A COMPARISON OF PARALLEL SOLVERS FOR DIAGONALLY DOMINANT AND GENERAL NARROW-BANDED LINEAR SYSTEMS

PETER ARBENZ[†], ANDREW CLEARY[‡], JACK DONGARRA[§], AND MARKUS HEGLAND[¶]

Abstract. We investigate and compare stable parallel algorithms for solving diagonally dominant and general narrow-banded linear systems of equations. Narrow-banded means that the bandwidth is very small compared with the matrix order and is typically between 1 and 100. The solvers compared are the banded system solvers of ScaLAPACK [11] and those investigated by Arbenz and Hegland [3, 6]. For the diagonally dominant case, the algorithms are analogs of the well-known tridiagonal cyclic reduction algorithm, while the inspiration for the general case is the lesser-known bidiagonal cyclic reduction, which allows a clean parallel implementation of partial pivoting. These divide-and-conquer type algorithms complement fine-grained algorithms which perform well only for wide-banded matrices, with each family of algorithms having a range of problem sizes for which it is superior. We present theoretical analyses as well as numerical experiments conducted on the Intel Paragon.

Key words. narrow-banded linear systems, stable factorization, parallel solution, cyclic reduction, ScaLAPACK

1. Introduction. In this paper we compare implementations of direct parallel methods for solving banded systems of linear equations

$$A\mathbf{x} = \mathbf{b}. \tag{1.1}$$

The n-by-n matrix A is assumed to have lower half-bandwidth k_l and upper half-bandwidth k_u, meaning that k_l and k_u are the smallest integers that imply

$$a_{ij} \neq 0 \quad \Longrightarrow \quad -k_l \leq j - i \leq k_u.$$

We assume that the matrix A has a *narrow* band, such that $k_l + k_u \ll n$. Linear systems with wide band can be solved efficiently by methods similar to full system solvers. In particular, parallel algorithms using two-dimensional mappings (such as the torus-wrap mapping) and Gaussian elimination with partial pivoting have achieved reasonable success [16, 10, 18]. The parallelism of these algorithms is the same as that of dense matrix algorithms applied to matrices of size $\min\{k_l, k_u\}$, independent of n, from which it is obvious that small bandwidths severely limit the usefulness of these algorithms, even for large n.

Parallel algorithms for the solution of banded linear systems with small bandwidth have been considered by many authors, both because they serve as a canonical form of recursive equations, as well as having direct applications. The latter include the solution of eigenvalue problems with inverse iteration [17], spline interpolation and smoothing [9], and the solution of boundary value problems for ordinary differential equations using finite difference or finite element methods [27]. For these

[†]Institute of Scientific Computing, Swiss Federal Institute of Technology (ETH), 8092 Zurich, Switzerland (`arbenz@inf.ethz.ch`)

[‡]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-561, Livermore CA 94551, U.S.A. (`acleary@llnl.gov`)

[§]Department of Computer Science, University of Tennessee, Knoxville TN 37996-1301, U.S.A. (`dongarra@cs.utk.edu`)

[¶]Computer Sciences Laboratory, RSISE, Australian National University, Canberra ACT 0200, Australia (`Markus.Hegland@anu.edu.au`)

one-dimensional applications, bandwidths typically vary between 2 and 30. The discretisation of partial differential equations leads to applications with slightly larger bandwidths, for example, the computation of fluid flow in a long narrow pipe. In this case, the number of grid points orthogonal to the flow direction is much smaller than the number of grid points along the flow and this results in a matrix with bandwidth relatively small compared to the total size of the problem. There is a tradeoff for these type of problems between band solvers and general sparse techniques, in that the band solver assumes that all of the entries within the band are nonzero, which they are not, and thus performs unnecessary computation, but its data structures are much simpler and there is no indirect addressing as in general sparse methods.

In section 2 we review an algorithm for the class of nonsymmetric narrow-banded matrices that can be factored stably without pivoting, such as diagonally dominant matrices or M-matrices. This algorithm has been discussed in detail in [3, 11] where the performance of implementations of this algorithm on distributed memory multicomputers like the Intel Paragon [3] or the IBM SP/2 [11] is analyzed as well. Johnsson [23] considered the same algorithm and its implementation on the Thinking Machine CM-2 which required a different model for the complexity of the interprocessor communication. Related algorithms have been presented in [26, 15, 14, 7, 12, 28] for shared memory multiprocessors with a small number of processors. The algorithm that we consider here can be interpreted as a generalization of *cyclic reduction* (CR), or more usefully, as Gaussian elimination applied to a symmetrically permuted system of equations $(PAP^T)P\mathbf{x} = P\mathbf{b}$. The latter interpretation has important consequences, such as it implies that the algorithm is backward stable [5]. It can also be used to show that the permutation necessarily causes Gaussian elimination to generate *fill-in* which in turn increases the computational complexity as well as the memory requirements of the algorithm.

In section 3 we consider algorithms for solving (1.1) for arbitrary (narrow-) banded matrices A that may require pivoting for stability reasons. This algorithm was proposed and thoroughly discussed in [6]. It can be interpreted as a generalization of the well-known (block) tridiagonal cyclic reduction to (block) bidiagonal matrices, and again, it is also equivalent to Gaussian elimination applied to a permuted (nonsymmetrically for the general case) system of equations $(PAQ^T)Q\mathbf{x} = P\mathbf{b}$. Block bidiagonal cyclic reduction for the solution of banded linear systems was introduced by Hegland [19].

In section 4 we compare the ScaLAPACK implementations [11] of the two algorithms above with the implementations by Arbenz [3] and Arbenz and Hegland [6], respectively, by means of numerical experiments conducted on the Intel Paragon. ScaLAPACK is a software package with a diverse user community. Each subroutine should have an easily intelligible calling sequence (interface) and work with easily manageable data distributions. These constraints may reduce the performance of the code. The other two codes are experimental. They have been optimized for low communication overhead. The number of messages sent among processors and the marshaling process has been minimized for the task of solving a system of equations. The code does, for instance, not split the LU factorization from the forward elimination which prohibits the solution of a sequence of systems of equations with equal system matrix (without factoring the system matrix over and over again). Our comparisons shall give an answer to the question how much performance the ScaLAPACK algorithms may have lost through the constraint to be user friendly. We further continue a discussion started in [5] on the overhead introduced by partial pivoting. Is it necessary to have a pivoting as well as a non-pivoting algorithm for nonsymmetric band systems in ScaLAPACK? In LAPACK [2], for instance, there are only pivoting subroutine for solving dense and banded systems of equations, respectively.

2. Parallel Gaussian elimination for the diagonally dominant case. In this section we assume that the matrix $A = [a_{ij}]_{i,j=1,\dots,n}$ in (1.1) is diagonally dominant, i.e., that

$$|a_{ii}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}|, \quad i = 1, \dots, n.$$

Then the system of equations can be solved by Gaussian elimination without pivoting in the following three steps:

1. Factorization into $A = LU$.
2. Solution of $Lz = y$ (forward elimination)
3. Solution of $Ux = z$ (backward substitution)

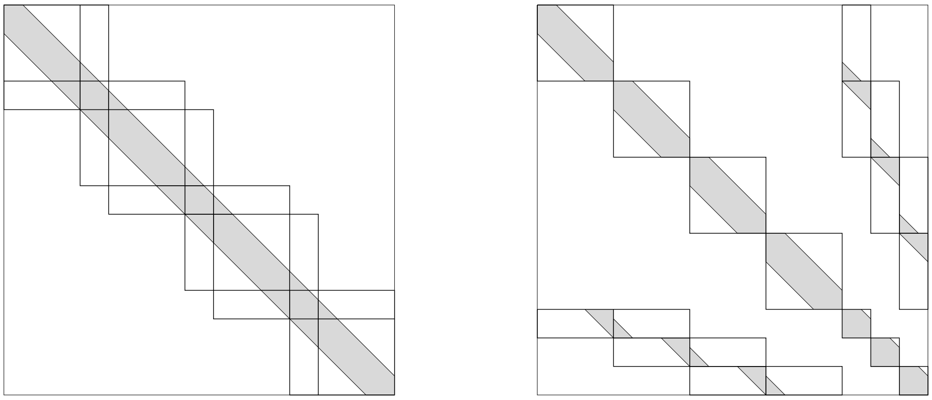
The lower and upper triangular Gauss factors L and U are banded with bandwidths k_l and k_u, respectively, where k_l and k_u are the half-bandwidths of A. The number of floating point operations φ_n for solving the banded system (1.1) with r right-hand sides by Gaussian elimination is (see also e.g. [17])

$$\varphi_n = (2k_u+1)k_l n + (2k_l+2k_u-1)rn + \mathcal{O}((k+r)k^2), \qquad k := \max\{k_l, k_u\}. \tag{2.1}$$

For solving (1.1) in parallel on a p processor multicomputer we partition the matrix A, the solution vector $\mathbf{x}$ and the right-hand hand side $\mathbf{b}$ according to

$$\begin{pmatrix} A_1 & B_1^U & & & & & \\ B_1^L & C_1 & D_2^U & & & & \\ & D_2^L & A_2 & B_2^U & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & B_{p-1}^L & C_{p-1} & D_p^U \\ & & & & D_p^L & A_p \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \boldsymbol{\xi}_1 \\ \mathbf{x}_2 \\ \vdots \\ \boldsymbol{\xi}_{p-1} \\ \mathbf{x}_p \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \boldsymbol{\beta}_1 \\ \mathbf{b}_2 \\ \vdots \\ \boldsymbol{\beta}_{p-1} \\ \mathbf{b}_p \end{pmatrix}, \tag{2.2}$$

where $A_i \in \mathbb{R}^{n_i \times n_i}$, $C_i \in \mathbb{R}^{k\times k}$, $\mathbf{x}_i$, $\mathbf{b}_i \in \mathbb{R}^{n_i}$, $\boldsymbol{\xi}_i$, $\boldsymbol{\beta}_i \in \mathbb{R}^k$, and $\sum_{i=1}^p n_i + (p-1)k = n$. This *block tridiagonal* partition is feasible only if $n_i > k$. This condition restricts the degree of parallelism, i.e. the maximal number of processor p that can be exploited for parallel execution, $p < (n+k)/(2k)$. The structure of A and its submatrices is depicted in Fig. 2.1(a) for the case $p = 4$. In ScaLAPACK [11, 8], the local portions of A on each processor are stored in the LAPACK scheme as depicted in Fig. 2.2. This input scheme requires a preliminary step of moving the triangular block D_i^L from processor $i-1$ to processor i. This transfer of the block can be overlapped with computation and has a negligible effect on the overall performance of the algorithm [11]. The input format used by Arbenz does not require this initial data movement.



(a) (b)

FIG. 2.1. *Non-zero structure (shaded area) of (a) the original and (b) the block odd-even permuted band matrix with $k_l > k_u$.*

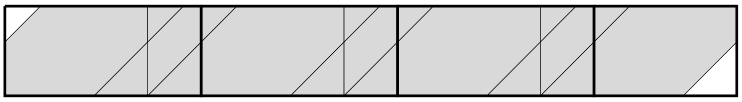


FIG. 2.2. *Storage scheme of the band matrix. The thick lines frame the local portions of A.*

We now execute the first step of block-cyclic reduction [22]. This is best regarded as block Gaussian elimination of the block odd-even permuted A,

$$\left[\begin{array}{ccccc|cccc} A_1 & & & & & B_1^U & & & \\ & A_2 & & & & D_2^L & B_2^U & & \\ & & \ddots & & & & \ddots & \ddots & \\ & & & A_{p-1} & & & & \ddots & B_{p-1}^U \\ & & & & A_p & & & & D_p^L \\ \hline B_1^L & D_2^U & & & & C_1 & & & \\ & B_2^L & \ddots & & & & C_2 & & \\ & & \ddots & \ddots & & & & \ddots & \\ & & & B_{p-1}^L & D_p^U & & & & C_{p-1} \end{array}\right] \left[\begin{array}{c} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_{p-1} \\ \mathbf{x}_p \\ \hline \boldsymbol{\xi}_1 \\ \boldsymbol{\xi}_2 \\ \vdots \\ \boldsymbol{\xi}_{p-1} \end{array}\right] = \left[\begin{array}{c} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_{p-1} \\ \mathbf{b}_p \\ \hline \boldsymbol{\beta}_1 \\ \boldsymbol{\beta}_2 \\ \vdots \\ \boldsymbol{\beta}_{p-1} \end{array}\right]. \tag{2.3}$$

The structure of this matrix is depicted in Fig. 2.1(b). We write (2.3) in the form

$$\begin{bmatrix} \hat{A} & B^U \\ B^L & C \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\xi} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \boldsymbol{\beta} \end{bmatrix}, \tag{2.4}$$

where the respective submatrices and subvectors are indicated by the lines in equation (2.3). If $LR = \hat{A}$ is the ordinary LU factorization of $\hat{A}$ then

$$\begin{bmatrix} \hat{A} & B^U \\ B^L & C \end{bmatrix} = \begin{bmatrix} L & 0 \\ B^L R^{-1} & I \end{bmatrix} \begin{bmatrix} R & L^{-1} B^U \\ 0 & S \end{bmatrix}, \qquad S = C - B^L \hat{A}^{-1} B^U. \tag{2.5}$$

The matrices $B_i^L R_i^{-1}$, $L_i^{-1} B_i^U$, $D_i^U R_i^{-1}$, and $L_i^{-1} D_i^L$ overwrite B_i^L, B_i^U, D_i^U, and D_i^U, respectively. As D_i^U, and D_i^L, suffer from fill-in, cf. Fig. 2.3, additional memory

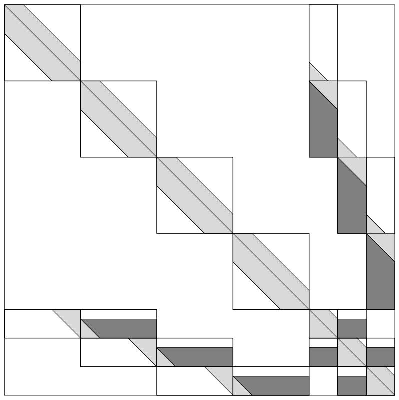


FIG. 2.3. *Fill-in produced by block Gaussian elimination. The bright-shaded areas indicate original nonzeros, the dark-shaded areas indicate the (potential) fill-in.*

space for $(k_l + k_u)n$ floating point numbers has to be provided. The overall memory requirement of the parallel algorithm is about twice as high as that of the sequential algorithm. The blocks $B_i^L R_i^{-1}$ and $L_i^{-1} B_i^U$ keep the structure of B_i^L and B_i^U and can be stored at their original places, cf. Fig. 2.2.

The Schur complement S of $\hat{A}$ in A is a diagonally dominant $(p-1)$-by-$(p-1)$ block tridiagonal matrix whereby the blocks are k-by-k,

$$S = \begin{pmatrix} T_1 & U_2 & & & \\ V_2 & T_2 & U_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & U_{p-1} \\ & & & V_{p-1} & T_{p-1} \end{pmatrix}, \tag{2.6}$$

where

$$\begin{aligned} T_i &= C_i - B_i^L A_i^{-1} B_i^U - D_{i+1}^U A_{i+1}^{-1} D_{i+1}^L \\ &= C_i - (B_i^L R_i^{-1})(L_i^{-1} B_i^U) - (D_{i+1}^U R_{i+1}^{-1})(L_{i+1}^{-1} D_{i+1}^L), \\ U_i &= -(D_i^U R_i^{-1})(L_i^{-1} B_i^U), \qquad V_i = -(B_i^L R_i^{-1})(L_i^{-1} D_i^L). \end{aligned}$$

As indicated in Fig. 2.3 these blocks are not full if $k_l < k$ or $k_u < k$. This is taken into account in the ScaLAPACK implementation but not in the implementation by Arbenz where the block-tridiagonal CR solver treats the k-by-k blocks as full blocks. Using the factorization (2.5), (2.4) gets

$$\begin{bmatrix} R & L^{-1}B^U \\ & S \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\xi} \end{bmatrix} = \begin{bmatrix} L & \\ B^L R^{-1} & I \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{b} \\ \boldsymbol{\beta} \end{bmatrix} = \begin{bmatrix} L^{-1} & \\ -B^L R^{-1} L^{-1} & I \end{bmatrix} \begin{bmatrix} \mathbf{b} \\ \boldsymbol{\beta} \end{bmatrix} =: \begin{bmatrix} \mathbf{c} \\ \boldsymbol{\gamma} \end{bmatrix}, \tag{2.7}$$

where the sections $\mathbf{c}_i$ and $\boldsymbol{\beta}_i$ of the vectors $\mathbf{c}$ and $\boldsymbol{\beta}$ are given by

$$\mathbf{c}_i = L_i^{-1} \mathbf{b}_i, \qquad \boldsymbol{\gamma}_i = \boldsymbol{\beta}_i - B_i^L R_i^{-1} \mathbf{c}_i - D_{i+1}^U R_{i+1}^{-1} \mathbf{c}_{i+1}.$$

Up to this point of the algorithm, no interprocessor communication is necessary, as each processor independently factors its diagonal block of $\hat{A}$, $A_i = L_i R_i$, and computes the blocks $B_i^L R_i^{-1}$, $L_i^{-1} B_i^U$, $D_i^U R_i^{-1}$, $L_i^{-1} D_i^L$, and $L_i^{-1}\mathbf{b}_i$. Each processor forms its portions of the *reduced system* $S\boldsymbol{\xi} = \boldsymbol{\gamma}$,

$$\begin{bmatrix} -D_i^U R_i^{-1} L_i^{-1} D_i^L & -D_i^U R_i^{-1} L_i^{-1} B_i^U \\ -B_i^L R_i^{-1} L_i^{-1} D_i^L & C_i - B_i^L R_i^{-1} L_i^{-1} B_i^U \end{bmatrix} \in \mathbb{R}^{2k\times 2k} \quad \text{and} \quad \begin{bmatrix} -D_i^U R_i^{-1}\mathbf{c}_i \\ \boldsymbol{\beta}_i - B_i^L R_i^{-1}\mathbf{c}_i \end{bmatrix} \in \mathbb{R}^{2k}.$$

Standard theory in Gaussian elimination shows that the reduced system is diagonally dominant. One option is to solve the reduced system on a single processor. This may be reasonable on shared memory multiprocessors with small processor numbers [25, p.124], but complexity analysis reveals that this quickly dominates total computation time on multicomputers with many processors. An attractive parallel alternative for solving the system $S\boldsymbol{\xi} = \boldsymbol{\gamma}$ is block cyclic reduction [4]. Implementationally, the reduction step described above is repeated until the reduced system becomes a dense k-by-k system, which is trivially solved on a single processor. Since the order of the remaining system is halved in each reduction step, $\lfloor \log_2(p-1) \rfloor$ steps are needed. Note that the degree of parallelism is also halved in each step.

In order to understand how we proceed with CR we take another look at how the $(2p-1)$-by-$(2p-1)$ block tridiagonal matrix A in (2.2) is distributed over the p processors. Processor i, $i < p$, holds the 2-by-2 diagonal block $\begin{pmatrix} A_i & B_i^U \\ B_i^L & C_i \end{pmatrix}$ together with the block D_i^U above it and the block D_i^L to the left of it. To obtain a similar situation with the reduced system we want the 2-by-2 diagonal blocks $\begin{pmatrix} T_{i-1} & U_i \\ V_i & T_i \end{pmatrix}$ of S in (2.6) together with the block U_{i-1} above and V_{i-1} to the left to reside on processor i, $i = 2, 4, \ldots$ which then allows to proceed with these reduced number of processors as earlier. To that end the odd-numbered processor i has to send some of its portion of S to the neighboring processors $i-1$ and $i+1$. The even-numbered processors will then continue to compute the even-numbered portions of $\boldsymbol{\xi}$. Having done so the odd-numbered processors receive $\boldsymbol{\xi}_{i-1}$ and $\boldsymbol{\xi}_{i+1}$ from their neighboring processors which allows them to compute their portion $\boldsymbol{\xi}_i$ of $\boldsymbol{\xi}$ provided they know the i-th block row of S. This is easily attained if in the beginning of this CR step not only the odd-numbered but all processors send the needed information to their neighbors.

Finally, if the vectors $\boldsymbol{\xi}_i$, $1 \le i < p$, are known, each processor can compute its section of $\mathbf{x}$,

$$\begin{aligned} \mathbf{x}_1 &= R_1^{-1}(\mathbf{c}_1 - L_i^{-1} B_1^U \boldsymbol{\xi}_1), \\ \mathbf{x}_i &= R_i^{-1}(\mathbf{c}_i - L_i^{-1} D_1^L \boldsymbol{\xi}_{i-1} - L_i^{-1} B_i^U \boldsymbol{\xi}_i), \qquad 1 < i < p, \\ \mathbf{x}_p &= R_p^{-1}(\mathbf{c}_p - L_p^{-1} D_p^L \boldsymbol{\xi}_{p-1}). \end{aligned} \tag{2.8}$$

Notice that the even-numbered processors have to receive $\boldsymbol{\xi}_{i-1}$ from their direct neighbors. In the *back substitution phase* (2.8) each processor can again proceed independently without interprocessor communication.

The *parallel* complexity of the above divide-and-conquer algorithm as implemented by Arbenz [4, 3] is

$$\begin{aligned} \varphi_{n,p}^{\mathrm{AH}} &\approx 2k_l(4k_u+1)\frac{n}{p} + (4k_l+4k_u+1)r\frac{n}{p} \\ &\quad + \left(\frac{32}{3}k^3 + 9k^2 r + 4t_s + 4k(k+r)t_w\right)\lfloor \log_2(p-1) \rfloor. \end{aligned} \tag{2.9}$$

In the ScaLAPACK implementation, the factorization phase is separated from the forward substitution phase. This allows a user to solve several systems of equations without the need to factor the system matrix over and over again. In the implementation by Arbenz, several systems can be solved simultaneously but only in connection with the matrix factorization. The close coupling of factorization and forward substitution reduces communication at the expense of flexibility of the code. In the ScaLAPACK solver the number of messages sent is higher while overall message volume and operation count remain the same,

$$\begin{aligned}\varphi_{n,p}^{\text{ScaLAPACK}} \approx\ & 2k_l(4k_u+1)\frac{n}{p} + (4k_l+4k_u+1)r\frac{n}{p} \\ & + \left(\frac{32}{3}k^3 + 9k^2r + 6t_s + 4k(k+r)t_w\right)\lfloor\log_2(p-1)\rfloor.\end{aligned} \tag{2.10}$$

The complexity for solving a system with an already factored matrix consists of the terms in (2.10) containing r, the number of right-hand sides. In (2.9) and (2.10) we have assumed that the time for the transmission of a message of length n floating point numbers from one to another processor is independent of the processor distance and can be represented in the form [24]

$$t_s + nt_w .$$

t_s denotes the startup time relative to the time of a floating point operation, i.e. the number of flops that can be executed during the startup time. t_w denotes the number of floating point operations that can be executed during the transmission of one word, here a (8-Byte) floating point number. Notice that t_s is much larger than t_w. On the Intel Paragon, for instance, the transmission of m bytes takes about $0.11 + 5.9\cdot 10^{-5}m$ msec. The bandwidth between applications is thus about 68 MB/s. Comparing with the 10 Mflop/s performance for the LINPACK benchmark we get $t_s \approx 1100$ and $t_w \approx 4.7$ if floating point numbers are stored in 8 bytes of memory. On the SP/2 or the SGI/Cray T3D the characteristic numbers t_s and t_w are even bigger.

Dividing (2.1) by (2.9) and by (2.10), respectively, the *speedups* become

$$S_{n,p}^{\text{AH}} = \frac{\varphi_n}{\varphi_{n,p}^{\text{AH}}}, \qquad S_{n,p}^{\text{ScaLAPACK}} = \frac{\varphi_n}{\varphi_{n,p}^{\text{ScaLAPACK}}}, \tag{2.11}$$

The processor number for which highest speedup is observed is $\mathcal{O}(n/k)$ [4]. Speedup and efficiency are relatively small, however, due to the high redundancy of the parallel algorithm. Redundancy is the ratio of the serial complexity of the parallel algorithm and the serial algorithm, i.e. it indicates the parallelization overhead with respect to floating point operations. If r is small, say $r=1$, then the redundancy is about 4 if $k_l = k_u$ and even higher otherwise [5]. If r is bigger, then the redundancy tends to 2. In Fig 2.4 speedup is plotted versus processor number for three different problem sizes as predicted by (2.11). The Paragon values for $t_s = 1000$ and $t_w = 4.7$ have been chosen. Because there are fewer messages sent in the Arbenz/Hegland implementation than in the ScaLAPACK implementation, the former can be expected to yield slightly higher speedups. However, the gain in flexibility with the ScaLAPACK routine certainly justifies the small performance loss. Notice however that the formulae given for $\varphi_{n,p}^{\text{AH}}$ and for $\varphi_{n,p}^{\text{ScaLAPACK}}$ must be considered very approximative. The assumption that all floating point operations take the same amount of time is completely unrealistic on modern RISC processors. Also, the numbers t_w and t_s are crude

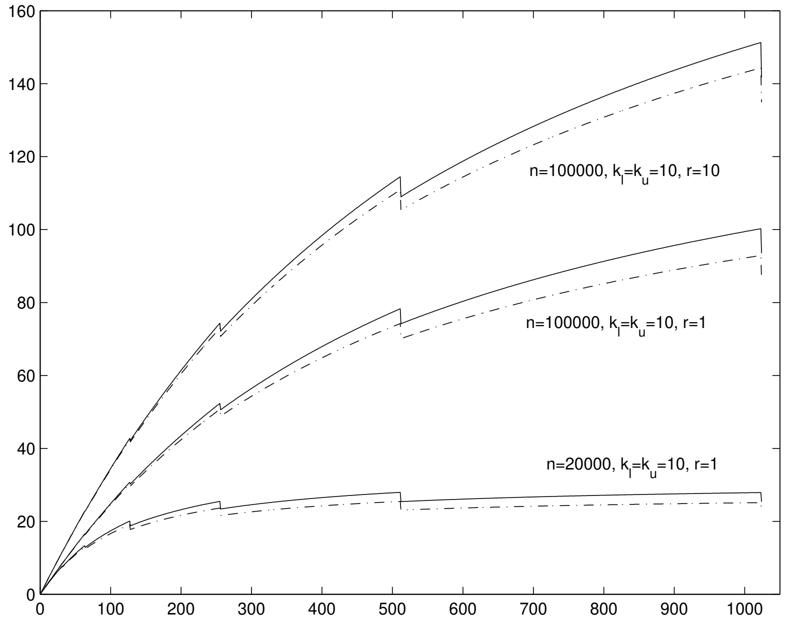


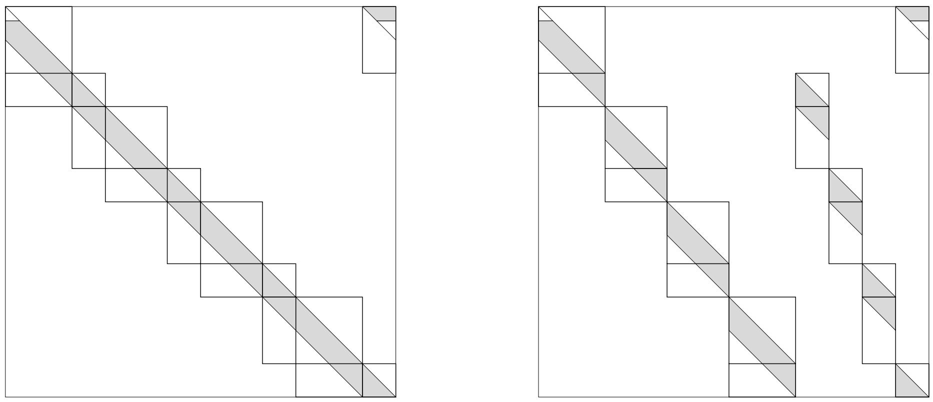
FIG. 2.4. *Speedups vs. processor numbers for various problem-sizes as predicted by* (2.11). *Drawn lines indicate the Arbenz/Hegland implementation, dashed lines the ScaLAPACK implementation.*

estimates of the reality. However, the saw-teeth caused by the term $\lfloor \log_2(p-1) \rfloor$ are clearly visible in timings [3]. The numerical experiments of section 4 will give a more realistic comparison of the implementations and will tell more on the value of the above complexity measures.

3. Parallel Gaussian elimination with partial pivoting. In this section we treat the case where A is not diagonally dominant. Then the LU factorization may not exist or its computation may be unstable. Thus, it is advisable to use partial pivoting with elimination in this case. The corresponding factorization is $PA = LU$ where P is the row permutation matrix. Pivoting requires additionally about $k_l n$ comparisons and n row interchanges. More importantly, the bandwidth of U can get as large as $k_l + k_u$. (L looses its bandedness but has still only $k_l + 1$ nonzeros per column and can therefore be stored at its original place.) The wider the band of U the higher the number of arithmetic operations and our previous upper bound for the floating point operations increases in the worst case to

$$\varphi_n^{pp} = (2k_l + 2k_u + 1)k_l n + (4k_l + 2k_u + 1)rn + \mathcal{O}((k+r)k^2), \qquad k := k_l + k_u, \quad (3.1)$$

where again r is the number of right-hand sides. This bound is obtained by counting the flops for solving a banded system with lower and upper half-bandwidth k_l and $k_l + k_u$, respectively, without pivoting. The overhead introduced by pivoting, which may be as big as $(k_l + k_u + 1)/(k_u + 1)$, is inevitable if stability of the LU factorization cannot be guaranteed. Therefore the methods for solving banded systems in packages like LAPACK incorporate partial pivoting and accept the overhead. (This is actually a deficiency of LAPACK which has been eliminated in ScaLAPACK: the memory space wasted is simply too big. Further, back-substitution can be implemented faster if it is known that there are no row interchanges.) Note that this overhead is particular for banded and sparse linear systems, it does not occur with dense matrices!



(a) (b)

FIG. 3.1. *Non-zero structure (shaded area) of (a) the original and (b) the block column permuted band matrix with* $k_l > k_u$.

The partition (2.2) is not suited for the parallel solution of (1.1) if partial pivoting is to be applied. In order that pivoting can take place independently in block columns they must not have elements in the same row. Therefore, the separators have to be $k := k_l + k_u$ columns wide, cf. Fig. 3.1(a). As discussed in detail in [6] we consider the matrix A as a *cyclic* band matrix by moving the last k_l rows to the top. Then we partition this matrix into a cyclic lower block bidiagonal matrix,

$$\begin{pmatrix} A_1 & & & & & D_1 \\ B_1 & C_1 & & & & \\ & D_2 & A_2 & & & \\ & & \ddots & \ddots & & \\ & & & D_p & A_p & \\ & & & & B_p & C_p \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \boldsymbol{\xi}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_p \\ \boldsymbol{\xi}_p \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \boldsymbol{\beta}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_p \\ \boldsymbol{\beta}_p \end{pmatrix}, \tag{3.2}$$

where $A_i \in \mathbb{R}^{m_i \times n_i}$, $C_i \in \mathbb{R}^{k \times k}$, $\mathbf{x}_i$, $\mathbf{b}_i \in \mathbb{R}^{n_i}$, $\boldsymbol{\xi}_i$, $\boldsymbol{\beta}_i \in \mathbb{R}^k$, $k := k_l + k_u$, and $\sum_{i=1}^{p} m_i = n$, $m_i = n_i + k$. If $n_i > 0$ for all i, then the degree of parallelism is p [6].

For solving $A\mathbf{x} = \mathbf{b}$ in parallel we apply a generalization of cyclic reduction that permits pivoting [19]. We again discuss the first step more closely. The later steps are similar except the matrix blocks are square. We first (formally) apply a block odd-even permutation to the columns of the matrix in (3.2). For simplicity of exposition we consider the case $p = 4$. Then, the permuted system becomes

$$\left[\begin{array}{cccc|cccc} A_1 & & & & & & & D_1 \\ B_1 & & & & C_1 & & & \\ & A_2 & & & D_2 & & & \\ & B_2 & & & & C_2 & & \\ & & A_3 & & & D_3 & & \\ & & B_3 & & & & C_3 & \\ & & & A_4 & & & D_4 & \\ & & & B_4 & & & & C_4 \end{array}\right] \begin{pmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_4 \\ \boldsymbol{\xi}_1 \\ \vdots \\ \boldsymbol{\xi}_4 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \boldsymbol{\beta}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_4 \\ \boldsymbol{\beta}_4 \end{pmatrix}. \tag{3.3}$$

The structure of the matrix in (3.3) is depicted in Fig. 3.1(b). Notice that the permutation that moves the last rows to the top was done for pedagogical reasons: it makes the diagonal blocks A_i and C_i square and the first elimination step gets formally equal with e the successive ones. A different point of view (which leads to the same factorisation) could allow the first and the last diagonal blocks to be non-square.

The local matrices are stored in the LAPACK storage scheme for non-diagonally dominant matrices [2, 8]: in addition to the $k_l + k_u + 1$ rows that store the original portions of the matrix, an additional $k_l + k_u$ rows have to be provided for storing the fill-in. In the ScaLAPACK algorithm processor i stores the blocks A_i, B_i, C_i, D_{i+1}. In the Arbenz/Hegland implementation processor i stores A_i, B_i, C_i, and D_i. It is assumed that D_i^T is stored in an extra k-by-n_i array. The ScaLAPACK algorithm constructs this situation by an initial communication step that consumes a negligible fraction of the overall computing time, as in the discussion in the previous section. In both algorithms, the blocks B_p, C_p, and D_1 do not really appear but are incorporated into A_p.

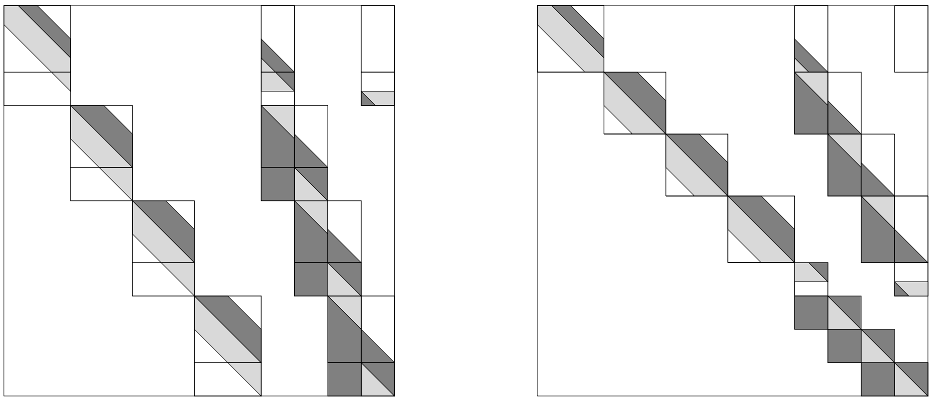
Let

$$P_i \begin{bmatrix} A_i \\ B_i \end{bmatrix} = L_i \begin{bmatrix} R_i \\ O_{k\times n_i} \end{bmatrix}, \qquad 1 \le i \le p, \tag{3.4}$$

be the LU factorizations of the blocks on the left of (3.3), and let

$$L_i^{-1} P_i \begin{bmatrix} O_{m_i\times k} \\ C_i \end{bmatrix} = \begin{bmatrix} X_i \\ T_i \end{bmatrix}, \qquad L_i^{-1} P_i \begin{bmatrix} D_i \\ O_{k\times k} \end{bmatrix} = \begin{bmatrix} Y_i \\ V_i \end{bmatrix}, \qquad L_i^{-1} P_i \begin{bmatrix} \mathbf{b}_i \\ \boldsymbol{\beta}_i \end{bmatrix} = \begin{bmatrix} \mathbf{c}_i \\ \boldsymbol{\gamma}_i \end{bmatrix}. \tag{3.5}$$

Then, we can rewrite (3.3) in the form



(a) (b)

FIG. 3.2. *Fill-in produced by GE with partial pivoting. Here $p = 4$.*

$$
\begin{bmatrix}
L_1^{-1}P_1 & & & \\
& L_2^{-1}P_2 & & \\
& & L_3^{-1}P_3 & \\
& & & L_4^{-1}P_4
\end{bmatrix}
\left[\begin{array}{cccc|cccc}
A_1 & & & & & & & D_1 \\
B_1 & & & & C_1 & & & \\
& A_2 & & & D_2 & & & \\
& B_2 & & & & C_2 & & \\
& & A_3 & & & D_3 & & \\
& & B_3 & & & & C_3 & \\
& & & A_4 & & & D_4 & \\
& & & B_4 & & & & C_4
\end{array}\right]
$$

$$
= \left[\begin{array}{cccc|cccc}
R_1 & & & & X_1 & & & Y_1 \\
O & & & & T_1 & & & V_1 \\
& R_2 & & & Y_2 & X_2 & & \\
& O & & & V_2 & T_2 & & \\
& & R_3 & & & Y_3 & X_3 & \\
& & O & & & V_3 & T_3 & \\
& & & R_4 & & & Y_4 & X_4 \\
& & & O & & & V_4 & T_4
\end{array}\right]
$$

$$
= P \left[\begin{array}{cccc|cccc}
R_1 & & & & X_1 & & & Y_1 \\
& R_2 & & & Y_2 & X_2 & & \\
& & R_3 & & & Y_3 & X_3 & \\
& & & R_4 & & & Y_4 & X_4 \\
\hline
& & & & T_1 & & & V_1 \\
& & & & V_2 & T_2 & & \\
& & & & & V_3 & T_3 & \\
& & & & & & V_4 & T_4
\end{array}\right]
\tag{3.6}
$$

P denotes odd-even permutation of the rows. The structure of the second and third matrix in (3.6) is depicted in Fig. 3.2(a) and 3.2(b), respectively. The last equation shows that again we end up with a reduced system

$$
S\boldsymbol{\xi} = \begin{pmatrix} T_1 & & & V_1 \\ V_2 & T_2 & & \\ & \ddots & \ddots & \\ & & V_p & T_p \end{pmatrix} \boldsymbol{\xi} = \boldsymbol{\gamma} \tag{3.7}
$$

with the same cyclic block bidiagonal structure as the original matrix A in (3.2).

The reduced system can be treated as before by $\lceil p/2 \rceil$ processors. This procedure is discussed in detail by Arbenz and Hegland [6].

Finally, if the vectors $\boldsymbol{\xi}_i$, $1 \leq i < p$, are known, each processor can compute its section of $\mathbf{x}$,

$$
\mathbf{x}_1 = R_1^{-1}(\mathbf{c}_1 - X_1\boldsymbol{\xi}_1 - Y_1\boldsymbol{\xi}_p), \tag{3.8}
$$

$$
\mathbf{x}_i = R_i^{-1}(\mathbf{c}_i - Y_i\boldsymbol{\xi}_{i-1} - X_i\boldsymbol{\xi}_i), \qquad 1 < i \leq p. \tag{3.9}
$$

The *back substitution phase* does not need interprocessor communication.

The parallel complexity of this algorithm is

$$
\varphi_{n,p}^{pp,AH} \approx (4k^2 + (6k-1)r)\frac{n}{p} + \left(\frac{23}{3}k^3 + 12k^2 r + 2t_s + 3k^2 t_w\right) \lfloor \log_2(p) \rfloor. \tag{3.10}
$$

in the Arbenz/Hegland implementation [6] and

$$\varphi_{n,p}^{pp,ScaLAPACK} = \varphi_{n,p}^{pp,AH} + t_s \lfloor \log_2(p) \rfloor. \tag{3.11}$$

in the ScaLAPACK implementation. We treat the blocks in S as full k-by-k blocks, as their non-zero pattern is not predictable due to the pivoting process. The speedups for these algorithms are

$$S_{n,p}^{pp,\mathrm{AH}} = \frac{\varphi_n^{pp}}{\varphi_{n,p}^{pp,\mathrm{AH}}} \qquad S_{n,p}^{pp,\mathrm{ScaLAPACK}} = \frac{\varphi_n^{pp}}{\varphi_{n,p}^{pp,\mathrm{ScaLAPACK}}}. \tag{3.12}$$

In general, the redundancy of the pivoting algorithm is only about 2 for small numbers

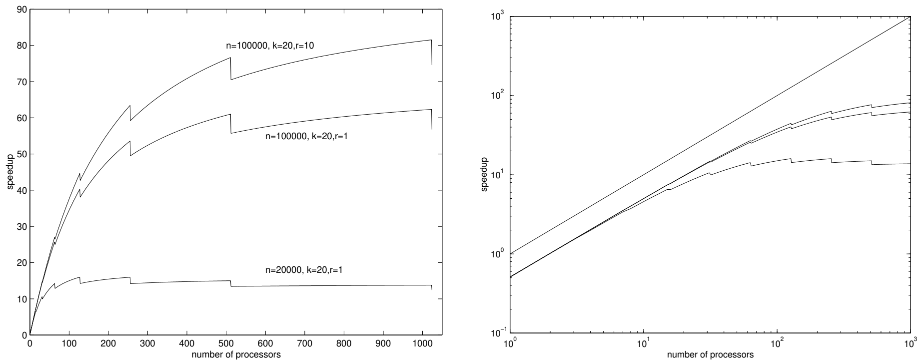


FIG. 3.3. *Speedups vs. processor numbers for various problem-sizes as predicted by* (3.12) *in linear scale (left) and doubly logarithmic scale (right).*

r of right-hand sides and around 1.5 if r is large. Note, though, that the redundancy varies according to the specific sequence of interchanges during partial pivoting. Here, we have always assumed that fill-in is the worst possible.

We make particular note of the following: since the matrix is partitioned differently than in the diagonally dominant case, this algorithm results in a different factorization than the diagonally dominant algorithm even if the original matrix A is diagonally dominant. In particular, the diagonal blocks A_i, $1 < i < p$, are treated like banded lower triangular matrices. In fact, the case of a diagonally dominant matrix is a somewhat poor case for this algorithm. This is easily seen: by reordering each diagonal block to be lower triangular, we have moved the largest elements from the diagonal and put them in the middle of each column, thus forcing interchanges for the partial pivoting algorithm when they were not necessary for the input matrix.

In Fig. 3.3 we have again plotted predicted speedups for two problem sizes and different numbers of right-hand sides. We do not distinguish between the two implementations as the t_s term is small compared with the others even for $k = k_l + k_u = 20$. The plot on the right shows the same in doubly logarithmic scale. This plot shows that the speedup is close to ideal for (very) small processor numbers and then detoriates. The gap between the actual and the ideal speedup for small processor numbers illustrates the impact of the redundancy.

Remark 1. If the original problem were actually cyclic-banded the redundancy would be 1, i.e. the parallel algorithm has (essentially) the same complexity as the serial algorithm [6]. This is so, as in this case, also the serial code generates fill-in. □

Remark 2. In (3.4) instead of the LU factorization a QR factorization could be computed [6, 20]. This doubles the computational effort but enhances stability. Similar ideas are pursued by Amestoy et al.[1] for the parallel computation of the QR factorization of large sparse matrices. □

4. Numerical experiments on the Intel Paragon. We compared the algorithms described in the previous two sections by means of three test-problems. The matrix A has all ones within the band and the number $\alpha \geq 1$ on the diagonal. The problem sizes were $(n, k_l, k_u) = (100000, 10, 10)$, $(n, k_l, k_u) = (20000, 10, 10)$, and $(n, k_l, k_u) = (100000, 50, 50)$. The condition numbers grow very large as α tends to one. Estimates of them obtained by Higham's algorithm [21, p.295] are listed in Tab. 4.1 for various values of α. The right-hand sides were chosen such that the solution was $(1, \ldots, n)^T$ which enabled us to compute the error in the computed solution. We compiled a program for each problem size, adjusting the arrays to just the needed size. When compiling we chose the highest optimization level and turned off IEEE arithmetic. IEEE arithmetic turned on lead to erratic non-reproducible execution times [4].

n	(k_l, k_u)	$\alpha = 100$	$\alpha = 10$	$\alpha = 5$	$\alpha = 2$	$\alpha = 1.01$
20000	(10,10)	1.3	9.0	4.2e+4	3.3e+6	2.9e+6
100000	(10,10)	1.3	9.0	4.3e+5	3.6e+6	3.8e+6
100000	(50,50)	2.9	1.8e+5	6.0e+6	1.8e+7	4.7e+8

TABLE 4.1
Estimated condition numbers for systems of equations solved in the above tables.

We begin with the discussion of the diagonally dominant case ($\alpha = 100$). In Tab. 4.2 the execution times are listed for all problem sizes. For the ScaLAPACK and the Arbenz/Hegland (AH) implementation the one-processors times are quite close. The difference in this part of the code is that the AH implementation calls the level-2 BLAS based LAPACK routine `dgbtf2` for the triangular factorization, whereas in the ScaLAPACK implementation the level-3 BLAS based routine `dgbtrf` is called. The latter is advantageous with the wider bandwidth $k = 50$, while `dgbtf2` performs (slightly) better with the narrow band.

The two implementations show a noteworthy difference in their two-processor performance. The ScaLAPACK implementation performs about as fast as on one processor which is to be expected. The AH implementation however looses about 20%. We attribute this loss in performance to the zeroing of auxiliary arrays that are will be used to store the fill-in ('spikes'). This is done unnecessarily in the preparation phase of the algorithm.

In ScaLAPACK, for forward elimination and backward substitution the level-2 BLAS `dtbtrs` is called. In the AH implementation this routine is expanded in order to avoid unnecessary checks if rows have been exchanged in the factorization phase. This avoids the evaluation of `if`-statements. In the AH implementation the above mentioned auxiliary arrays are stored as 'lying' blocks to further improve the scalability and to better exploit the RISC architecture of the underlying hardware [13]. The speedups of the AH implementation *relative to the 2-processor* performance is very close to ideal up to at least 64 processors. The ScaLAPACK implementation does not scale so well. For large processor numbers the difference in execution times is about 2/3 which correlates with the ratio of messages sent in the two implementations.

As indicated by (2.10) the speedups for the medium size problem are best. The

Diagonally dominant case on the Intel Paragon									
(n, k_l, k_u)	(20000, 10, 10)			(100000, 10, 10)			(100000, 50, 50)		
p	t	S	ε	t	S	ε	t	S	ε
ScaLAPACK implementation									
1	1110	1.0	4e-10	5543	1.0	5e-9	*30750*	1.0	—
2	1210	.92	4e-10	5572	1.0	4e-9	—	—	—
4	662	1.7	3e-10	2849	1.9	4e-9	25335	1.2	1e-8
8	398	2.8	2e-10	1489	3.7	3e-9	14347	2.1	8e-9
16	233	4.8	2e-10	814	6.8	2e-9	7341	4.2	6e-9
24	172	6.5	2e-10	593	9.3	2e-9	5032	6.1	5e-9
32	142	7.8	1e-10	482	12	1e-9	3890	7.9	4e-9
48	118	9.4	1e-10	379	15	1e-9	2763	11	4e-9
64	109	10	1e-10	312	18	1e-9	2211	14	3e-9
96	109	10	9e-11	243	23	8e-10	1692	18	3e-9
128	65	17	8e-11	168	33	7e-10	1390	22	2e-9
Arbenz / Hegland implementation									
1	1102	1.0	4e-10	5499	1.0	5e-9	*32734*	1.0	—
2	1369	.81	4e-10	6840	.80	4e-9	—	—	—
4	687	1.6	3e-10	3423	1.6	4e-9	22908	1.4	9e-9
8	347	3.2	2e-10	1716	3.2	3e-9	11475	2.9	7e-9
16	179	6.2	2e-10	864	6.4	2e-9	5775	5.7	5e-9
24	126	8.7	2e-10	580	9.5	2e-9	3917	8.4	2e-9
32	98	11	1e-10	438	12.6	1e-9	2975	11	4e-9
48	72	15	1e-10	296	18.6	1e-9	2065	16	3e-9
64	59	19	1e-10	228	24.1	1e-9	1598	21	3e-9
96	48	23	8e-11	159	34.6	8e-10	1161	28	2e-9
128	41	27	7e-11	124	44.3	7e-10	930	35	2e-9

TABLE 4.2

Selected execution times t in milliseconds, speedups $S = S(p)$, and error of the two implementations for the three problem sizes. ε denotes the 2-norm error of the computed solution.

$1/p$-term that containes the factorization of the A_i and the computations of the 'spikes' $D_i^U R_i^{-1}$ and $L_i^{-1} D_i^L$ consumes five times as much time as with the small problem size and scales very well. This portion is still increased with the large problem size. However, there the solution of the reduced system gets expensive also.

We now compare the performance of the ScaLAPACK and the Arbenz-Hegland implementation of the pivoting algorithm of section 3. Tables 4.3, 4.4, and 4.5 contain the respective numbers, execution time, speedup and 2-norm of the error, for the three problem sizes.

Relative to the AH implementation the execution times for ScaLAPACK comprise overhead proportional to the problem size, mainly zeroing elements of work arrays. This is done in the AH implementation during the building of the matrices. Therefore, the comparison in the non-diagonally dominant case should not be based on execution times but on speedups. Nevertheless, it should be noted that the computing time increases with the difficulty, i.e. with the condition, of the problem. They are of course hard or even impossible to predict as the pivoting procedure is unknown. At least the two problems with bandwidth $k = k_l + k_u = 20$ can be discussed along similar lines. The AH implementation scales better than ScaLAPACK. Its execution times for large processor numbers is about 2/3 of that of the ScaLAPACK implementation

Non-diagonally dominant case on the Intel Paragon. Small problem size.												
	$\alpha = 10$			$\alpha = 5$			$\alpha = 2$			$\alpha = 1.01$		
p	t	S	ε	t	S	ε	t	S	ε	t	S	ε
ScaLAPACK implementation												
1	1669	1.0	6e-10	1700	1.0	6e-8	2352	1.0	2e-6	2354	1.0	2e-7
2	1717	1.0	6e-10	1715	1.0	7e-8	1946	1.2	4e-6	1879	1.3	2e-6
4	867	1.9	4e-10	868	2.0	3e-8	982	2.4	2e-6	948	2.5	7e-7
8	455	3.7	3e-10	455	3.7	4e-8	514	4.6	1e-6	497	4.7	1e-7
16	252	6.6	3e-10	254	6.7	3e-8	283	8.3	1e-6	276	8.5	5e-7
24	184	9.1	3e-10	185	9.2	2e-8	207	11	1e-6	199	12	6e-7
32	159	11	3e-10	160	11	2e-8	177	13	5e-7	172	14	2e-7
48	113	15	3e-10	114	15	2e-8	128	18	1e-6	124	19	3e-7
64	127	13	2e-10	127	13	2e-8	138	17	4e-7	133	18	4e-8
96	124	14	2e-10	125	14	1e-8	134	18	2e-6	132	18	1e-7
128	84	20	2e-10	87	20	1e-8	94	25	2e-7	92	26	1e-7
Arbenz / Hegland implementation												
1	1329	1.0	7e-10	1362	1.0	9e-8	2030	1.0	2e-6	2033	1.0	2e-6
2	1306	1.0	6e-10	1305	1.0	3e-8	1526	1.3	4e-6	1482	1.4	3e-7
4	663	2.0	5e-10	662	2.1	5e-8	773	2.6	2e-6	750	2.7	6e-7
8	342	3.9	4e-10	342	4.0	3e-8	396	5.1	1e-6	386	5.3	3e-7
16	184	7.2	3e-10	184	7.4	1e-8	211	9.6	1e-6	206	9.9	2e-7
24	135	9.8	3e-10	135	10.1	1e-8	153	13	5e-7	149	14	1e-7
32	108	12	3e-10	108	13	2e-8	121	17	2e-7	118	17	1e-7
48	86	16	3e-10	86	16	1e-8	94	22	1e-6	93	22	1e-7
64	72	19	3e-10	73	19	1e-8	79	26	2e-7	78	26	1e-7
96	64	21	3e-10	64	21	1e-8	68	30	1e-7	68	30	1e-7
128	57	23	2e-10	57	24	1e-8	61	33	1e-6	60	34	2e-7

TABLE 4.3

Selected execution times t in milliseconds, speedups S, and 2-norm errors ε of the two implementations for the small problem size $(n, k_l, k_u) = (20000, 10, 10)$ with varying α.

again reflecting the ratio of messages sent. Notice that here the block size of the reduced system but also of the fill-in blocks ('spikes') are twice as big as in the diagonally dominant case. Therefore, the performance in Mflop/s is higher here. This plays a role mainly in the computation of the fill-in. The redundancy does not have the high weight that the flop count of the previous section indicates. In fact, the pivoting algorithm performs almost as good or sometimes even better than the algorithm for the diagonally dominant case. This may suggest to *always* use the former algorithm [5]. This consideration is correct with respect to computing time. It must however be remembered that the pivoting algorithm requires twice as much memory space as the algorithm for the diagonally dominant case. (In the serial algorithm the ratio is only $(2k_l + k_u)/(k_l + k_u)$.) In any case, the overhead for pivoting in the solution of the reduced system by bidiagonal cyclic reduction is not so big that it justifies sacrificing stability.

The picture is different for the largest problem size. Here, ScaLAPACK scales quite a bit better than the implementation by Arbenz and Hegland. The reduction of the number of messages and marshaling overhead without regard to the message volume is counterproductive here. With the wide band, the volume of the message times t_w by far outweighs the cumulated startup-times, cf. (3.10). So, for the largest processor numbers ScaLAPACK is fastest and yields the highest speedups.

Non-diagonally dominant case on the Intel Paragon. Intermediate problem size.												
	$\alpha = 10$			$\alpha = 5$			$\alpha = 2$			$\alpha = 1.01$		
p	t	S	ε	t	S	ε	t	S	ε	t	S	ε
ScaLAPACK implementation												
1	8331	1.0	7e-9	8489	1.0	9e-7	11759	1.0	2e-5	11756	1.0	1e-5
2	8528	1.0	7e-9	8516	1.0	2e-6	9689	1.2	2e-5	9327	1.3	6e-6
4	4277	1.9	6e-9	4274	2.0	2e-6	4856	2.4	1e-5	4684	2.5	5e-6
8	2157	3.9	5e-9	2156	3.9	1e-6	2448	4.8	8e-6	2365	5.0	5e-6
16	1103	7.6	3e-9	1103	7.7	1e-6	1251	9.4	8e-6	1210	9.7	1e-6
24	770	11	3e-9	771	11	9e-7	870	14	5e-6	842	14	1e-6
32	585	14	2e-9	588	14	1e-6	663	18	4e-6	642	18	1e-6
48	429	19	2e-9	431	20	8e-7	481	24	6e-6	450	26	1e-6
64	338	25	2e-9	342	25	7e-7	382	31	5e-6	370	32	7e-7
96	264	32	2e-9	268	32	3e-7	296	40	2e-6	290	42	5e-7
128	188	44	1e-9	193	44	4e-7	215	55	2e-6	206	57	8e-7
Arbenz / Hegland implementation												
1	6645	1.0	8e-9	6811	1.0	2e-6	10158	1.0	2e-5	10178	1.0	7e-6
2	6499	1.0	8e-9	6493	1.0	7e-7	7614	1.3	2e-5	7418	1.4	9e-6
4	3260	2.0	6e-9	3257	2.1	7e-7	3815	2.7	1e-5	3715	2.7	2e-6
8	1640	4.1	5e-9	1639	4.2	6e-7	1917	5.3	9e-6	1869	5.4	2e-6
16	834	8.0	4e-9	833	8.2	1e-6	971	11	5e-6	949	11	2e-6
24	569	12	3e-9	569	12	5e-7	660	15	5e-6	643	16	1e-6
32	433	15	3e-9	433	16	5e-7	501	20	4e-6	490	21	2e-6
48	303	22	2e-9	302	23	4e-7	348	29	4e-6	340	30	1e-6
64	236	28	2e-9	235	29	6e-7	269	38	5e-6	263	39	4e-7
96	173	38	2e-9	172	40	2e-7	195	52	2e-6	191	53	6e-7
128	139	48	2e-9	139	49	4e-7	155	66	6e-6	153	67	9e-7

TABLE 4.4

Selected execution times t in milliseconds, speedups S, and 2-norm errors ε of the two implementations for the medium problem size $(n, k_l, k_u) = (100000, 10, 10)$ with varying α.

5. Conclusion. We have shown that the algorithms implemented in ScaLAPACK are stable and perform reasonably well. The comparison with the implementations of the same algorithms by Arbenz and Hegland that are designed to reduce the number of messages that are communicated are faster for very small bandwidth. The difference is however not too big. The flexibility and versatility of the ScaLAPACK justifies the loss in performance.

Nevertheless, it may be useful to have in ScaLAPACK a routine that combines the factorization and solution phase. Appropriate routines would be the 'drivers' `pddbsv.f` for the diagonally dominant case and `pdgbsv.f` for the non-diagonally dominant case. In the present version of ScaLAPACK, the former routine consecutively calls `pddbtrf.f` and `pddbtrs.f`, the latter calls `pdgbtrf.f` and `pdgbtrs.f`, respectively. The storage policy could stay the same. So, the flexibility in how to apply the routines remains.

We found that the pivoting algorithm does not imply a large computational overhead over the solver for the diagonally dominant systems of equations. We even observed shorter solution times in some cases. However, as the pivoting algorithm requires twice as much memory space it should only be used in uncertain situations.

Non-diagonally dominant case on the Intel Paragon. Large problem size.												
	$\alpha = 10$			$\alpha = 5$			$\alpha = 2$			$\alpha = 1.01$		
p	t	S	ε	t	S	ε	t	S	ε	t	S	ε
ScaLAPACK implementation												
1	*41089*	1.0	—	*45619*	1.0	—	*68553*	1.0	—	*68737*	1.0	—
8	24540	1.7	2e-6	24524	1.9	7e-5	30820	2.2	2e-4	27857	2.5	8e-5
16	12931	3.2	3e-6	12926	3.5	3e-5	16000	4.3	1e-4	14567	4.7	8e-5
24	9035	4.5	2e-6	9020	5.1	3e-5	11053	6.2	1e-4	10112	6.8	8e-5
32	7319	5.6	2e-6	7305	6.2	1e-5	8790	7.8	1e-4	8111	8.5	8e-5
48	5313	7.7	1e-6	5309	8.6	5e-6	6255	11	1e-4	5830	12	8e-5
64	4670	8.8	2e-6	4665	9.8	6e-6	5345	13	1e-4	5034	14	2e-4
96	3690	11	1e-6	3680	12	1e-5	4101	17	1e-4	3926	18	2e-5
128	3470	12	1e-6	3459	13	1e-5	3744	18	1e-4	3632	19	2e-5
Arbenz / Hegland implementation												
1	*36333*	1.0	—	*40785*	1.0	—	*64100*	1.0	—	*64308*	1.0	—
8	21598	1.7	2e-6	21647	1.9	6e-6	27929	2.3	1e-4	24837	2.6	3e-4
16	11734	3.1	2e-6	11767	3.5	2e-6	14863	4.3	1e-4	13334	4.8	5e-5
24	8737	4.2	1e-6	8740	4.7	3e-6	10787	5.9	2e-4	9777	6.6	1e-4
32	7019	5.2	2e-6	7023	5.8	9e-6	8537	7.5	1e-4	7798	8.2	1e-4
48	5716	6.4	2e-6	5721	7.1	2e-5	6704	9.6	1e-4	6208	10	8e-5
64	4858	7.5	1e-6	4855	8.4	5e-6	5575	12	1e-4	5226	12	6e-6
96	4415	8.2	1e-6	4402	9.3	6e-6	4861	13	2e-4	4648	14	8e-5
128	3981	9.1	7e-7	3973	10	1e-5	4301	15	8e-5	4149	16	3e-5

TABLE 4.5

Selected execution times t in milliseconds, speedups S, and 2-norm errors ε of the two implementations for the large problem size $(n, k_l, k_u) = (100000, 50, 50)$ with varying α. The single processor execution times (in italics) have been estimated.

REFERENCES

[1] P. R. AMESTOY, I. S. DUFF, AND C. PUGLISI, *Multifrontal QR factorization in a multiprocessor environment*, Numer. Linear Algebra Appl., 3 (1996), pp. 275–300.

[2] E. ANDERSON, Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. D. CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, S. OSTROUCHOV, AND D. SORENSEN, *LAPACK Users' Guide - Release 2.0*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1994. (Software and guide are available from Netlib at URL `http://www.netlib.org/lapack/`).

[3] P. ARBENZ, *On experiments with a parallel direct solver for diagonally dominant banded linear systems*, in Euro-Par '96, L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, eds., Springer, Berlin, 1996, pp. 11–21. (Lecture Notes in Computer Science, 1124).

[4] P. ARBENZ AND W. GANDER, *A survey of direct parallel algorithms for banded linear systems*, Tech. Report 221, ETH Zürich, Computer Science Department, October 1994. Available at URL `http://www.inf.ethz.ch/publications/`.

[5] P. ARBENZ AND M. HEGLAND, *Scalable stable solvers for non-symmetric narrow-banded linear systems*, in Seventh International Parallel Computing Workshop (PCW'97), P. Mackerras, ed., Australian National University, Canberra, Australia, 1997, pp. P2–U–1 – P2–U–6.

[6] ———, *On the stable parallel solution of general narrow banded linear systems*, in High Performance Algorithms for Structured Matrix Problems, P. Arbenz, M. Paprzycki, A. Sameh, and V. Sarin, eds., Nova Science Publishers, Commack, NY, 1998, pp. 47–73.

[7] M. W. BERRY AND A. SAMEH, *Multiprocessor schemes for solving block tridiagonal linear systems*, Internat. J. Supercomputer Appl., 2 (1988), pp. 37–57.

[8] L. S. BLACKFORD, J. CHOI, A. CLEARY, E. D'AZEVEDO, J. DEMMEL, I. DHILLON, J. DONGARRA, S. HAMMARLING, G. HENRY, A. PETITET, K. STANLEY, D. WALKER, AND R. C. WHALEY, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997. (Software and guide are available from Netlib at URL

http://www.netlib.org/scalapack/).
[9] C. de Boor, *A Practical Guide to Splines*, Springer, New York, 1978.
[10] R. Brent, A. Cleary, M. Dow, M. Hegland, J. Jenkinson, Z. Leyk, M. Nakanishi, M. Osborne, P. Price, S. Roberts, and D. Singleton, *Implementation and performance of scalable scientific library subroutines on Fujitsu's VPP500 parallel-vector supercomputer*, in Proceedings of the Scalable High-Performance Computing Conference, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 526–533.
[11] A. Cleary and J. Dongarra, *Implementation in ScaLAPACK of divide-and-conquer algorithms for banded and tridiagonal systems*, Tech. Report CS-97-358, University of Tennessee, Knoxville, TN, April 1997. (Available as LAPACK Working Note #125 from URL http://www.netlib.org/lapack/lawns/.
[12] J. M. Conroy, *Parallel algorithms for the solution of narrow banded systems*, Appl. Numer. Math., 5 (1989), pp. 409–421.
[13] M. J. Daydé and I. S. Duff, *The use of computational kernels in full and sparse linear solvers, efficient code design on high-performance RISC processors*, in Vector and Parallel Processing – VECPAR'96, J. M. L. M. Palma and J. Dongarra, eds., Springer, Berlin, 1997, pp. 108–139. (Lecture Notes in Computer Science, 1215).
[14] J. J. Dongarra and L. Johnsson, *Solving banded systems on a parallel processor*, Parallel Computing, 5 (1987), pp. 219–246.
[15] J. J. Dongarra and A. H. Sameh, *On some parallel banded system solvers*, Parallel Computing, 1 (1984), pp. 223–235.
[16] J. Du Croz, P. Mayes, and G. Radicati, *Factorization of band matrices using level 3 BLAS*, Tech. Report CS-90-109, University of Tennessee, Knoxville, TN, July 1990. (LAPACK Working Note #21).
[17] G. H. Golub and C. F. van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 2nd ed., 1989.
[18] A. Gupta, F. G. Gustavson, M. Joshi, and S. Toledo, *The design, implementation and evaluation of a symmetric banded linear solver for distributed-memory parallel computers*, ACM Trans. Math. Softw., 24 (1998), pp. 74–101.
[19] M. Hegland, *Divide and conquer for the solution of banded linear systems of equations*, in Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 394–401.
[20] M. Hegland and M. Osborne, *Algorithms for block bidiagonal systems on vector and parallel computers*, in International Conference on Supercomputing ICS'98, D. Gannon, G. Egan, and R. Brent, eds., ACM, New York, NY, 1998, pp. 1–6.
[21] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1996.
[22] R. W. Hockney and C. R. Jesshope, *Parallel Computers 2*, Hilger, Bristol, 2nd ed., 1988.
[23] S. L. Johnsson, *Solving narrow banded systems on ensemble architectures*, ACM Trans. Math. Softw., 11 (1985), pp. 271–288.
[24] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*, Benjamin/Cummings, Redwood City CA, 1994.
[25] J. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, New York, 1998.
[26] A. Sameh and D. Kuck, *On stable parallel linear system solvers*, J. ACM, 25 (1978), pp. 81–91.
[27] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, Springer, New York, 2nd ed., 1993.
[28] S. J. Wright, *Parallel algorithms for banded linear systems*, SIAM J. Sci. Stat. Comput., 12 (1991), pp. 824–842.