node, i.e., the number of hypercube dimensions spanning the set of nodes involved in the all–to–all broadcast. For a 2-dimensional hypercube, all–to–all broadcast is no more expensive than a standard column spread. Although twice as much data must be received by each node, there are twice as many channels available for the data to use. Therefore on a four node CM–200, the data layout shown in Figure 9 would give better performance for dense linear algebra than any layout currently used, because with the Figure 9 layout and cyclic elimination order, all vector operations would be perfectly load–balanced.

8 Summary

We describe LU and QR factorization and solve algorithms for a block–cyclic ordered elimination for both square and rectangular nodal arrays. We show how prepermutation can be performed to guarantee pivoting on the diagonal for diagonally dominant matrices, without a need for postpermutation. The algorithms have been implemented on the Connection Machine systems CM–2 and CM–200, and are part of the Connection Machine Scientific Subroutine Library [29]. The peak execution rate of the LU factorization routine on a Connection Machine system CM-200 is about 9.4 Gflops/s in 64–bit precision.

The routines accept any data layout that can be created in the higher level languages on the Connection Machine systems, either by the default layout rules, or through the use of compiler directives. The routines also perform operations on multiple instances concurrently, with instances distributed over different sets of nodes. The algorithms use standard communication functions, such as multiple instance (segmented) broadcast and reduction, and generalized shuffle permutations. Optimized routines are used for broadcast and reduction, while the router currently is used for the generalized shuffle permutations.

For small matrices the *execution times* for QR factorization without pivoting and LU factorization with partial pivoting are comparable, while for large matrices the *execution rates* become comparable, and hence the execution time for LU factorization with partial pivoting considerably shorter. The execution rate for the block cyclic triangular solve for LU decomposition is 50 - 100% higher than for the factorization for a number of right–hand sides equal to the number of unkowns. The behavior is similar for QR factorization.

The value of blocking operations on rows and columns increases with matrix size. For small matrices, blocking may yield an enhanced execution rate by 20%, while for large matrices the blocking may offer an increased execution rate of about a factor of two. In our implementations the optimum blocking factor for LU factorization with partial pivoting increases from 4 to 16 with the matrix size increasing from 128 to 16896. The optimum blocking factor for QR decomposition without pivoting was observed to increase slower with the matrix size than for LU factorization. The optimum blocking factors for the solve routines were always higher than for the factorization routines.

The execution rate for LU factorization without partial pivoting is higher than when partial pivoting is used. The difference in performance depends upon the problem size, but is typically in the 10 – 20% range.

The peak performance for the global factorization routines is about two thirds of the peak performance of the local level–2 BLAS routines used for the $O(N^3)$ work in the factorization (but approximately equal to the performance of the global matrix multiplication routine in the CMSSL). The $O(N^3)$ work is well load–balanced and performed at high efficiency through blocking of row and column operations. The $O(N^2)$ work introduces a significant performance penalty, even for very large matrices, and a few thousand nodes. For submatrices of a size of about 600×600, about one third of the time is spent on $O(N^2)$ work on the Connection Machine system CM–200 with 2048 nodes. About two thirds of this overhead time is spent in communication with the remainder spent in poorly load–balanced arithmetic.

We also show that for architectures with a very simple communication performance model, and for data layouts from a very large regular family, it is not possible to eliminate the overhead from poorly load–balanced arithmetic. Specifically, we show, that in a communication model in which the time for a spread or reduction is determined only by the amount of data that either leaves or enters a node regardless of the configuration of the nodes, assignment of subarrays to nodes based on a two–dimensional nodal array is optimal both for the factorization and the triangular system solution. We also show that the ratio between the lengths of the axes of the two–dimensional nodal array is proportional to the ratio between the corresponding axes of the data array operated upon, i.e., the matrix subject to factorization, or the set of right hand sides for triangular system solution. This proof does not apply directly to the CM–200, because for hypercubes the available bandwidth for a spread depends upon the node configuration, and the simple communication model does not fully capture the communications capabilities.

Acknowledgment

Many people at Thinking Machines Corp. have provided support for the dense linear algebra work reported here. These include Mark Bromley, Tim Harris, Steve Heller, Robert L. Krawitz, Doug MacDonald, Kapil K. Mathur, Jill P. Mesirov, Luis F. Ortiz, and Leo Unger.

Special thanks go to Hank Dardy at the Naval Research Laboratories, Washington, D.C, who made the Laboratories' Connection Machine system CM–200 available for the benchmarks presented in Tables 7 and 8. Robert Whaley's assistance in this effort is greatly appreciated. Last, but not least, we would like to thank the referees for their careful reading of the manuscript and several helpful comments and suggestions.

References

[1] Edward Anderson, Z. Bai, Christian Bischof, Jim Demmel, Jack Dongarra, Jeremey DuCroz, Anne Greenbaum, Sven Hammarling, A. McKenney, and Danny Sorensen. Preliminary LAPACK users's guide. Technical report, University of Tennesse, July 1991.

[2] Jean-Philippe Brunet and S. Lennart Johnsson. All-to-all broadcast with applications on the Connection Machine. *International Journal of Supercomputer Applications*, 6(3):241–256, 1992.

[3] M.Y. Chan. Embedding of grids into optimal hypercubes. *SIAM J. Computing*, 20(5):834–864, 1991.

[4] Jack J. Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. Technical Report Reprint No. 1, Argonne National Laboratories, Mathematics and Computer Science Division, August 1988.

[5] Jack J. Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms: Model implementation and test programs. Technical Report Reprint No. 2, Argonne National Laboratories, Mathematics and Computer Science Division, August 1988.

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. Technical Report Technical Memorandum 41, Argonne National Laboratories, Mathematics and Computer Science Division, November 1986.

[7] Geoffrey Fox, S. Hiranandani, Kenneth Kennedy, Charles Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Department of Computer Science, Rice University, December 1990.

[8] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors*. Prentice-Hall, 1988.

[9] W. Morven Gentleman. Some complexity results for matrix computations on parallel processors. *J. ACM*, 25(1):112–115, January 1978.

[10] P. Harten, H.-Y. Lin, H Rajic, and D. Wells. BLAS and FFT performance on the Intel i860 microprocessor. Technical report, 1991.

[11] I. Havel and J. Móravek. B-valuations of graphs. *Czech. Math. J.*, 22:338–351, 1972.

[12] Ching-Tien Ho and S. Lennart Johnsson. Embedding meshes in Boolean cubes by graph decomposition. *J. of Parallel and Distributed Computing*, 8(4):325–339, April 1990.

[13] S. Lennart Johnsson. Band matrix systems solvers on ensemble architectures. In *Algorithms,Architecture, and the Future of Scientific Computation*, pages 195–216. University of Texas Press, Austin, TX, 1985.

[14] S. Lennart Johnsson. Dense matrix operations on a torus and a Boolean cube. In *The National Computer Conference*, July 1985.

[15] S. Lennart Johnsson. Fast banded systems solvers for ensemble architectures. Technical Report YALEU/DCS/RR-379, Dept. of Computer Science, Yale University, March 1985.

[16] S. Lennart Johnsson. Solving narrow banded systems on ensemble architectures. *ACM TOMS*, 11(3):271–288, November 1985.

[17] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distributed Computing*, 4(2):133–172, April 1987.

[18] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.

[19] S. Lennart Johnsson and Ching-Tien Ho. Generalized shuffle permutations on Boolean cubes. *J. Parallel and Distributed Computing*, 16(1):1–14, 1992.

[20] S. Lennart Johnsson and Luis F. Ortiz. Local Basic Linear Algebra Subroutines (LBLAS) for distributed memory architectures and languages with an array syntax. *The International Journal of Supercomputer Applications*, 6(4):322–350, 1992.

[21] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM TOMS*, 5(3):308–323, September 1979.

[22] Guangye Li and Thomas F. Coleman. A parallel triangular solver for a distributed memory multiprocessor. *SIAM J. Sci. Statist. Comput.*, 9(3):485–502, 1988.

[23] Guangye Li and Thomas F. Coleman. A new method for solving triangular systems on a distributed memory message-passing multiprocessor. *SIAM J. Sci. Statist. Comput.*, 10(2):382–396, 1989.

[24] Kapil K. Mathur and S. Lennart Johnsson. Multiplication of matrices of arbitrary shape on a Data Parallel Computer. *Parallel Computing*, 20(7):919–951, July 1994.

[25] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms.* Prentice-Hall, Englewood Cliffs. NJ, 1977.

[26] B.T. Smith, J.M Boyle, J.J. Dongarra, B.S. Garbov, Y. Ikebe, V.C. Klema, and C.B. Moler. *Matrix Eigensystem Routines – EISPACK Guide.* Springer Verlag, 1976. Lecture Notes in Computer Science, vol. 6.

[27] Thinking Machines Corp. *CM–200 Technical Summary*, 1991.

[28] Thinking Machines Corp. *CM Fortran optimization notes: slicewise model, version 1.0*, 1991.

[29] Thinking Machines Corp. *CMSSL for CM Fortran, Version 3.1*, 1993.

[30] Robert A. van de Geijn. Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems. Technical report, The University of Texas at Austin, July 1991.

[31] Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and Andreas Schwald. Vienna Fortran – A language specification version 1.1. Technical report, ICASE, Interim Report 21, March 1992.