PoLAPACK: Parallel Factorization Routines with Algorithmic Blocking

Jaeyoung Choi
School of Computing
Soongsil University
1-1, Sangdo-Dong, Dongjak-Ku
Seoul 156-743, KOREA
choi@comp.soongsil.ac.kr

Abstract

LU, QR, and Cholesky factorizations are the most widely used methods for solving dense linear systems of equations, and have been extensively studied and implemented on vector and parallel computers. Most of these factorization routines are implemented with block-partitioned algorithms in order to perform matrix-matrix operations, that is, to obtain the highest performance by maximizing reuse of data in the upper levels of memory, such as cache. Since parallel computers have different performance ratios of computation and communication, the optimal computational block sizes are different from one another to generate the maximum performance of an algorithm. Therefore, the data matrix should be distributed with the machine specific optimal block size before the computation. Too small or large a block size makes getting good performance on a machine nearly impossible. In such a case, getting a better performance may require a complete redistribution of the data matrix.

In this paper, we present parallel LU, QR, and Cholesky factorization routines with an "algorithmic blocking" on 2-dimensional block cyclic data distribution. With the algorithmic blocking, it is possible to obtain the near optimal performance irrespective of the physical block size. The routines are implemented on the Intel Paragon and the SGI/Cray T3E and compared with the corresponding ScaLAPACK factorization routines.

1. Introduction

In many linear algebra algorithms the distribution of work may become uneven as the algorithm proceeds, for example as in parallel LU factorization algorithm [?, ?], in which rows and columns are successively eliminated from the computation. The way in which a matrix is distributed over the processors has a major impact on the load balance and communication characteristics of a parallel algorithm, and hence largely determines its performance and scalability.

The two-dimensional block cyclic data distribution [?, ?], in which matrix blocks separated by a fixed stride in the row and column directions are assigned to the same processor, has been used as a general purpose basic data distribution for parallel linear algebra software libraries because of its scalability and load balance properties. And most of the parallel version of algorithms have been implemented on the two-dimensional block cyclic data distribution [?, ?].

Since parallel computers have different performance ratios of computation and communication, the optimal computational block sizes are different from one another to generate the maximum performance of an algorithm. The data matrix should be distributed with the machine specific optimal block size before the computation. Too small or large a block size makes getting good performance on a machine nearly impossible. In such case, getting a better performance may require a complete redistribution of the data matrix.

The matrix multiplication, $\mathbf{C} \Leftarrow \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$, might be the most fundamental operation in linear algebra. Several parallel matrix multiplication algorithms have been proposed on the two-dimensional block-cyclic data distribution [?, ?, ?, ?, ?]. High performance, scalability, and simplicity of the parallel matrix multiplication schemes using rank-K updates has been demonstrated [?, ?]. It is assumed that the data matrices are distributed on the two-dimensional block cyclic data distribution and the column block size of $\mathbf{A}$ and the row block size of $\mathbf{B}$ are K. However getting a good performance when the block size is very small or very large is difficult, since the computation are not effectively overlapped with the communication. The LCM (Least Common Multiple) concept [?] has been introduced to DIMMA [?] to use a computationally optimal block size irrespective of the physically distributed block size for the parallel matrix multiplication. In DIMMA, if the physical block size is smaller than the optimal block size, the small blocks are combined into a larger block. And if the physical block size is larger than the optimal block size, the block is divided into smaller pieces. This is the "algorithmic blocking" strategy.

There have been several efforts to develop parallel factorization algorithms with the algorithmic blocking on distributed-memory concurrent computers. Lichtenstein and Johnsson [?] developed and implemented block-cyclic order elimination algorithms for LU and QR factorization on the Connection Machine CM-200. They used a cyclic order elimination on a block data distribution, the only scheme that the Connection Machine system compilers supported.

P. Bangalore [?] has tried to develop a data distribution-independent LU factorization algorithm. He recomposed computational panels to obtain a computationally optimal block size, but followed the original matrix ordering. According to the results, the performance is superior to the other case, in which the matrix is redistributed when the block size is very small. He used a tree-type communication scheme to make computational panels from several columns of processors. However, using a pipelined communication scheme, if possible, which overlaps communication and computation effectively, would be more efficient.

The actual algorithm which is selected at runtime depending on input data and machine parameters is called "polyalgorithms" [?]. We are developing "PoLAPACK" (Poly LAPACK) factorization routines, in which computers select the optimal block size at run time according to machine characteristics and size of data matrix. In this paper, we expanded and generalized the idea in [?]. We developed and implemented parallel LU, QR, and Cholesky factorization routines with the algorithmic blocking on the 2-dimensional block cyclic data distribution. With PoLAPACK, it is always possible to have the near optimal performance of LU, QR, and Cholesky factorization routines on distributed-memory computers irrespective of the physical data-distribution on distributed-memory concurrent computers if all of the processors have the same size of submatrices.

The PoLAPACK factorization routines are implemented based on ScaLAPACK, but the internals are very different. ScaLAPACK uses global parameters, but PoLAPACK uses both global and local parameters because of the computational complexity to compute indices, which represent the current row and column of processors, and the global size of the matrix as well as local sizes of submatrices in each processor to be computed. Currently the PoLAPACK factorization routines are implemented based on the block cyclic data distribution, but it is also possible to apply the idea to other decompositions.

The PoLAPACK LU, QR, and Cholesky factorization routines are implemented on the Intel Paragon computer at Samsung Advanced Institute of Technology, and on the Cray T3E at KORDIC Supercomputing Center, Korea. And their performance is compared with that of the corresponding ScaLAPACK factorization routines.

2. PoLAPACK LU Factorization Algorithm

The basic LU factorization routine is to find the solution vector x after applying LU factorization to A from the following linear equation

$$Ax = b.$$

After converting A to $P \cdot A = L \cdot U$, compute y from $Ly = b_0$, where $U \cdot x = y$ and $P \cdot b = b_0$. And compute x.

Most of the LU factorization algorithms including LAPACK [?] and ScaLAPACK [?] find the solution vector x after computing the factorization of $P \cdot A = L \cdot U$. And in the ScaLAPACK factorization routines, a column of processors performs a factorization on its own column of blocks, and broadcasts it to others. Then all of processors update the rest of the data matrix. The basic unit of the computation is the physical size of the block, with which the data matrix is already distributed over processors.

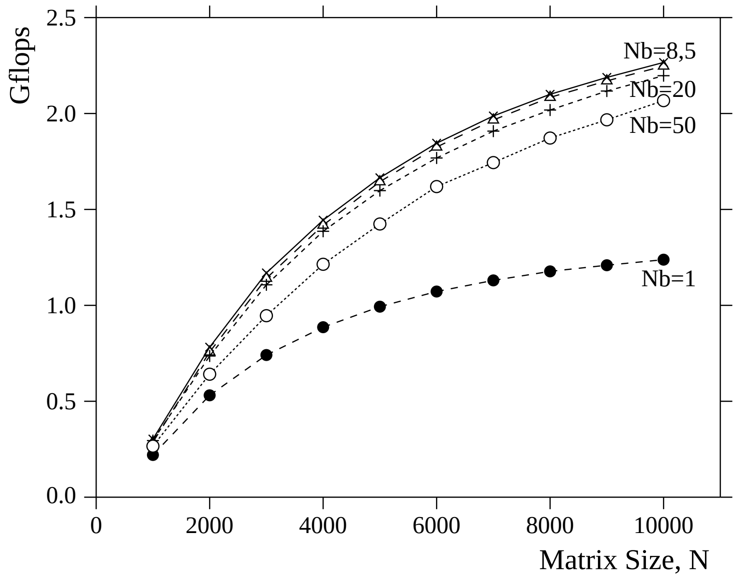
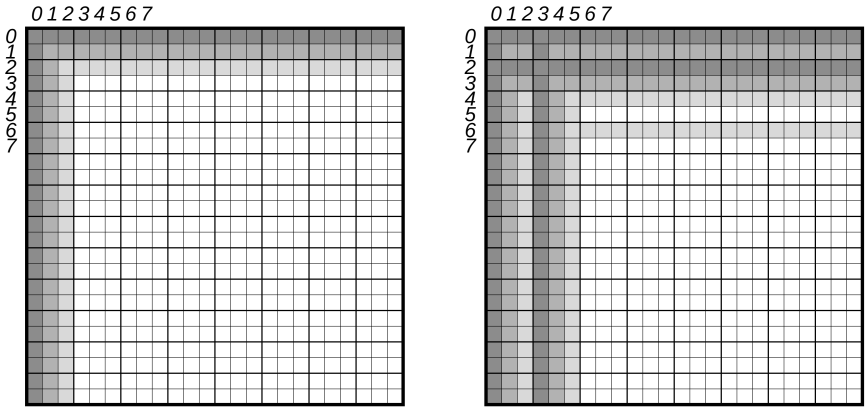


Figure 1: Performance of ScaLAPACK LU factorization routine on an 8×8 Intel Paragon

We measured the performance the ScaLAPACK LU factorization routine and its solution routine with various block sizes on the Intel Paragon. Figure 1 shows the performance on an 8×8 processor grid from $N = 1,000$ to 10,000 with block sizes of $N_b = 1, 5, 8, 20$, and 50. It shows that the near optimal performance is obtained when $N_b = 8$, and almost the same but slightly slower when $N_b = 5$. The performance deteriorated by 2-3% when $N_b = 20$, 13% when $N_b = 50$, and more than 45% when $N_b = 1$. If the data matrix is distributed with $N_b = 1$, it may be much more efficient to perform the factorization after redistributing the data matrix with the optimal block size of $N_{opt} = 8$.

In ScaLAPACK, the performance of the algorithm is greatly affected by the block size. However the PoLAPACK LU factorization is implemented with the concept of algorithmic blocking and always shows the best performance of $N_{opt} = 8$ irrespective of physical block sizes.

If a data matrix A is decomposed over 2-dimensional $p \times q$ processors with the block cyclic data distribution, it may be possible to regard the matrix A being decomposed along the row and column directions of processors. Then the new decomposition along the row and column directions are the same as applying permutation matrices from the left and the right, respectively. One step further. If we want to compute a matrix with a different block size, we may need to redistribute the matrix, and we can assume that the redistributed matrix is of the form $P_p \cdot A \cdot P_q^T$, where P_p and P_q are permutation matrices. It may be possible to avoid redistributing the matrix physically if the new computation doesn't follow the given ordering of the matrix A. That is, by assuming that the given matrix A is redistributed with a new optimal block size and the resulting matrix is $P_p \cdot A \cdot P_q^T$, it is now possible to apply the factorization to A with the optimal block size for the computation. And this factorization will show the same performance regardless of the physical block sizes if each processor gets



(a) Nopt = Nb (b) Nopt = 2 · Nb

Figure 2: The order of block to be computed in PoLAPACK. The darkest area is computed first. (a) when the optimal block size is the same as the physical block size ($N_{opt} = N_b$), (b) when the optimal block size is twice the physical block size ($N_{opt} = 2 \cdot N_b$).

the same size of the submatrix of A. These statements are illustrated with the following equations,

$$(P_p \, A \, P_q^T) \cdot (P_q x) \; = \; P_p \cdot b. \tag{1}$$

Let $A_1 = P_p \, A \, P_q^T$, and $x_1 = P_q \, x$. After factorizing $P_1 A_1 = P_1 \cdot (P_p A P_q^T) = L_1 \cdot U_1$, then we compute the solution vector x. The above equation Eq. 1 is transformed as follows:

$$L_1 \cdot U_1 \cdot (P_q x) \; = \; L_1 \cdot U_1 \cdot x_1 \; = \; P_1 \cdot (P_p b) \; = \; b_1.$$

Then, y_1 is computed from

$$L_1 \cdot y_1 \; = \; b_1, \tag{2}$$

and x_1 is computed from

$$U_1 \cdot x_1 \; = \; y_1. \tag{3}$$

Finally the solution vector x is computed from

$$P_q \cdot x \; = \; x_1. \tag{4}$$

The computations are performed with A and b in place with the optimal block size, and x is computed with P_q as in Eq. 4. But we want $P_p \cdot x$ rather than x in order to make x have the same physical data distribution as b. That is, it is required to compute

$$P_p \cdot x \; = \; P_p \cdot P_q^T \cdot x_1. \tag{5}$$

Assume that we have a 24×24 block matrix A is distributed with a block size of N_b over a 2×3 processor grid as in Figure 2. When the optimal block size is the same as

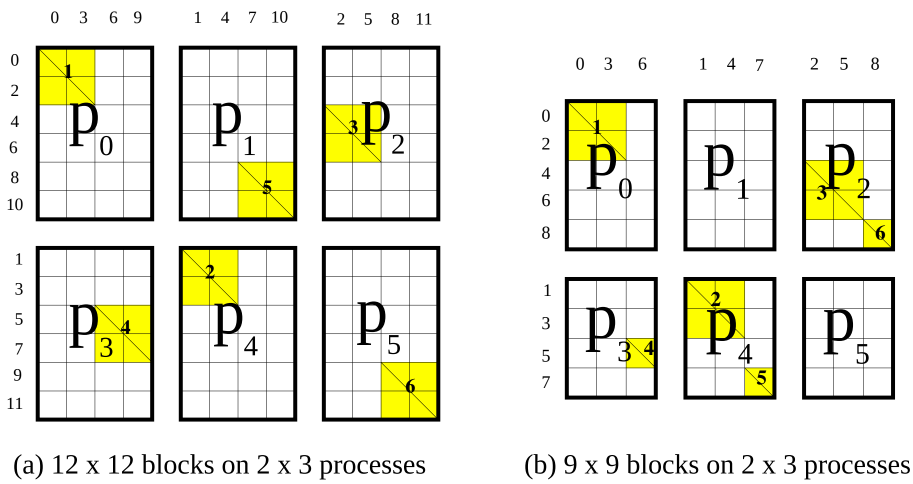


Figure 3: Computational Procedure in PoLAPACK. Matrices of 12×12 and 9×9 blocks are distributed on 2×3 processors with $N_{opt} = N_b$ and $N_{opt} = 2 \cdot N_b$, respectively.

the physical block size (i.e., $N_{opt} = N_b$), the computational ordering of the PoLAPACK is the same as that of the ScaLAPACK as in Figure 2(a). However, if the optimal block size is twice the physical block size (i.e., $N_{opt} = 2 \cdot N_b$), the PoLAPACK routine computes with two columns of blocks, $A(:,0)$ and $A(:,3)$, and two rows of blocks, $A(0,:)$ and $A(2,:)$ at the first step as in Figure 2(b). But those two columns and rows of blocks belong to the same column and row of processors, respectively. In this example, the computational orderings of the row and column blocks have been changed $(0,2),(1,3),(4,6),(5,7),\cdots$, and $(0,3),(1,4),(2,5),(6,9),(7,10),\cdots$, respectively. And the new orderings are obtained by multiplying P_p to A from the left and P_q^T from the right. This procedure isn't involved with any physical data redistribution.

3. Implementation of PoLAPACK LU Factorization

The PoLAPACK LU factorization routine is composed of three parts: LU factorization, triangular solver, and redistribution of the solution vector. We will describe the details of each part.

3.1. LU Factorization

We implemented the right-looking version of the LAPACK LU factorization routines, DGETRF, DGETF2, DLAMAX, DLAPIV, DLASWP, and DGETRS as the corresponding PoLAPACK LU factorization routines, PoDGETRF, PoDGETF2, PoDLAMAX, PoDLAPIV, PoDLASWP, and PoDGETRS, respectively. And we partially implemented the Level 2 and Level 3 BLAS routines, DGER, DGEMM, and DTRSM, as the corresponding PoLAPACK BLAS routines, PoDGER, PoDGEMM, and PoDTRSM, respectively.

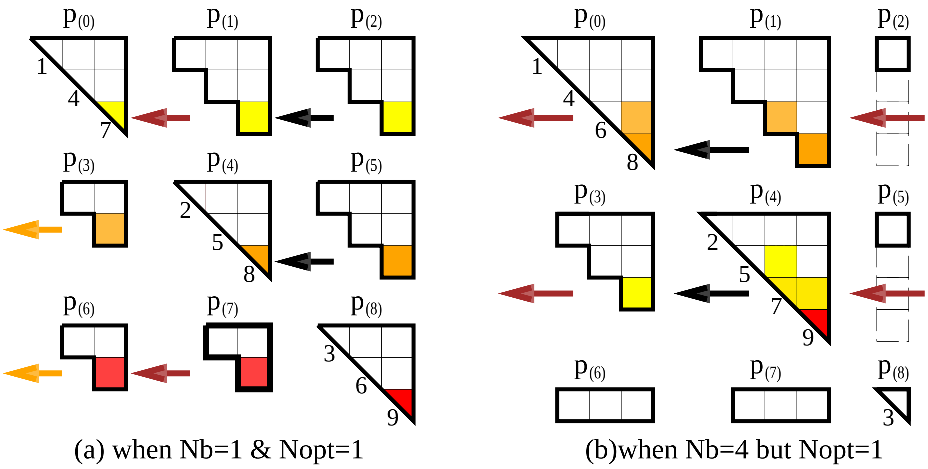


Figure 4: A snapshot of PoLAPACK solver. A matrix T of 9×9 blocks is distributed on 3×3 processors with $N_b = 1$ and $N_b = 4$, respectively, while the optimal computational block size for both cases is $N_{opt} = 1$.

Figure 3 shows the computational procedure of the PoLAPACK LU factorization. It is assumed that a matrix A of 12×12 blocks is distributed over a 2×3 processor grid as in Figure 3(a), and the LU routine computes 2 blocks at a time (imagine $N_b = 4$ but $N_{opt} = 8$). Since the routine follows the 2-D block cyclic ordering, the positions of the diagonal blocks are regularly changed by incrementing one column and one row of processors at each step. However, if A is 9×9 blocks as in Figure 3(b), the next diagonal block of $A(5,6)$ on $p_{(3)}$ is $A(7,7)$ on $p_{(4)}$, not on $p_{(1)}$. Then the next block is $A(8,8)$ on $p_{(2)}$. The computational procedure of the PoLAPACK is very complicated.

In the PoLAPACK, it is difficult to directly compute the local sizes of data from the global size of the remaining data matrix. And during the middle of the computation, a row and a column of processors may have no data at all to compute the factorization. It is necessary for each processor to keep the local sizes of data in each processor.

3.2. Triangular Solver

We implemented the Li and Coleman's algorithm [?] on a two dimensional processor grid for the PoLAPACK routines, PoDTRSM. The 2-dimensional version of the algorithm is already implemented in the PBLAS routine, PDTRSM [?]. But the implementation of PoDTRSM is much more complicated since the diagonal block may not be located regularly if p is not equal to q as in Figure 3.

If p is equal to q, the implementation is still complicated. Figure 4(a) shows a snapshot of the Li and Coleman's algorithm from the processors point-of-view, where 9×9 blocks of an upper triangular matrix T are distributed over a 3×3 processor grid with $N_b = N_{opt} = 1$. Let's look over the details of the algorithm to solve $x = T \setminus b$.

At first, the last block at $p_{(8)}$ computes $x(9)$ from $T(9,9)$ and $b(9)$. Processors in the last column update 2 blocks - actually $p_{(2)}$ and $p_{(5)}$ update $b(7)$ and $b(8)$, respectively - and send them to their left processors. The rest of b ($b(1:6)$) is updated later. At the second step, $p_{(4)}$ computes $x(8)$ from $T(8,8)$ and $b(8)$, the latter is received from $p_{(5)}$. While $p_{(1)}$ receives $b(7)$ from $p_{(2)}$, updates it, and sends it to $p_{(0)}$, $p_{(7)}$ updates a temporal $b(6)$ and sends it to $p_{(6)}$.

Figure 4(b) shows the same size of the matrix distribution T with $N_b = 4$, but it is assumed that the matrix T is derived with an optimal block size $N_{opt} = 1$. So the solution routine has to solve the triangular equations of Eq. 2 and Eq. 3 with $N_{opt} = 1$. The first two rows and the first two columns of processors have 4 rows and 4 columns of T, respectively, while the last row and the last column have 1 row and 1 column, respectively. Since $N_{opt} = 1$, the computation starts from $p_{(4)}$, which computes $x(9)$. Then $p_{(1)}$ and $p_{(4)}$ update $b(8)$ and $b(7)$, respectively, and send them to their left. The rest of b ($b(1:6)$) is updated later. At the next step, $p_{(0)}$ computes $x(8)$ from $T(8,8)$ and $b(8)$, the latter is received from $p_{(1)}$. While $p_{(3)}$ receives $b(7)$ from $p_{(4)}$, updates it, and sends it to the left $p_{(5)}$, $p_{(0)}$ updates a temporal $b(6)$ and sends it to its left $p_{(2)}$. However $p_{(2)}$ and $p_{(5)}$ don't have their own data to update or compute at the current step, and hand them over to their left without touching the data. The PoLAPACK solver has to comply with this kind of all abnormal cases.

3.3. Solution Vector Redistribution

It may be necessary to redistribute the solution vector x to $P_p \cdot P_q^T \cdot x$ as in Eq. 5. However, if p is equal to q, then P_p becomes P_q, and $P_p \cdot P_q^T \cdot x = x$, therefore, the redistribution is not necessary. But if p is not equal to q, the redistribution of x is required to get the solution with the same data distribution as the right hand vector b. And if p and q are relatively prime, then the problem is changed to all-to-all personalized communication.

Figure 5 shows a case of the physical block size $N_b = 1$ and the optimal block size $N_{opt} = 2$ on a 2×3 processor grid. Originally the vector b is distributed with $N_b = 1$ as the ordering on the left of Figure 5. But the solution vector x is distributed as the ordering on the right after the computation with $N_{opt} = 2$. The result is the same as a vector on the left is transposed twice – at first transposed with $N_b = 1$ to the vector on the top, then later transposed with $N_{opt} = 2$ to the vector on the right.

In PoLAPACK, it is necessary to redistribute the vector x to the same ordering of b if $p \neq q$. If we transpose the vector x twice, it may be possible to get x with the same ordering of b. However, all of the processors are involved in the redistribution process of the vector.

The pseudo code of the redistribution process of the solution vector is shown in Figure 6. At first, the column of processors, which holds the solution vector, computes the data ordering in each processor and the relative position in each block. Then they transpose their local data twice, where they only transpose the indices of data, not the real data. We also exploit the LCM concept to simplify the redistribution. If the vector is distributed with N_b and computed with N_{opt} on $p \times q$ processors, the computation is repeated with LCMNUM $= \mathrm{LCM}(p,q) \cdot \mathrm{LCM}(N_b, N_{opt})$.

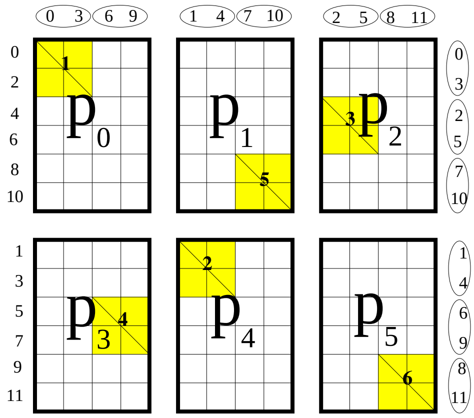


Figure 5: A snapshot of PoLAPACK solver. A matrix T of 9×9 blocks is distributed on 3×3 processors with $N_b = 1$ and $N_b = 4$, respectively, while the optimal computational block size for both cases is $N_{opt} = 1$.

1. Compute the local & global indices of data in each processor
2. 1st transpose its indices of data (with N_{opt})
   (not transpose the real data)
3. 2nd transpose its indices of data (with N_b)
   (not transpose the real data)
4. For $i = 0$ to $p - 1$
   i) Copy & send data with indices to P_{i+1}
   ii) Receive & move data with indices from P_{i-1}

Figure 6: The pseudocode of the solution vector redistribution.

3.4. PoLAPACK LU Result

We implemented the PoLAPACK LU factorization routine and measured its performance on an 8×8 processor grid of the Intel Paragon. Figure 7 shows the performance of the routine with the physical block sizes of $N_b = 1, 5, 8, 20$, and 50, but the optimal block size of $N_{opt} = 8$. As shown in Figure 7, the performance lines are very close to the others and always show nearly the maximum performance irrespective of the physical block sizes. Since all processors don't have the same size of the submatrices of A with various block sizes, some processors have more data to compute than others, which causes computational load imbalance among processors and slight performance degradation. For example, the line with a small white circle in Figure 7 shows the case of $N_b = 50$, in which processors in the first half have more data to compute than processors in the second half if the matrix size $N = 7,000$

or $9,000$.

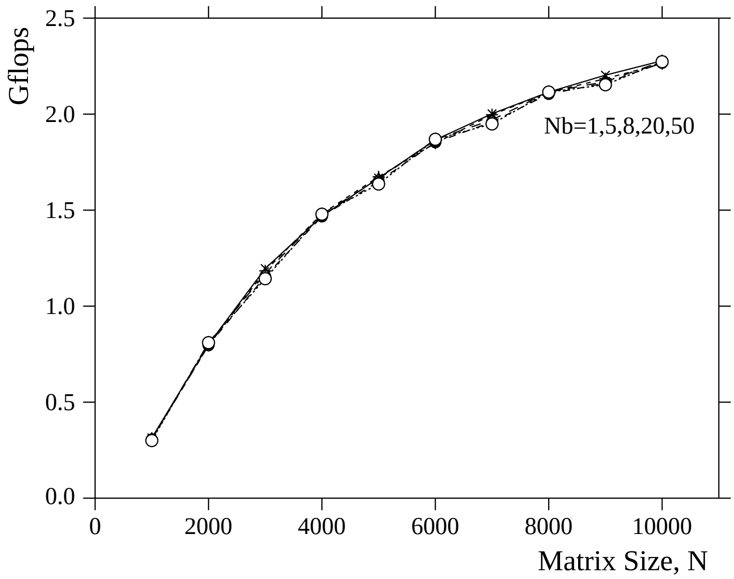


Figure 7: Performance of PoLAPACK LU on an 8×8 Intel Paragon

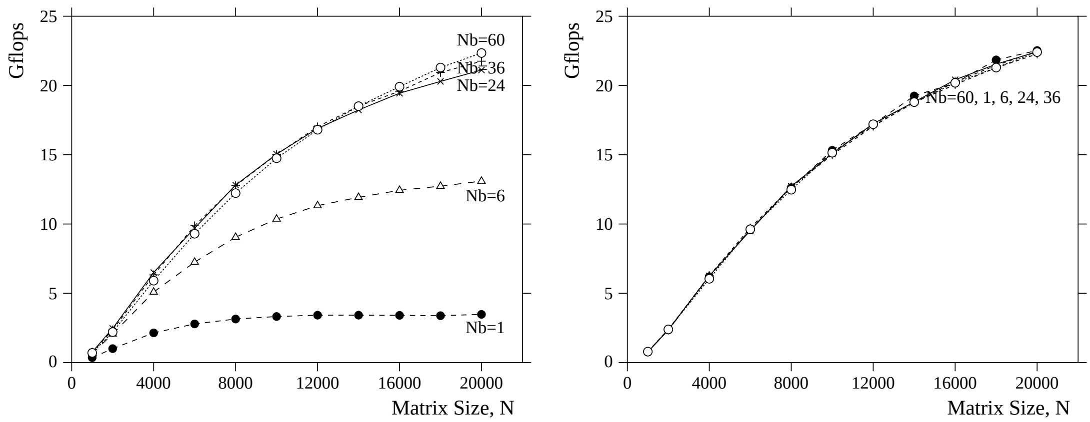


Figure 8: Performance of ScaLAPACK LU and PoLAPACK LU on an 8×8 Cray T3E

4. PoLAPACK QR Factorization

The QR factorization is used to solve the least squares problem [**?**],

$$\min_{x \in \Re} \| \, A\,x \, - \, b \, \|_2. \tag{6}$$

Given an $M \times N$ matrix A, the QR factorization is computed by $A = QR$, where Q is an $M \times M$ orthogonal matrix, and R is an $M \times N$ upper triangular matrix.

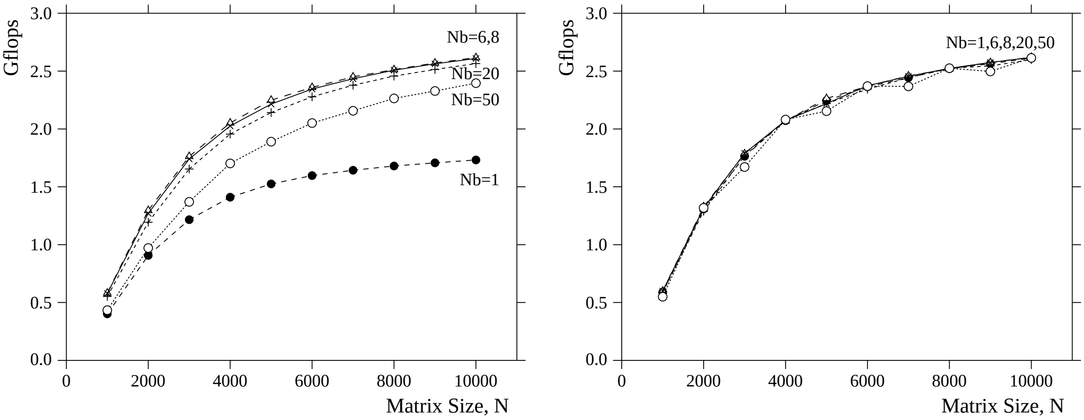


Figure 9: Performance of ScaLAPACK QR and PoLAPACK QR on an 8×8 Intel Paragon

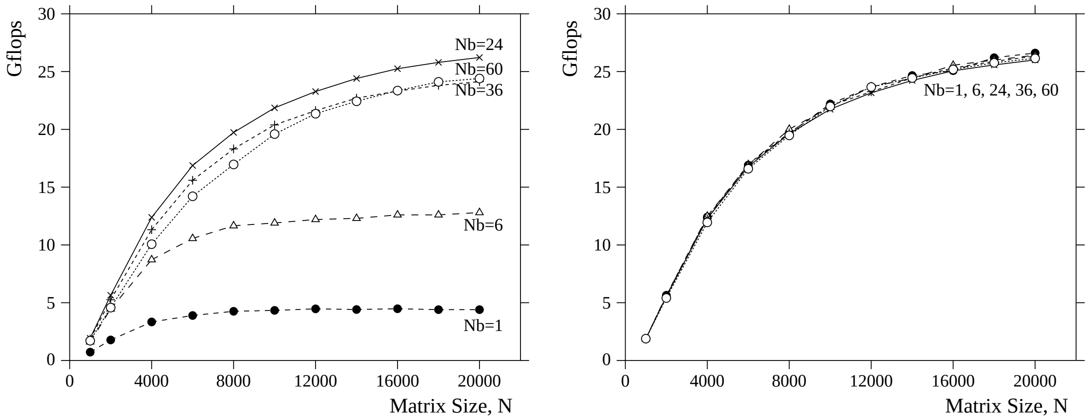


Figure 10: Performance of ScaLAPACK QR and PoLAPACK QR on an 8×8 Cray T3E

The PoLAPACK QR factorization and its solution of the factored matrix equations are performed in a manner analogous to the PoLAPACK LU factorization and the solution of the triangular systems. Eq. 6 is changed to

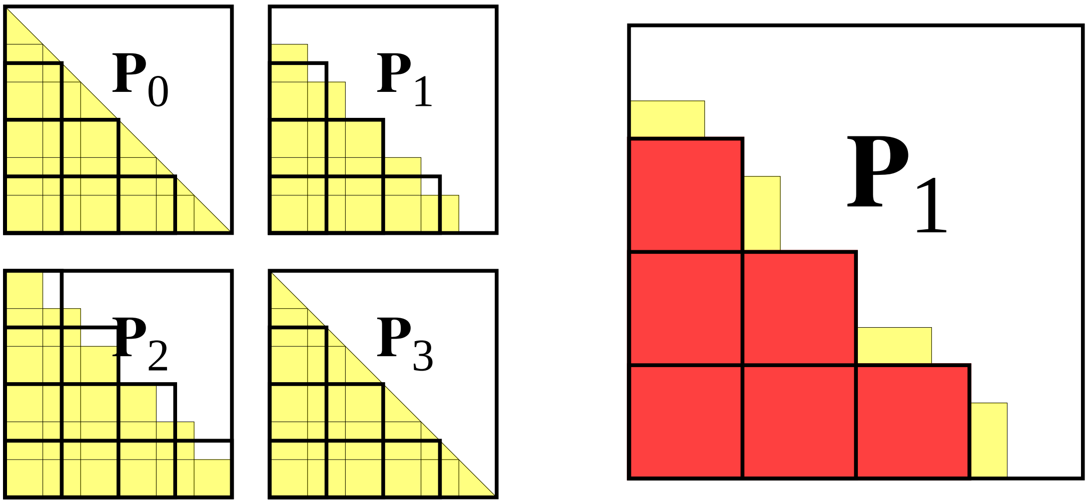
$$\| \, A\,x \,-\, b \, \|_2 \,=\| \, (P_p A P_q^T) \cdot (P_q x) \,-\, P_p b \, \|_2$$

where P_p and P_q are permutation matrices. Let $A_2 = P_p \cdot A \cdot P_q^T$, $x_2 = P_q\, x$, and $b_2 = P_p\, b$. After factorizing $A_2 = P_p A P_q^T$ to $Q_2 \cdot R_2$, then we compute the solution vector x. The above equation is transformed as follows:

$$\| \, Q_2 \cdot R_2 \cdot (P_q x) \,-\, (P_p b) \, \|_2 \,=\| \, Q_2 \cdot R_2 \cdot x_2 \,-\, b_2 \, \|_2. \tag{7}$$

If applying Q_2^T to b_2 to form b_3, Eq. 7 is changed to

$$\| \, R_2 \cdot x_2 \,-\, Q_2^T b_2 \, \|_2 \,=\| \, R_2 \cdot x_2 \,-\, b_3 \, \|_2,$$



(a) data distribution on a 2x2 processor grid (b) data movement on P1 before the computation

Figure 11: Data Preservation of the lower triangular matrix by changing the computational block size

where $b_3 = Q_2^T b_2$. Then x_2 is computed from

$$R_2 \cdot x_2 \;=\; b_3.$$

Finally the solution vector x is computed from

$$P_q \cdot x \;=\; x_2.$$

Again as in Eq. 5, we want $P_p \cdot x = P_p P_q^T x$ to make x have the same physical data distribution as b.

Figures 9 and 10 show the performance of the ScaLAPACK and PoLAPACK QR factorizations and their solution on an 8×8 processor grid of the Intel Paragon and the Cray T3E, respectively. Performance of the ScaLAPACK QR factorization routine depends on the physical block size, and the best performance is obtained when $N_b = 6$ on an Intel Paragon, and $N_b = 24$ on a Cray T3E. However the PoLAPACK QR factorization routine, which computes with the optimal block size of N_{opt}, always shows nearly the maximum performance independent of physical block sizes.

5. PoLAPACK Cholesky Factorization

The Cholesky factorization factors an $N \times N$, symmetric, positive-definite matrix A into the product of a lower triangular matrix L and its transpose, i.e., $A = LL^T$ (or $A = U^T U$, where U is upper triangular). It is assumed that the lower triangular portion of A is stored in the lower triangle of a two-dimensional array and that the computed elements of L overwrite the

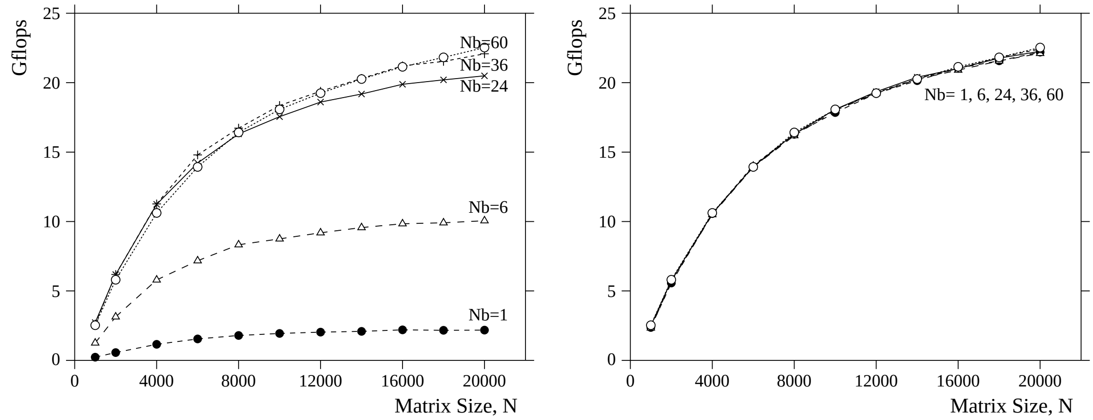


Figure 12: Performance of ScaLAPACK and PoLAPACK Cholesky on an 8×8 Cray T3E

given elements of A. If A is factored to $L \cdot L^T$, compute y with $L \cdot y = b$, then compute x from the following equation, $L^T \cdot x = y$.

Though A is symmetric, $P_p A P_q^T$ is not symmetric if $p \neq q$. That is, if $P_p A P_q^T$ is not symmetric, it is impossible to exploit the algorithmic blocking technique to the Cholesky factorization routine as used in the PoLAPACK LU and QR factorization. If $p \neq q$, the PoLAPACK Cholesky computes the factorization with the physical block size. That is, it computes the factorization as the same way of the ScaLAPACK Cholesky routine. However, it is possible to obtain the benefit of algorithmic blocking for the limited case of $p = q$.

In the PoLAPACK Cholesky factorization, we compute $P_p A P_p^T = L_3 L_3^T$ when $p = q$. Then

$$(P_p A P_p^T) \cdot (P_p x) \;=\; L_3 L_3^T \cdot (P_p x) \;=\; P_p b = b_4,$$

$$L_3 \cdot y_3 = b_4,$$

$$L_3^T \cdot x_3 = y_3.$$

In the PoLAPACK Cholesky factorization, the redistribution of the solution vector is omitted, since $p = q$. However, for the symmetric matrix, the data is contained in only the lower (or upper) triangular part of the matrix, and data in the other part should be preserved after the computation. So it is necessary to use another trick in order to compute the Cholesky factorization with the algorithmic blocking.

Figure 11 shows a case to compute the Cholesky factorization with $N_b = 2$ and $N_{opt} = 3$ on a 2×2 processor grid. For the non-diagonal processors, such as $p_{(1)}$ and $p_{(2)}$, the original data should be reallocated. For example, if the physical block size is 2 ($N_b = 2$), the data is stored in the lightly shaded area. But for the optimal block size is 3 ($N_{opt} = 3$, the data should be located in the dark gray area before the computation. $p_{(1)}$ should send the lightly shaded area to $p_{(2)}$, and it should receive data in the non-overlapped area from $p_{(2)}$

as in Figure 11(b). But, before moving the data, processors should save the original data and restore them after the computation to preserve data in the other part of the triangular matrix.

Figure 12 shows the performance of the ScaLAPACK and the PoLAPACK Cholesky factorization and their solution on an 8×8 processor grid of the Cray T3E. Similarly, the performance of the ScaLAPACK Cholesky factorization routine depends on the physical block size. However the PoLAPACK Cholesky factorization routine, which computes with the optimal block size of $N_{opt} = 60$, always shows the maximum performance.

6. Conclusions

Generally in most parallel factorization algorithms, a column of processors performs the factorization on a column of blocks of A at a time, whose block size is already fixed, and then the other processors update the rest of the matrix. If the block size is very small or very large, then the processors can't show their optimal performance, and the data matrix may be redistributed for a better performance. The computation follows the original ordering of the matrix.

It may be faster and more efficient to perform the computation, if possible, by combining several columns of blocks if the block size is small, or by splitting a large column of blocks if the block size is large. This is the main concept of algorithmic blocking. The PoLAPACK factorization routines rearrange the ordering of the computation. They compute $P_p A P_q^T$ instead of directly computing A. The computation proceeds with the optimal block size without physically redistributing A. And the solution vector x is computed by solving triangular systems, then converting x to $P_p P_q^T x$. The final rearrangement of the solution vector can be omitted if $p = q$ or $N_b = N_{opt}$.

According to the results of the ScaLAPACK and the PoLAPACK LU, QR, and Cholesky factorization routines on the Intel Paragon and the Cray T3E, the ScaLAPACK factorization routines have a large performance difference with different values of N_b, however the PoLAPACK factorizations always show a steady performance, which is the near optimal, irrespective of the values of N_b. The routines we presented in this paper are developed based on the block cyclic data distribution. This simple idea can be easily applied to the other data distributions. But it is required to develop specific algorithms to rearrange the solution vector for each distribution.

It is very difficult to get the best value of N_{opt} for a specific factorization routine on a concurrent computer with distributed memory multiprocessors. However, it may be possible to obtain the optimal value by applying to parallel libraries the idea of ATLAS [?], which has been developed for automatic generation and optimization of numerical software on a single processor. This kind of idea is currently applied into higher level kernels such as LAPACK's LU and Cholesky [?], not yet to parallel software libraries.

7. References

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A High-Performance Matrix-Multiplication Algorithm on a Distributed-Memory Parallel Computer Using Overlapped Communication. *IBM Journal of Research and Development*, 38(6):673–681, 1994.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. In *Proceedings of Supercomputing '90*, pages 1–10. IEEE Press, 1990.

[3] P. V. Bangalore. The Data-Distribution-Independent Approach to Scalable Parallel Libraries. 1995. Master Thesis, Mississippi State University.

[4] L. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. In *Proceedings of SIAM Conference on Parallel Processing*, 1997.

[5] L. Blackford, J. Choi, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide*. SIAM Press, Philadelphia, PA, 1997.

[6] J. Choi. A New Parallel Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers. *Concurrency: Practice and Experience*, 10:655–670, 1998.

[7] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. LAPACK Working Note 100, Technical Report CS-95-292, University of Tennessee, 1995.

[8] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming*, 5:173–184, 1996.

[9] J. Choi, J. J. Dongarra, and D. W. Walker. The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers. In *Proceedings of Environment and Tools for Parallel Scientific Computing Workshop, (Saint Hilaire du Touvet, France)*, pages 3–15. Elsevier Science Publishers, September 7-8, 1992.

[10] J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. *Concurrency: Practice and Experience*, 6:543–570, 1994.

[11] J. J. Dongarra and S. Ostrouchov. LAPACK Block Factorization Algorithms on the Intel iPSC/860. LAPACK Working Note 24, Technical Report CS-90-115, University of Tennessee, October 1990.

[12] G. H. Golub and C. V. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 1989. Second Edition.

[13] S. Huss-Lederman, E. M. Jacobson, A. Tsao, and G. Zhang. Matrix Multiplication on the Intel Touchstone Delta. *Concurrency: Practice and Experience*, 6:571–594, 1994.

[14] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[15] G. Li and T. F. Coleman. A Parallel Triangular Solver for a Distributed-Memory Multiprocessor. *SIAM J. of Sci. Stat. Computing*, 9:485–502, 1986.

[16] W. Lichtenstein and S. L. Johnsson. Block-Cyclic Dense Linear Algebra. *SIAM J. of Sci. Stat. Computing*, 14(6):1259–1288, 1993.

[17] R. van de Geijn and J. Watts. SUMMA Scalable Universal Matrix Multiplication Algorithm. LAPACK Working Note 99, Technical Report CS-95-286, University of Tennessee, 1995.

[18] R. A. van de Geijn. *Using PLAPACK*. The MIT Press, Cambridge, 1997.

[19] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proceedings of SC '98*. IEEE Publication, 1998.

[20] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS project. *Parallel Computing (submitted)*, 2000.