DESIGN AND PERFORMANCE MODELING OF PARALLEL BLOCK MATRIX FACTORIZATIONS FOR DISTRIBUTED MEMORY MULTICOMPUTERS

KRISTER DACKLAND AND ERIK ELMROTH
INSTITUTE OF INFORMATION PROCESSING
UNIVERSITY OF UMEÅ
S-901 87 UMEÅ, SWEDEN

Abstract. Efficient and scalable parallel block algorithms for the LU factorization with partial pivoting, the *Cholesky*, and QR factorizations in a distributed memory multicomputer environment are presented. The distributed system is viewed as a ring of processors and the algorithms correspond to shared memory algorithms parallelized on block level (explicit parallelism). Performance of the algorithms are analyzed theoretically and illustrated empirically by implementations on the Intel iPSC/2 hypercube. A model predicting performance and optimal block size is presented.

Keywords: Block matrix factorizations, distributed memory multicomputers, performance modeling.

1. Introduction. Block matrix factorizations for uniprocessor and shared memory multiprocessors have been the object for an intensive research during the past few years. The results have laid a ground for efficient implementations of block algorithms for basic matrix computations on different hierarchical memory and shared memory environments (e.g. see [1, 6, 7, 8, 12, 14, 20]). The current development of scalable *distributed memory multicomputers* (DMM) asks for the corresponding basis of efficient block algorithms.

Several research projects have also been focusing on algorithms for matrix factorizations for DMM. For example, in the mid eighties non-block algorithms were discussed in [16, 17, 23, 24]. For more references, see [14]. The development of distributed block algorithms have started more recently (e.g. see [3, 13]).

This paper is a contribution to the design, analysis, and evaluation of distributed block algorithms for some matrix factorizations which are efficient, and scalable in the sense that they preserve their maximal performance (measured in $Mflops$/node) when both the problem size and the number of processors increases. Further, the algorithms are transportable over a range of DMM environments.

More precisely, ring-oriented block algorithms for the LU factorization with partial pivoting, the *Cholesky* and QR factorizations, corresponding to the explicitly parallel shared memory block algorithms in [7], are presented. The algorithms have been implemented in double precision on a 64 nodes Intel iPSC/2 hypercube, viewed as a ring of nodes. In order to obtain portability and high performance, all message passing primitives follows the proposal for BLACS (Basic Linear Algebra Communication Subprograms) [9] and most of the computations are performed by higher level BLAS (Basic Linear Algebra Subprograms) [10, 11, 20, 21, 22].

The performance is theoretically analyzed and the times required for different suboperations are quantified. A model determining the performance and the optimal block size is presented. The results show the impact of important key factors, such as, computation-to-communication ratio, computation-to-waiting ratio, arithmetical complexity of algorithms, arithmetic speedup, and block sizes.

The outline of the rest of the paper is as follows: In Section 2, important properties of distributed block algorithms for matrix factorizations are discussed, and ring-oriented block algorithms for the LU, *Cholesky*, and QR factorizations are presented. Performance results are shown in Section 3 and theoretically analyzed and modeled in Section 4. In Section 5, the performance of the algorithms are discussed and, finally, in Section 6, some concluding remarks are summarized.

2. Distributed Block Matrix Factorizations. In a DMM environment we strive for algorithms that have a high computation-to-communication ratio and a granularity large enough to obtain good uniprocessor performance on each individual processor. The algorithms should also have minimal requirement of extra storage and be scalable, in the sense that they preserve their maximal performance (measured in $Mflops$/node) when both the problem size and the number of processors increase.

It has been shown in [7, 8, 12, 14] that block algorithms, rich in level 3 BLAS operations, can be used to exploit the full potential of many high performance computers in matrix computations. For the matrix factorizations considered (LU, $Cholesky$, and QR) there are different block variants such as the block right, block left looking algorithms, and hybrids of these [1, 7, 12, 14].

A *block right looking* algorithm is essentially performed in two steps. First, a level 2 factorization of a block column followed by an update of the remaining matrix with respect to the factorized block column. The same algorithm is repeatedly applied until the factorization is completed.

In the *block left looking* algorithm the update is performed only on the next block column to be factorized. Consequently, the update must be performed with respect to all previous factorizations and not only to the last one, as in the block right looking algorithm. A major drawback in a distributed environment is that all processors must either store all previous factorized blocks or have the blocks sent to them repeatedly.

The weak points of the block left looking algorithm are more or less inherited to the hybrid algorithms. Therefore, our work has been focused on distributed block right looking algorithms. For detailed algorithm descriptions of other variants, see [7].

In Section 2.1, a high level distributed algorithm for a general block matrix factorization is presented followed by some general remarks on ring-oriented algorithms in Section 2.2. In Section 2.3, a detailed ring-oriented block algorithm for the LU factorization is presented. High level algorithms for the $Cholesky$ and QR factorizations are presented in Section 2.4 - 2.5. For more detailed descriptions, see [5].

2.1. A Distributed Block Right Looking Matrix Factorization. Consider an $m \times n$ matrix A that is *block column wrap-mapped* onto p processors. This means that processor i holds every *i'th* block column of A. Assume that nb is the block size and nbl $(= n/nb)$ is the number of block columns in A. Let *current block* denote the submatrix containing the information needed for the update in each iteration, i.e. the last factorized block column. Let *remaining blocks* denote the blocks on the right hand side of the last factorized block column, and let *current processor* denote the processor that performs the factorization. A general *distributed block right looking* matrix factorization is performed by the following node algorithm executing on all processors:

For $i = 1 : nbl$
  If (I hold block i)
    UPDATE block i w.r.t. *current block* (iteration $i-1$) (1)
    FACTORIZE block i & generate next *current block* (2)
    SEND next *current block* to all processors (3)
  End If
  UPDATE my *remaining blocks* w.r.t. *current block* (factorization $i-1$) (4)
  If (I do not hold block i)
    RECEIVE new *current block* (factor from iteration i) (5)
  End If
End For

While the first current processor produces the first current block, operation (2), the other processors are waiting for RECEIVE (5). When they receive the current block, all processors update their remaining blocks (4) except the next current processor that first updates (1), factorizes (2), and sends the next current block (3) before continuing the update (4). This means that the factorization of the next current block is overlapped by the update of the remaining blocks. In an ideal situation, the next current block is computed before it is needed. In [7], this is referred to as *pipelining between iterations.*

The communication is the only synchronization in the algorithm and it is here assumed to be *asynchronous* in the sense that the sending processor does not have to wait for the receiving processors to perform the reception. The only extra storage required is the storage for the current block, i.e. a matrix sized $m \times nb$.

The update operations (1) and (4) are effected by calls to one or more level 3 BLA subprograms [11]. The subprograms used are `DGEMM` (general matrix multiply and add), `DTRMM` (triangular matrix multiply), `DTRSM` (triangular solve with multiple right hand sides), and `DSYRK` (symmetric rank-k update). The factorization of a block column (2) is performed by a call to a level 2 routine.

2.2. Ring-oriented Block Algorithms. Most DMM environments may be viewed as a ring of processors, each connected to a left and right neighbour. In a unidirected ring, all messages are sent from e.g. the left to the right. According to this model, the current block is forwarded from neighbour to neighbour in the ring until all processors have received it. The next current block is always held by the processor to the right of the current processor.

If the system has a cut-through network, it is possible for the current processor to send the current block to all the other processors with a broadcast operation. Still, the algorithm can be viewed as a ring-oriented algorithm because of the order of the level 2 factorizations. In the rest of the paper the *neighbour to neighbour* strategy is referred to as *ring* and the *broadcast* strategy as *fan out.* In both cases, the implementations of SEND and RECEIVE follow the BLACS proposal for communication primitives [9].

In our implementations a binary reflected Gray code numbering of the nodes is used to assure that adjacent nodes in the logical ring also are physical neighbours.

2.3. Ring-oriented Block Right Looking LU Algorithm. The basic block right looking algorithm for computing a reasonable stable LU factorization is, e.g., presented in [6, 7, 15]. The algorithm computes

$$PA = LU \tag{2.1}$$

where, if A is $m \times n$, the $m \times n$ matrix L is lower trapezoidal, the $n \times n$ matrix U is upper triangular, and P is the permutation matrix corresponding to the row interchanges in A.

We illustrate an iteration at block level by the following matrix equality (permutations omitted):

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & A_{22} - L_{21}U_{12} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & I \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & A_{22} \end{bmatrix}. \tag{2.2}$$

The LU factorization of a block column is expressed by

$$\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11} \tag{2.3}$$

The blocks L_{11} and L_{21} correspond to the current block for the next iteration and are used in the update operation. An update consists of two operations: the block U_{12} is

computed by solving the triangular system $A_{12} = L_{11}U_{12}$ and the remaining matrix is updated by the matrix multiply and add operation $A_{22} = A_{22} - L_{21}U_{12}$. A detailed node algorithm is presented below.

For $i \leftarrow 1 : nbl$
  $s \leftarrow (i-1) \cdot nb + 1$; $e \leftarrow \min(i \cdot nb, m)$ {Global start & end of next current block}
  If $proc = me$ & $s \leq m$
    $sl \leftarrow lbl \cdot nb + 1$; $el \leftarrow \min((lbl+1) \cdot nb, myn)$ {Local start & end of next current block}
    $lbl \leftarrow lbl + 1$
    If $i > 1$
      Apply pivotings to $A_{os:oe,sl:el}$ (1)
      $U_{os:oe,sl:el} \leftarrow C^{-1}_{os:oe,1:nb} A_{os:oe,sl:el}$ {DTRSM} (2)
      $A_{s:m,sl:el} \leftarrow A_{s:m,sl:el} - C_{s:m,1:nb} U_{os:oe,sl:el}$ {DGEMM} (3)
    $A_{s:m,sl:el} = L_{s:m,sl:el} U_{s:e,sl:el}$ {level 2 factorization} (4)
    If $p > 1$
      SEND($L_{s:m,sl:el}, ipiv(s:e)$) (5)
  If $i > 1$
    $ssl \leftarrow lbl \cdot nb + 1$
    If $myn > ssl$
      Apply pivotings to $A_{os:oe,ssl:el}$ (6)
      $U_{os:oe,ssl:myn} \leftarrow C^{-1}_{os:oe,1:nb} A_{os:oe,ssl:myn}$ {DTRSM} (7)
      $A_{s:m,ssl:myn} \leftarrow A_{s:m,ssl:myn} - C_{s:m,1:nb} U_{os:oe,ssl:myn}$ {DGEMM} (8)
  If $s \leq m$
    $nnb \leftarrow \min(n - ((i-1)nb, e - s + 1)$
    If $me = proc$
      $C_{s:m,1:nnb} \leftarrow L_{s:m,sl:el}$ (9)
    Else If
      RECEIVE($C_{s:m,1:nnb}, ipiv(s:e)$) (10)
  $proc \leftarrow$right$(proc, p)$
  $os \leftarrow s$; $oe \leftarrow e$ {Local start & end of current block}

The variables *proc* and *lbl* are initialized to 0, and *nbl*, *nb*, and *myn* to the number of column blocks (or to the number of required iterations, if $m < n$), the block size, and the number of columns processor *me* owns, respectively. The only extra storage required is for the matrix C $(m \times nb)$ holding the current block. The L and U are used for clarity only. In practice, A $(m \times myn)$ is overwritten by L and U with $L's$ unit diagonal implicitly defined.

After a level 2 factorization has occurred (4), all permutations must be applied to the corresponding rows to the left and to the right of the factorized block. In the above algorithm they are applied to the right-hand side only (1), (6). The permutations on the left-hand side are performed by the following algorithm after all computations are completed.

$pe \leftarrow 1$
For $i \leftarrow 2 : nbl$
  If gray(mod$(i-2, p)) = me$ {I own block i-1}
    $pe \leftarrow pe + nb$
  If $pe > 1$
    $s \leftarrow (i-1) \cdot nb + 1$
    $e \leftarrow min(i \cdot nb, min(m, n))$
    Apply pivotings to $A_{s:e,1:pe}$ (11)

The LU factorization algorithm described is for $nb = 1$ similar to the unblocked algorithm in [16] except for different strategies for the applications of the permutations.

2.4. Ring-oriented Block Right Looking Cholesky Algorithm. The basic block right looking algorithm for computing a Cholesky factorization of an $n \times n$ symmetric, positive definite matrix A is, e.g., presented in [7, 15]. It computes

$$A = LL^T, \tag{2.4}$$

where L is an $n \times n$ lower triangular matrix. By replacing U_{11}, U_{12} in (2.2) with L_{11}^T, L_{21}^T we can illustrate a block iteration as follows.

Since A is symmetric and positive definite no permutations are required and only the lower triangular part of A has to be referred. This implies that the factorization of a block column corresponding to (2.3) is divided into two operations. First, the diagonal block A_{11} is factorized by a level 2 algorithm and, second, the next current block L_{21} is computed from $L_{21}L_{11}^T = A_{21}$, accomplished as a level 3 operation. That gives the Cholesky factorization a larger level 3 fraction [15] compared to the LU factorization.

The remaining matrix is updated by the symmetric rank-k update $A_{22} = A_{22} - L_{21}L_{21}^T$. The update is performed with the `GEMM`-based approach described in [21], i.e., all diagonal blocks are updated by the level 3 routine `DSYRK` and all off-diagonal blocks by `DGEMM`.

In a detailed implementation corresponding to the general algorithm in Section 2.1, the only extra storage required is for a matrix sized $n \times nb$ to hold the current block.

2.5. Ring-oriented Block Right Looking QR Algorithm. Any $m \times n$ matrix A can be factorized

$$A = QR \tag{2.5}$$

where Q is orthogonal $m \times m$ and R is upper triangular (trapezoidal) $m \times n$. There are several well-known block algorithms that compute the QR factorization [4]. Here, as in [7], the *compact WY* algorithm is considered [15, 25]. The major operations are level 2 QR factorization, generation of block Householder transformations, and update of the remaining blocks. As before, we illustrate one iteration at the block level. The first block column of

$$\mathrm{A} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \tag{2.6}$$

is factorized

$$\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = Q_{nb} \begin{bmatrix} R_{11} \\ 0 \end{bmatrix}, \tag{2.7}$$

where the product of nb Householder transformations Q_{nb} is not explicitly computed. It has been shown in [25] that Q_{nb} can be expressed as the following rank-nb update of the identity matrix:

$$Q_{nb} = I - YSY^T \tag{2.8}$$

where Y is $m \times nb$ lower trapezoidal and S is $nb \times nb$ upper triangular. The update of the remaining blocks is expressed as

$$Q_{nb}^T \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} = \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} - Y(([\, A_{12}^T \;\; A_{22}^T \,]\, Y)S)^T, \tag{2.9}$$

which involves one triangular matrix multiply operation and two general matrix multiply and add operations.

In a detailed implementation corresponding to the general algorithm in Section 2.1, the only extra storage required is for S and Y that constitutes the current block.

3. Performance Results. Implementations of the distributed block LU, $Cholesky$, and QR algorithms described in Section 2 are made in `FORTRAN 77` for a 64 nodes Intel iPSC/2 hypercube [2].

To implement routines for message passing with interfaces corresponding to the operations `DGESD` and `DGERV` in BLACS [9], the iPSC/2 system routines `CSEND` and `CRECV` [18] are used. Tests were run for both the ring and the fan out strategies. The difference in performance never exceeded 5 percentages, and fan out was faster overall. Because of the similarity in performance, results for only the fan out strategy are presented.

For a given problem size the the minimum number of nodes that can solve the problem is determined by the storage available (at most 4 Mbyte on each node including the code). For example, the minimum number is $1, 4, 16$ for $n = 500, 1000, 2000$, respectively.

The execution time $T(p)$ on p nodes is measured in seconds using the iPSC/2 system routine `mclock` [18]. On each node the time is measured for the factorization only (i.e., no time for the host-to-node or node-to-host communications is included) and $T(p)$ is taken as the maximum over the p nodes. The performance measured in $Mflops$ is computed as $(\#\ flops/T(p))\cdot 10^{-6}$, where the $\#\ flops$ are from Table 3.1. The *generalized speedup* $S_G(p)$ [26] is computed as

$$S_G(p) = \frac{Mflops(p)}{Max\_Mflops(1)} \tag{3.1}$$

where $Max\_Mflops(1)$ is the maximal measured performance in $Mflops$ on one processor for any matrix size (which here is the largest matrix that can be factorized on one processor). The main reason for using generalized speedup instead of *traditional speedup*, $S(p) = T(1)/T(p)$, is that it makes it possible to compare the performance for large problems solved on many processors with uniprocessor performance.

The *generalized efficiency* $E_G(p)$ is computed as

$$E_G(p) = \frac{S_G(p)}{p}. \tag{3.2}$$

Notably, this gives lower efficiency numbers than the ones presented in [5] computed as $E(p) = (T(p_{min})/T(p))/(p/p_{min})$, where p_{min} is the minimum number of processors that can solve the problem. However, the generalized efficiency enables measurement of the processor utilization for large problems solved on many processors compared to uniprocessor execution.

TABLE 3.1
Number of floating point operations in the LU, Cholesky, and QR factorizations.

Routine	Number of floating point operations
LU	$mn^2 - \frac{n^3}{3} - \frac{n^2}{2} + \frac{5n}{6}$
$Cholesky$	$\frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$
QR $(m \geq n)$	$2mn^2 - \frac{2n^3}{3} + mn + n^2 + \frac{14n}{3}$
$(m \leq n)$	$2nm^2 - \frac{2m^3}{3} + 3nm - m^2 + \frac{14n}{3}$

3.1. Performance Results for the LU, Cholesky, and QR factorizations. In Tables 3.2, 3.3, and 3.4, performance results are shown for the ring-oriented block right looking LU, $Cholesky$, and QR factorizations, respectively. Results for three different matrix sizes ($n = 500, 1000$, and 2000) are presented for each of the routines. The performance of the level 3 BLAS used are presented in [5].

TABLE 3.2
Results for the LU factorization of square matrices.

n	p	nb	$T(p)$ $(secs)$	$Mflops$	$Mflops/p$	$S_G(p)$	$E_G(p)$
500	1	16	254.0	0.3276	0.3276	1.000	1.000
	2	8	127.8	0.6511	0.3256	1.990	0.9939
	4	4	65.77	1.265	0.3163	3.860	0.9655
	8	4	35.05	2.374	0.2968	7.250	0.9060
	16	2	19.37	4.296	0.2685	13.11	0.8196
	32	2	11.48	7.245	0.2264	22.11	0.6911
	64	2	7.603	10.94	0.1709	33.41	0.5217
1000	4	4	515.9	1.291	0.3228	3.941	0.9853
	8	4	265.7	2.508	0.3135	7.656	0.9570
	16	2	140.0	4.758	0.2974	14.52	0.9078
	32	2	76.48	8.710	0.2722	26.59	0.8309
	64	2	44.75	14.89	0.2327	45.45	0.7103
2000	16	2	1075.	4.959	0.3099	15.14	0.9460
	32	2	562.0	9.487	0.2965	28.96	0.9051
	64	2	304.7	17.50	0.2734	53.42	0.8346

TABLE 3.3
Results for the Cholesky factorization.

n	p	nb	$T(p)$ $(secs)$	$Mflops$	$Mflops/p$	$S_G(p)$	$E_G(p)$
500	1	16	129.0	0.3240	0.3240	1.000	1.000
	2	16	65.60	0.6371	0.3186	1.970	0.9833
	4	8	33.50	1.248	0.3120	3.850	0.9630
	8	4	17.60	2.375	0.2969	7.330	0.9164
	16	4	9.820	4.256	0.2660	13.14	0.8210
	32	2	6.600	6.332	0.1979	19.55	0.6108
	64	2	5.830	7.168	0.1120	22.13	0.3457
1000	4	8	259.0	1.289	0.3223	3.978	0.9948
	8	8	132.0	2.529	0.3161	7.806	0.9756
	16	4	69.10	4.831	0.3019	14.91	0.9318
	32	2	39.80	8.388	0.2621	25.89	0.8090
	64	2	26.10	12.79	0.1998	39.47	0.6167
2000	16	4	528.0	5.054	0.3159	15.60	0.9750
	32	4	275.0	9.704	0.3033	29.95	0.9361
	64	2	160.0	16.68	0.2606	51.48	0.8043

TABLE 3.4
Results for the QR factorization of square matrices.

n	p	nb	$T(p)$ $(secs)$	$Mflops$	$Mflops/p$	$S_G(p)$	$E_G(p)$
500	1	4	428.6	0.3900	0.3900	1.000	1.000
	2	2	216.0	0.7741	0.3871	1.980	0.9926
	4	2	109.1	1.532	0.3830	3.930	0.9821
	8	2	55.55	3.009	0.3761	7.720	0.9644
	16	1	28.87	5.790	0.3619	14.85	0.9279
	32	1	15.48	10.80	0.3375	27.69	0.8654
	64	1	10.23	16.34	0.2553	41.91	0.6546
1000	4	2	863.3	1.547	0.3868	3.967	0.9918
	8	2	436.0	3.063	0.3829	7.854	0.9818
	16	2	222.4	6.005	0.3753	15.40	0.9623
	32	1	114.6	11.65	0.3641	29.87	0.9336
	64	1	61.51	21.71	0.3392	55.67	0.8697
2000	16	2	1751.	6.096	0.3810	15.63	0.9769
	32	1	888.4	12.02	0.3756	30.82	0.9631
	64	1	457.9	23.31	0.3642	59.77	0.9338

For the LU factorization on a given number of processors, Table 3.2 shows that the generalized efficiency increases with increased matrix size. Further, it is possible to obtain a generalized efficiency over 90 percentages for 8, 16, and 32 processors and matrix sizes 500, 1000, and 2000, respectively. The optimal block size nb decreases by increasing number of processors.

The performance in $Mflops$ for the $Cholesky$ factorization is slightly lower compared to the LU factorization. However, the generalized efficiency is approximately the same for the two factorizations, as long as the number of processors is small compared to the matrix size (e.g. $p \leq 16$ for $n = 1000$). For larger number of processors the generalized efficiency decreases faster for the $Cholesky$ factorization than for the LU. Table 3.3 also shows that the optimal block size nb is larger than for the LU factorization.

Table 3.4 shows that the QR factorization has the best performance in $Mflops$ of the three factorization routines presented. It also has the best generalized efficiency, e.g. over 93 percentage on a 2000×2000 matrix on 64 nodes, giving a generalized speedup close to 60, even though the routine has the best uniprocessor result. Notably, the optimal block size nb is small, in fact often 1.

4. Performance Modeling. To understand the behaviour of the algorithms and to determine optimal block sizes, we need a model that identifies different sources (parts) that contribute to the total execution time. A general model predicting the execution time $T_{pred}(p)$ using p processors is expressed by

$$T_{pred}(p) = T_a(p) + T_c(p) + T_w(p), \tag{4.1}$$

where $T_a(p)$ is the average arithmetic time, $T_w(p)$ is the average waiting time, including both idle and busy waiting, and $T_c(p)$ is the average communication time. The motivation to use average times is that for sufficiently large matrices and optimal block sizes, the load balance, inherited from the matrix distribution, gives the processors approximately the same amount of work. Here, only the fan out communication strategy will be considered.

4.1. Arithmetic Time. The average arithmetic time $T_a(p)$ is expressed as:

$$T_a(p) = \frac{\mathcal{K}(nb, m) \cdot \#flops \cdot t_a}{S_a(p, nb)}. \tag{4.2}$$

As before, $\#flops$ is the number of floating point operations in Table 3.1 and t_a is the time required for one floating point operation. For iPSC/2, t_a is $1.67\mu sec$ [2].

Due to the memory hierarchy and software overhead, the arithmetic time on one processor vary for different matrix and block sizes. The function $\mathcal{K}(nb, m)$ models this variation in performance. It is determined from a least squares fit of the timing results on one processor for $m = n = 100, 200, \ldots, 500$ and $nb = 1, 2, \ldots, 20$ to the non-linear function:

$$(f_1 + f_2 \cdot nb)e^{f_3 + f_4/nb}, \tag{4.3}$$

where f_1, f_2 and f_4 are linear functions in $1/m$ and f_3 is exponential:

$$f_3 = e^{c_1 + c_2/m}. \tag{4.4}$$

In Figure 4.1 the real execution time $T(1)$ and the modeled arithmetic time $T_a(1)$ are illustrated for the QR factorization of a 500×500 matrix (largest matrix that can be factorized on 1 processor). We conclude that the optimal block size for one processor is $nb = 3$ ($nb = 2, 4, 5$ give very similar results).

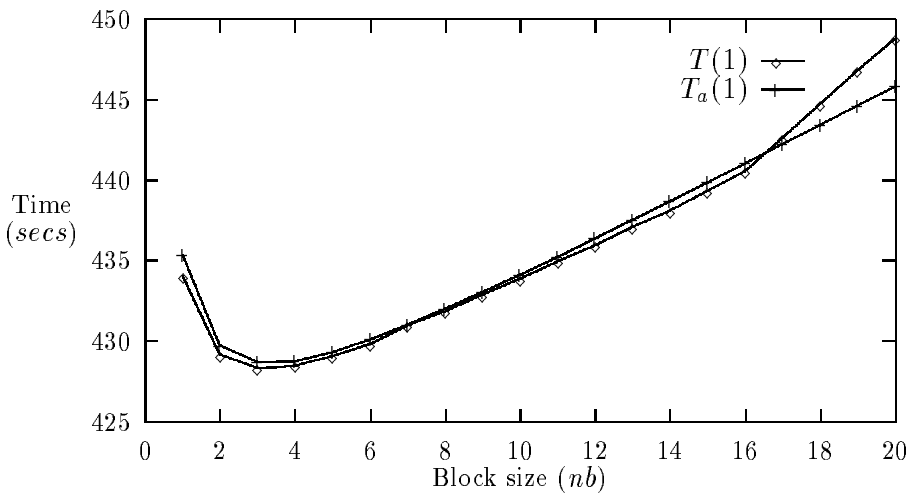

FIG. 4.1. *$T_a(1)$ and $T(1)$ for the QR factorization ($m = n = 500$ and $nb = 1, \ldots, 20$).*

To approximate the performance on p processors $\mathcal{K}(nb, m)$ is divided with the arithmetic speedup $S_a(p, nb)$ (no communication or waiting). Because of a nonlinear behaviour of the speedup in the arithmetic performance, $S_a(p, nb)$ is computed as p (the linear speedup) multiplied by a function $(1 + f(p - 1))$, where $1/(1 - p_{max}) < f \leq 0$ and $p_{max} = 64$ for the interval of current interest. It turns out that also the arithmetic speedup depends on the block size, making f a function in nb. The function f is determined from a least squares fit of a polynomial to the results for a matrix of size 500×500 and $p = 1, 2, \ldots, 64$ processors using $nb = 1, 2, 4, 8, 16$. In Figure 4.2 the predicted arithmetic time computed both by using $S_a(p, nb)$ and by assuming linear speedup are illustrated and compared to the real parallel arithmetic time on 16 processors for the QR factorization. Notably, the different best block size for $T_a(16)$ ($nb = 1$) compared to $T_a(1)$ ($nb = 2, 4$) is detected only because $S_a(p, nb)$ is dependent of the block size.

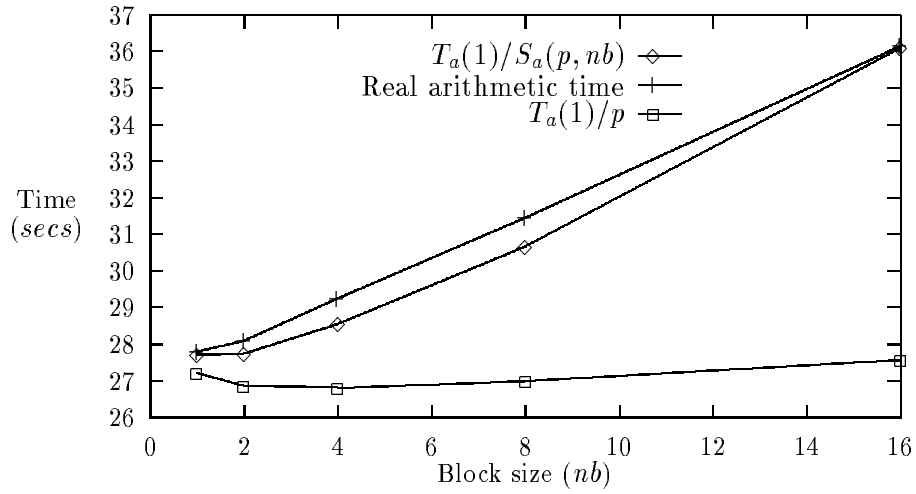

FIG. 4.2. *Real and predicted average arithmetic time per processor for $p = 16$ determined as $T_a(1)/S_a(p, nb)$ and $T_a(1)/p$ for the QR factorization ($m = n = 500$, and $nb = 1, 2, 4, 8, 16$).*

We assume that the arithmetic speedup on a square 500 matrix (the largest matrix that can be factorized on one processor) is representative for other matrix sizes. The functions $\mathcal{K}(nb, m)$ and $S_a(p, nb)$ are determined similarly for the three algorithms.

4.2. Communication Time. The communication time for a given problem is expressed as a function $T_c(p)$, depending on the number of blocks to be sent, the size of the blocks, and the number of processors. For clarity, m is assumed to be equal to n and to be a multiple of nb, in the formulas of this section. The number of blocks and their size may be expressed in terms of n and nb.

A summary of improved linear models of the communication times for the iPSC/2 is found in Table 4.1 [19]. Only buffered receive [2] is considered.

TABLE 4.1
Communication time for iPSC/2 in μsecs.

Size (bytes)	Send time	Reception time
$b = 8 - 96$	$221 + 0.221b$	$342 + 0.250b$
$b = 104\ldots$	$429 + 0.361b$	$321 + 0.105b$

In the general algorithm in Section 2.1, the current block is sent from one processor to all the others in every iteration. In the fan out communication strategy, one message is sent and $p - 1$ messages are received in every iteration. Assuming that the current block always consists of at least 12 double precision elements, the linear model in the last row of Table 4.1 is used to compute the time for sending and receiving the current block. Further, there are $nbl = n/nb$ block iterations where one block is fanned out.

In the LU factorization, the current block and a part of the pivot vector are sent in each iteration. The average communication time for a complete factorization is therefore expressed by:

$$T_c(p) = \frac{S_{cb} + R_{cb} + S_{ipiv} + R_{ipiv}}{p}. \tag{4.5}$$

where e.g.

$$S_{cb} = \frac{n}{nb}429 + \sum_{i=0}^{\frac{n}{nb}-1} 8(n - i \cdot nb)nb0.361, \tag{4.6}$$

Here S_{cb} denotes the time to send all current blocks and R_{cb} denotes the time required for $p-1$ processors to receive all current blocks. The times S_{ipiv} and R_{ipiv} are the corresponding communication times for the pivot vectors. The other partial sums are expressed similarly as for S_{cb}. If the number of double precision elements to be sent from the vector *ipiv* in each iteration is less than 12, the first row in Table 4.1 is used to determine S_{ipiv} and R_{ipiv}. The modeled times S_{cb}, R_{cb}, S_{ipiv}, and R_{ipiv} for the *LU* factorization can be found in Table 4.2.

TABLE 4.2
Communication time in μsecs for the LU factorization of an $n \times n$ matrix.

Operation	Time ($\mu secs$)
Send *cb*	$\frac{n}{nb}429 + 4(n^2 + n \cdot nb)0.361$
Receive *cb*	$(p-1)(\frac{n}{nb}321 + 4(n^2 + n \cdot nb)0.105)$
Send *ipiv*	$\frac{n}{nb}221 + 8n0.221$
Receive *ipiv*	$(p-1)(\frac{n}{nb}342 + 8n0.250)$

In the *Cholesky* factorization, only the current block is sent in each iteration, except for the last one where no communication is needed. Moreover, the size of the current block in iteration i is $(n-(i+1)nb)nb$ while it is $(n-i \cdot nb)nb$ for the *LU* factorization. In Table 4.3, S_{cb} and R_{cb} for the *Cholesky* factorization are presented.

TABLE 4.3
Communication time in μsecs for the Cholesky factorization of an $n \times n$ matrix.

Operation	Time ($\mu secs$)
Send *cb*	$(\frac{n}{nb} - 1)429 + 4(n^2 - n \cdot nb)0.361$
Receive *cb*	$(p-1)((\frac{n}{nb} - 1)321 + 4(n^2 - n \cdot nb)0.105)$

The times required to send and receive current block and an $nb \times nb$ triangular matrix S in every iteration in the QR factorization are summarized in Table 4.4. The message passing times for the matrix S are those in the first row in Table 4.1. This is correct for $nb \leq 3$, which in Table 3.4 was shown to be the best choice for the QR factorization.

TABLE 4.4
Communication time in μsecs for the QR factorization of an $n \times n$ matrix.

Operation	Time ($\mu secs$)
Send *cb*	$\frac{n}{nb}429 + 4(n^2 + n \cdot nb)0.361$
Receive *cb*	$(p-1)(\frac{n}{nb}321 + 4(n^2 + n \cdot nb)0.105)$
Send S	$\frac{n}{nb}221 + 8n \cdot nb0.221$
Receive S	$(p-1)(\frac{n}{nb}342 + 8n \cdot nb0.250)$

The communication time for the ring strategy is computed similarly as for the fan out. The only difference is that each send in the fan out strategy has to be done $p-1$ times instead of one.

4.3. Waiting Time. The average waiting time $T_w(p)$ can be separated into two parts; *busy waiting* time $T_{bw}(p)$ and *idle waiting* time $T_{iw}(p)$. $T_{bw}(p)$ comprises the time needed for the first processor to compute and distribute the first current block, and the time when different processors are waiting for messages to arrive. $T_{iw}(p)$ comprises mainly the time some processors are idle at the end of the computations (mainly due to imbalance of the load).

Here, $T_w(p)$ is computed in the following steps. The initial waiting time when all other processors wait for the first current processor to factorize and distribute the first current block is determined as the time required for the factorization. The waiting time in all other block iterations is approximated as the possible extra time the current processor needs to produce the next current block compared to the time the other processors needs for the update corresponding to the last factorization. This gives the waiting time that would arise in each iteration if all processors started the update corresponding to the last current block at exactly the same time. In the presented algorithms this will not always be the case, and therefore, the waiting time is expected to be overestimated for large block sizes.

4.4. Theoretical Results and Choice of Optimal Block Sizes. The total predicted execution time

$$T_{pred}(p) = T_a(p) + T_c(p) + T_w(p) \tag{4.7}$$

is presented in Table 4.5 for the LU, $Cholesky$, and QR factorizations of a matrix sized 1000×1000. The block sizes used for the modeled results are selected as in Tables 3.2 - 3.4, i.e., the best in real execution.

TABLE 4.5
Modeled times in secs for the LU, Cholesky, and QR factorizations of a 1000×1000 *matrix.*

Routine	p	nb	$T_a(p)$	$T_c(p)$	$T_w(p)$	$T_{pred}(p)$
LU	4	4	514.7	0.846	0.047	515.6
	8	4	262.4	0.718	0.086	263.0
	16	2	133.5	0.818	0.034	134.3
	32	2	69.72	0.786	0.105	70.62
	64	2	38.28	0.770	0.420	39.42
Cholesky	4	8	256.0	0.714	0.401	257.2
	8	8	130.0	0.585	1.381	132.0
	16	4	67.23	0.564	0.718	69.10
	32	2	39.08	0.613	0.418	40.11
	64	2	23.03	0.596	1.944	25.60
QR	4	2	858.3	1.011	0.030	859.3
	8	2	432.9	0.884	0.049	433.8
	16	2	220.3	0.820	0.022	221.2
	32	1	112.2	1.117	0.056	113.4
	64	1	58.41	1.101	0.198	59.70

Note that the arithmetic time $T_a(p)$, for QR is over 1.5 times that of LU, and that $T_a(p)$ for LU is over 1.7 that of $Cholesky$. It is clear that the communication time and the waiting time are negligible for small number of processors and optimal block sizes. For the $Cholesky$ factorization they are considerable when the number of processors increases to 64. It is clear that higher arithmetic complexity (as in QR) results in higher computation-to-communication and computation-to-waiting ratios.

In Table 4.6 predicted timing results for the $Cholesky$ factorization are presented for different parts of the computation, i.e. $T_a(p), T_c(p)$, etc. As expected, the communication time decreases and the waiting time increases with increased block sizes. For the arithmetic time the best block size shows to be smaller for an increased number of processors. The choice of optimal block size is therefore a trade off between good

TABLE 4.6

Real and predicted execution times in secs for the Cholesky factorization of a 1000×1000 *matrix and different block sizes.*

Routine	p	Block size (nb)				
Cholesky		1	2	4	8	16
$T_a(p)$	4	314.1	275.9	260.3	**256.0**	260.0
	8	163.1	140.3	131.6	**130.0**	134.2
	16	88.34	72.64	**67.23**	67.12	71.78
	32	53.02	39.07	**35.17**	35.89	-
	64	44.21	**23.03**	19.37	-	-
$T_c(p)$	4	1.023	0.848	0.760	**0.714**	0.687
	8	0.882	0.714	0.629	**0.585**	0.560
	16	0.811	0.647	**0.564**	0.521	0.497
	32	0.776	0.613	**0.531**	0.489	-
	64	0.758	**0.596**	0.515	-	-
$T_w(p)$	4	0.003	0.014	0.070	**0.401**	2.650
	8	0.006	0.032	0.198	**1.381**	10.65
	16	0.018	0.104	**0.718**	5.531	46.43
	32	0.074	0.418	**2.946**	23.80	-
	64	0.472	**1.944**	13.00	-	-
$T_{pred}(p)$	4	315.1	276.8	261.1	**257.1**	263.3
	8	164.0	141.1	132.4	**132.0**	145.4
	16	89.17	73.39	**68.51**	73.17	118.7
	32	53.87	40.11	**38.64**	60.18	-
	64	45.44	**25.58**	32.89	-	-
$T(p)$	4	303.0	272.0	263.0	**259.0**	266.0
	8	159.0	139.0	134.0	**132.0**	139.0
	16	87.1	72.5	**69.1**	70.9	91.2
	32	51.1	**39.8**	40.3	49.9	-
	64	33.3	**26.1**	30.9	-	-

arithmetic performance and short communication and waiting times. The best timing results in Table 4.6 for $T(p)$ and $T_{pred}(p)$ are highlighted in bold.

Overall the predicted times agree with the experimental results for reasonable block sizes. Predicted timing results for other matrix sizes and for the LU and QR factorizations have also shown the same agreement with the experimental results. The model that computes the predicted execution time, also predicts the best block size to be the same as the best one in real execution, except occasionally, when it predicts a nearby block size which in real execution shows a performance close to the optimal.

5. Discussion. The results in Tables 3.2 - 3.4 show that QR retains good performance longer than the LU and *Cholesky* factorizations for increasing number of processors. It is also clear that the LU factorization shows the same relationship to the *Cholesky* factorization.

These differences are mainly inherited from the differences in $T_a(p)$ for the three routines when the number of processors increases, i.e. the single most important explanation for these relationships is the difference in arithmetic speedup $S_a(p)$. The differences are reinforced by the differences in computation-to-waiting ratio $T_a(p)/T_w(p)$. Although $T_a(p)$ in the QR factorization is over 1.5 times that of the LU factorization, $T_w(p)$ is smaller for QR than for LU. This implies that the relative waiting time is

much smaller in the QR factorization, leading to a larger speedup. The small $T_w(p)$ for QR is explained by the small optimal block size, implying good load balance.

The difference between the LU and $Cholesky$ factorizations is similar to the one between the QR and LU factorizations and can also be explained with the same reasoning.

Notice that the LU factorization does not always show a performance between $Cholesky$ and QR, even though the arithmetic complexity would suggest so. The main reason is the extra work required for pivotings. It has been shown in [7] that pivotings, that are not counted as flops, may consume a considerable amount of time.

All three algorithms are scalable in the sense that the maximal performance per processor is almost constant if the matrix is large enough, see Tables 3.2 - 3.4.

6. Conclusions. Parallel *block right looking* algorithms for LU, $Cholesky$, and QR factorization designed for a distributed memory multicomputer system and a model that predicts the execution time and determines the optimal block sizes have been presented. The algorithms correspond to those for shared memory, described as explicitly parallel in [7]. The idea to view both the shared and distributed memory machines as a ring of processors and to modify the algorithms to fit the distributed environment has shown to give highly efficient and scalable algorithms.

In the implementations, most of the computations were performed by level 3 BLAS (Basic Linear Algebra Subprograms) [11] which, together with the use of BLACS (Basic Linear Algebra Communication Subprograms) [9] for the communication, give portability, readability, and efficiency.

Performance results on the Intel iPSC/2 show that all three algorithms have a generalized efficiency over 90 percentage for sufficiently large problems, i.e. matrix size 2000 and $p \leq 32$ for LU and $Cholesky$ factorizations and $p \leq 64$ for QR factorization. The best block sizes are very small overall, for all the algorithms, which in some cases have turned into level 2 algorithms. This implies that the implicit load balancing through the matrix partitioning works well.

By analyzing the algorithms and modeling the performance, the time for arithmetic, communication, and the processors being waiting, have been distinguished. For the parallel performance, the arithmetic speedup has shown to be the single most important factor. For large block sizes the waiting time also becomes significant. Overall the communication consumes a relatively small amount of time.

Increased arithmetic complexity results in higher computation-to-communication and computation-to-waiting ratios and, therefore, improves the performance. This is explained by a high arithmetic speedup and by high computation-to-communication and computation-to-waiting ratios. The small communication times compared to the total times for the algorithms imply small differences in performance between the ring and the fan out communication strategies.

The performance model has shown to predict the optimal block size or a near to optimal block size corresponding to a real execution time very close to the best one. It has also shown to predict the execution time close to the real execution time, especially for the best block sizes and for problem sizes suitable for a given number of processors.

Acknowledgements. We would like to thank our colleagues in the Parallel Computing Group at the University of Umeå for their support and especially our supervisor Bo Kågström for helpful discussions and constructive comments during the work and the preparation of this paper.

This work was partly supported by the Swedish Board of Technical Development under grant STU 89-02578P.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, *LAPACK Users' Guide*, Siam Publications, (1992).

[2] S. Arshi, R. Asbury, J. Brandenburg, and D. Scott "Application Performance Improvement on the iPSC/2 Computer", Concurrent Supercomputing, The second Generation, A technical Summary of the iPSC/2 Concurrent Supercomputer, Intel, (1988), pp 17-22.

[3] C. Bischof, "Adaptive Blocking in the QR Factorization", *The Journal of Supercomputing*, No. 3, Vol. 3, Kluwer Academic Publishers, (1989), pp 193-208.

[4] C. Bischof and C. Van Loan, "The WY Representation for Products of Householder Matrices", *SIAM J. Scientific and Statistical Computing*, Vol. 8, (1987), pp s2-s13.

[5] K. Dackland and E. Elmroth, "Parallel Block Matrix Factorizations for Distributed Memory Multicomputers", Report UMINF-92.03, Inst. of Information Processing, Univ. of Umeå, S-901 87 Umeå, May 1992.

[6] K. Dackland, E. Elmroth, B. Kågström, and C. Van Loan, "Design and Evaluation of Parallel Block Algorithms: LU Factorization on an IBM 3090 VF/600J", in D. Sorensen et al (eds.), *Parallel Processing for Scientific Computing*, SIAM Publications, (1991) (to appear).

[7] K. Dackland, E. Elmroth, B. Kågström, and C. Van Loan, "Parallel Block Matrix Factorizations on the Shared Memory Multiprocessor IBM 3090 VF/600J", *International Journal of Supercomputer Applications*, Vol 6.1, MIT Press (1992), pp 69-97

[8] M. Daydé and I. Duff, "Use of Level 3 BLAS in LU Factorization in a Multiprocessing Environment on Three Vector Multiprocessors: the ALLIANT FX/80, the CRAY-2, and the IBM 3090 VF", Tech. Report CERFACS, August 1990.

[9] J. Dongarra, "Workshop on the BLACS", Tech. Report CS-91-134, Univ. of Tennessee, Knoxville, May 1991. (LAPACK Working Note #34)

[10] J. Dongarra, J. Du Croz, S. Hammarling and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms", *ACM Trans. on Mathematical Software*, Vol. 14 (1988) pp 1-17, 18-32.

[11] J. Dongarra, J. Du Croz, S. Hammarling and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms", *ACM Trans. on Mathematical Software*, Vol. 16, (1990), pp 1-17, 18-28.

[12] J. Dongarra, I. Duff, D. Sorensen and H. Van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM Publications, (1991).

[13] J. Dongarra and S. Ostrouchov, "LAPACK Block Factorization Algorithms on the Intel iPSC/860", Univ. of Tennessee, Knoxville, October 1990. (LAPACK Working Note #24).

[14] K. Gallivan, R. Plemmons and A. Sameh, "Parallel Algorithms for Dense Linear Algebra Computations", *SIAM Review*, Vol. 32 (1990), pp 54-135.

[15] G. Golub and C. Van Loan, *Matrix Computations*, John Hopkins Press, 2nd edition, (1989).

[16] A. Geist and C. Romine, "LU factorization Algorithms on Distributed-Memory Multiprocessor Architectures", *SIAM J. Sci. Statist. Comput.*, 9(4), (1988), pp 639-649.

[17] M. Heath, "Parallel Cholesky factorization in message-passing multiprocessor environments", Tech. Report ORNL-6150, Oak Ridge National Laboratory, May 1985.

[18] Intel Corporation, "iPSC/2 and iPSC/860, Programmer's Reference Manual", June 1990.

[19] P. Jacobson, "Detailed Communication Benchmarks and Models for Intel iPSC/2 and iPSC/860 Hypercubes", Tech. Report UMINF-92.05, Inst. of Information Processing, Univ. of Umeå, S-901 87 Umeå, 1992.

[20] B. Kågström P. Ling, and C. Van Loan, "High Performance GEMM-Based Level-3 BLAS: Sample Routines for Double Precision Real Data", in M. Durand and F. El Dabaghi (eds.), *High Performance Computing II*, Elsevier Science Publishers B.V. (North-Holland), (1991).

[21] B. Kågström and C. Van Loan, "GEMM-Based Level-3 BLAS", Tech. Report CTC91TR47, Cornell University, December 1989.

[22] C. Lawson, R. Hanson, R. Kincaid and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage", *ACM Trans. on Mathematical Software*, Vol. 5 (1979), pp 308-323.

[23] C. Moler, "Matrix Computation on Distributed Memory Multiprocessors", in M. Heath (ed.), *Hypercube Multiprocessors*, SIAM (1986), pp 181-195.

[24] A. Pothen and P. Raghavan, "Distributed Orthogonal Factorization: Givens and Householder Algorithms", Tech. Report CS-87-24, Pennsylvania State Univ., 1987.

[25] R. Schreiber and C. Van Loan, "A Storage Efficient WY Representation for Products of Householder Transformations", *SIAM J. Scientific and Statistical Computing*, Vol. 10 (1989), pp 53-57.

[26] X.-H. Sun and J. L. Gustafson, "Toward a Better Parallel Performance Metric", *Parallel Computing*, 17 (1991), pp 1093-1109.