Parallel Sparse LU Decomposition on a Mesh Network of Transputers

Article *in* SIAM Journal on Matrix Analysis and Applications · July 1993

DOI: 10.1137/0614059

SIAM J. MATRIX ANAL. APPL.
Vol. 14, No. 3, pp. 853–879, July 1993

PARALLEL SPARSE LU DECOMPOSITION ON A MESH NETWORK OF TRANSPUTERS*

A. FRANK VAN DER STAPPEN†‡, ROB H. BISSELING†§, AND JOHANNES G. G. VAN DE VORST†

Abstract. A parallel algorithm is presented for the LU decomposition of a general sparse matrix on a distributed-memory MIMD multiprocessor with a square mesh communication network. In the algorithm, matrix elements are assigned to processors according to the grid distribution. Each processor represents the nonzero elements of its part of the matrix by a local, ordered, two-dimensional linked-list data structure. The complexity of important operations on this data structure and on several others is analysed. At each step of the algorithm, a parallel search for a set of m compatible pivot elements is performed. The Markowitz counts of the pivot elements are close to minimum, to preserve the sparsity of the matrix. The pivot elements also satisfy a threshold criterion, to ensure numerical stability. The compatibility of the m pivots enables the simultaneous elimination of m pivot rows and m pivot columns in a rank-m update of the reduced matrix. Experimental results on a network of 400 transputers are presented for a set of test matrices from the Harwell–Boeing sparse matrix collection.

Key words. sparse matrices, LU decomposition, parallel algorithms, distributed-memory multiprocessor, transputers

AMS subject classifications. 65F05, 65F50, 65Y05

1. Introduction.

Sparse linear systems of equations need to be solved in many application areas, such as oil reservoir simulation, chemical plant modelling, and linear programming. A sparse linear system of equations has the form

$$Ax = b, \tag{1}$$

where A is an $n \times n$ sparse matrix, and x and b are vectors of length n. Vector b is given, and x is the unknown solution vector. In this paper we assume that the matrix A is nonsingular, sparse (i.e., cn of its n^2 elements have a nonzero value, with $c \ll n$), and general (i.e., A has an arbitrary, not necessarily symmetric pattern of nonzeros). The computing time and the amount of memory needed to solve (1) can be reduced greatly by exploiting the sparsity of A.

Several methods exist for solving the sparse system $Ax = b$ [12], [34]. One of these methods is based on LU decomposition [19], which is closely related to Gaussian elimination. LU decomposition produces an $n \times n$ unit lower triangular matrix L, an $n \times n$ upper triangular matrix U, and permutations π and ρ of $\{0, \ldots, n-1\}$ such that

$$A_{\pi_i,\rho_j} = (LU)_{ij} \quad \text{for all } i,j,\ 0 \le i,j < n. \tag{2}$$

Permutations π and ρ appear in this equation because rows and columns may have to be permuted during the LU decomposition to preserve sparsity and ensure numerical stability.

* Received by the editors March 5, 1991; accepted for publication (in revised form) December 18, 1991.

† Koninklijke/Shell-Laboratorium, Amsterdam, P.O. Box 3003, 1003 AA Amsterdam, the Netherlands.

‡ Present address, Department of Computer Science, Utrecht University, P.O. Box 80089, 3508 TB Utrecht, the Netherlands (frankst@cs.ruu.nl).

§ Present address, Department of Mathematics, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, the Netherlands (bisseling@math.ruu.nl).

The system $Ax = b$ can be solved in five stages:

1. Decompose A to obtain matrices L, U and permutations π, ρ satisfying (2).
2. Permute b according to $d_i = b_{\pi_i}$, $0 \leq i < n$, to obtain a vector d.
3. Solve $Ly = d$ to obtain a vector y. This unit lower triangular system of equations is solved by forward substitution.
4. Solve $Uz = y$ to obtain a vector z. This upper triangular system of equations is solved by back-substitution.
5. Permute z according to $x_{\rho_j} = z_j$, $0 \leq j < n$, to obtain the solution vector x.

This paper deals exclusively with the first stage: we present an algorithm for the LU decomposition of a sparse matrix on a distributed-memory parallel computer. A suitable algorithm for the parallel solution of sparse triangular systems can be derived from the parallel dense triangular system solving algorithm in [2]. For a symmetric positive definite matrix A, it is more efficient to use Cholesky factorisation [19] instead of LU decomposition; an extensive review of parallel sparse Cholesky factorisation algorithms is given in [21].

Sequential sparse LU decompositions usually consist of many steps, each of which contains a search of the reduced matrix for one pivot element, followed by row and column permutations, and a rank-1 update of the reduced matrix. Sparsity can be preserved during the LU decomposition by choosing appropriate pivot elements. A heuristic pivot search strategy that achieves this aim is the Markowitz strategy [24] (see [14] for an experimental evaluation of its performance). The choice of a pivot element A_{ij} leads to the creation of at most $M_{ij} = (R_i - 1)(C_j - 1)$ new nonzeros, where R_i (C_j) is the number of nonzeros in row i (column j) of the reduced matrix. The upper bound M_{ij} is called the *Markowitz count* of A_{ij} [12, Chap. 7]. A pivot element that has the lowest Markowitz count, *mincount*, is chosen. Many variants of this basic Markowitz strategy exist. Zlatev [33] limits the search for pivot elements to the sparsest three rows, to prevent long searches; his experiments show that the total number of new nonzeros created, the *fill-in*, is not necessarily reduced by searching more rows.

Numerical stability must be maintained during LU decomposition to obtain an accurate solution. Sequential general-purpose programs for sparse LU decomposition such as MA28 [11] and Y12M [35] incorporate a mechanism to prevent small matrix elements from being chosen as a pivot. One variant is to accept only those pivot candidates A_{ij} that are nonzero and that satisfy

$$|A_{ij}| \geq u \cdot \max_{l} |A_{lj}|, \tag{3}$$

where u, $0 \leq u \leq 1$, is a threshold parameter [12, Chap. 9].

The aim of this paper is to present a parallel general-purpose sparse LU decomposition algorithm that includes features such as Markowitz pivoting to preserve sparsity and threshold pivoting to ensure numerical stability. The algorithm is suitable for distributed-memory message-passing multiprocessors such as transputer networks and hypercubes. The functionality of our parallel implementation is comparable to that of sequential programs such as MA28 and Y12M.

The potential parallelism in sparse LU decomposition is twofold: first, in dense LU decompositions the operations of each rank-1 update can be done in parallel; this parallelism is inherited by sparse algorithms. Second, the sparsity of the matrix allows for the parallel execution of some computations which in the dense case have to be performed in sequence. In the sparse case, several rank-1 updates of the matrix can be combined into one multiple-rank update with many potentially simultaneous operations, thereby avoiding synchronisation and idling of processors after each rank-1 update.

Both forms of parallelism have been exploited in shared-memory parallel sparse LU decomposition algorithms of Smart and White [29], Alaghband [1], Davis and Yew [8], [9], Gallivan, Sameh, and Zlatev [17], and others. In these algorithms the Markowitz strategy is modified to obtain a set S of pivot elements that can be handled simultaneously. To make this set as large as possible, pivot elements with a Markowitz count higher than *mincount* are accepted. Calahan [5] was the first to exploit the fact that two pivot elements A_{ij} and A_{kl} can be handled simultaneously if

$$A_{il} = A_{kj} = 0. \tag{4}$$

Such pivots are called *compatible* [1] or *independent* [29]. For a detailed discussion of compatibility, see [9].

Smart and White [29] investigate the parallel time complexity of sparse LU decomposition for an unlimited number of processors, by using task graph depths as a complexity measure. They present an algorithm in which the pivot set S contains compatible diagonal elements with a Markowitz count between *mincount* and *mincount* $+ a$, where a is an input parameter. The set S is constructed by starting with the empty set and successively adding new compatible pivot elements in order of increasing Markowitz count. For a 1000×1000 tridiagonal matrix, the algorithm with $a = 2$ leads to a task graph depth of 27, which is close to the theoretical minimum of 23; in this case, the basic Markowitz strategy leads to a much higher depth of 1998. For most of the examined electronic circuit matrices, however, the improvement in depth over the Markowitz strategy was only about 50 percent.

Alaghband [1] presents an algorithm which generates candidate pivot sets and then chooses a pivot set S of maximum size; ties between sets are decided according to the minimum total Markowitz sum. The pivot search uses an $n \times n$ table that represents the mutual compatibility of diagonal pivot candidates. Pivot elements with a Markowitz count higher than a user-specified value or with a numerical absolute value lower than a user-specified threshold are discarded from S. Experimental results on the Denelcor HEP shared-memory computer show a speedup of 4.8 on eight processors, for a 144×144 electronic circuit matrix with 616 nonzeros. In this example many pivots are handled in parallel: the matrix is decomposed in 10 steps, with pivot sets of sizes $m = 72, 25, 16, 11, 6, 5, 3, 2, 2, 1, 1$.

Davis and Yew [8], [9] present a shared-memory parallel algorithm and a program, D2, which has the full functionality of programs such as MA28 and Y12M. In this algorithm, the pivot set S contains compatible elements with a Markowitz count between *mincount* and $a \cdot$ *mincount*, where a is an input parameter ($a = 4$ in the experiments). All processors search for acceptable pivot candidates, and then try to add them to the current set S. If a candidate is compatible with all the elements of S, it is added to S. Conflicts between processors that simultaneously try to add a pivot are prevented by critical sections in the program. The processor that is the first to arrive at the entry of a critical section gains access to it. This implies that operating system factors influence the pivot choice; the program is therefore nondeterministic. Experiments on an Alliant FX/8 shared-memory computer show that D2 is a median 3.9 times faster on eight processors than on a single processor, and that the sequential version of D2 is 4.3 times faster than the sequential program MA28.

Gallivan, Sameh, and Zlatev [17] (see also [34, Chap. 10]) present three shared-memory parallel versions of the sequential program Y12M [35]: Y12M1, which is based on rank-1 updates; Y12M2, which is based on rank-m updates; and Y12M3, which exploits coarse-grain parallelism created by ordering the matrix A to an upper block-triangular form (see also [16] for alternative ordering techniques). Here, the enhanced parallelism of Y12M2 compared to Y12M1 is important. In the algorithm Y12M2, an ordered set S of m pivots is formed with the property that $A_{kj} = 0$ if the element A_{ij} precedes the element A_{kl} in the ordering of the set. This relaxation of the compatibility requirement (4) allows for the creation of larger pivot sets, at the expense of a more complicated matrix update. Experimental results on an Alliant FX/80 shared-memory parallel computer with eight processors show that Y12M1 is faster than Y12M2 for matrices that are relatively dense or become so during the LU decomposition, whereas Y12M2 is faster for matrices that are very sparse and remain so. The speedup of Y12M1 is between 2.4 and 5.4 and that of Y12M2 is between 2.3 and 5.0, for a set of 27 test matrices from the Harwell–Boeing sparse matrix collection [13].

Sadayappan and Rao [26] analyse the amount of communication in sparse LU decomposition on a distributed-memory parallel computer. They present the *fragmented distribution* which splits rows and columns into parts and distributes these parts over different processors; this is in contrast to the shared-memory algorithms above that treat rows or columns as basic indivisible units. Compared to a row/column-wrapped distribution, the fragmented distribution decreases the communication volume (i.e., the total length of the messages) of dense LU decomposition by a factor of Q, for Q^2 processors. Statistics for a number of circuit simulation matrices confirm that the communication volume for sparse matrices is reduced in the same way as for dense matrices, showing up to a five-fold decrease for $Q = 8$.

Skjellum [28] presents a distributed-memory parallel algorithm that is valid for a range of data distributions, including the grid distribution defined below. The generality of this algorithm enables the user to tune the granularity of the distribution to the characteristics of a particular computer architecture. In the current implementation, the partial row pivoting strategy is used, which gives one pivot element per step. Experimental results on a Symult s2010 for a 2500×2500 random sparse matrix with $c \approx 51$ nonzeros per row show a speedup of 9.7 on a 96 processor machine, compared to a 6 processor machine.

Our distributed-memory parallel LU decomposition algorithm for sparse matrices is based on an algorithm for dense matrices [3]. The dense algorithm allocates matrix elements to processors according to the *grid distribution* [32], defined by the mapping

$$(5) \qquad A_{ij} \longmapsto \text{processor}\,(i \bmod Q, j \bmod Q) \quad \text{for all } i, j,\ 0 \le i, j < n,$$

for Q^2 processors (s, t), $0 \le s, t < Q$. This distribution splits each row i into Q *row parts*, i.e., sets of the form $\{A_{ij} : 0 \le j < n \wedge j \bmod Q = t\}$, and it also splits each column into Q *column parts.* The grid distribution is also called *scattered square decomposition* [15] and *cyclic storage* [22]; it is similar to the fragmented distribution [26].

We choose the grid distribution (5) as the distribution scheme for sparse matrices because it has an optimal load balance and a low communication complexity for LU decomposition of dense matrices [3]. The optimal load balance in all steps of the dense LU decomposition algorithm implies that in the sparse algorithm all processors are responsible for an approximately equal number of (zero or nonzero) elements. If the statistical assumption holds that every element of the matrix has an equal probability of being nonzero, it follows that the nonzero elements are spread evenly over the processors. If this assumption does not hold because nonzeros cluster at certain places in the matrix, e.g., in the lower right-hand corner or in dense submatrices, then these nonzeros are scattered over many processors, still giving a good overall load balance. Memory use is also efficient, since the scattering effect makes it unlikely that one processor runs out of memory while the others still have much storage space available. (For a theoretical analysis of one-dimensional scattering applied to an irregular computational domain, see [25].)

Figure 1 shows four snapshots of the parallel LU decomposition of the 59×59 sparse matrix IMPCOL B from the Harwell–Boeing collection. At the start (a) of the LU decomposition, the matrix has 3169 zero elements and 312 nonzero elements, and at the end (d) it has 3053 zeros and 428 nonzeros. The zero elements are shown in yellow. The nonzero elements are assigned to the processors of a 2×2 mesh, according to the grid distribution. The elements of processor $(0,0)$ are shown in blue; those of $(0,1)$ in red; those of $(1,0)$ in green; and those of $(1,1)$ in purple. The matrix is shown at the start of step k of Algorithm 1, for (a) $k = 0$; (b) $k = 18$; (c) $k = 38$; (d) $k = 59$. The horizontal lines in (b) and (c) separate the matrix elements (i,j) with $i < k$ from those with $i \geq k$, and similarly for the vertical lines. The diagonal blocks of (d) are formed during the steps of the decomposition. Each block contains the permuted pivot elements of one step. These elements are found in a search for at most $m_{\max} = 10$ compatible pivot elements. The block sizes are $m = 9, 4, 5, 8, 7, 5, 6, 6$, and nine times 1.

An alternative to the matrix-independent grid distribution is a distribution that exploits knowledge of the sparsity pattern of the matrix to obtain an optimal load balance. Unfortunately, at every step of the LU decomposition of a general sparse matrix the sparsity pattern changes due to permutations and fill-in that are unpredictable, because they depend on the pivot choice and hence on the numerical values of the matrix elements. Adjustment to the changing sparsity pattern would necessitate frequent redistribution of the matrix, which is undesirable. For this reason, we do not adopt this approach and instead we rely on statistical expectations.

The low communication complexity of the dense LU decomposition algorithm implies a low communication complexity for the sparse algorithm, provided that the same statistical assumption as above holds. This can be seen as follows. In the dense case, the broadcast of a row of length n is done by simultaneously broadcasting Q row parts of length n/Q to Q processors each, with a time complexity of $\mathcal{O}(n/Q+Q)$. (The expressions $\mathcal{O}(x)$, $\Omega(x)$, and $\Theta(x)$ denote, respectively, at most, at least, and equal to a constant times x.) Similarly, in the sparse case, the broadcast of a row of length c is done by simultaneously broadcasting Q row parts of length c/Q to Q processors each, with a time complexity of $\mathcal{O}(c/Q + Q)$. This communication complexity is less than, for instance, the $\mathcal{O}(c + Q)$ complexity of a row broadcast for the row-wrapped distribution on a square mesh of processors. (The gain can be significant even for small values of c, where Q dominates c/Q, because in our algorithm m row broadcasts can be combined into one multiple-row broadcast of complexity $\mathcal{O}(mc/Q + Q)$.) Sadayappan and Rao [26] observed a similar reduction in communication.

The pivot search strategy of a distributed-memory parallel algorithm must be kept simple, to avoid excessive communication between searching processors. The chosen data distribution strongly influences the choice of the search heuristic. In our algorithm, each column $(*, t)$ of Q processors is responsible for about n/Q matrix columns. Each processor column searches a few of its sparsest matrix columns for numerically stable pivots with low Markowitz counts. A pivot set S of size m is then constructed by starting with the empty set and adding new compatible pivot candidates in order of increasing Markowitz count, similar to the algorithm of Smart and White [29]. This is done by a parallel algorithm which uses the distributed compatibility information supplied by the distributed matrix A. The pivot set is constructed by a pipeline of Q processors, and then broadcast to all Q^2 processors. After the pivot search, the matrix is permuted according to S and then modified by a rank-m update.

An important issue in sparse matrix computations is the choice of a data structure. In this paper, we analyse the theoretical time complexity of important parallel computations on several data structures. For reasons of simplicity, we decide to represent the nonzero elements of each processor in a local, ordered, two-dimensional linked-list data structure. This data structure has been introduced by Knuth [23] for sequential sparse matrix computations; it has been used by Alaghband [1] and Skjellum [28] for parallel computations. (Davis and Yew [8], [9] use two one-dimensional doubly linked-list structures, one for rows and one for columns. Each entry in such a structure represents a block of nonzeros.)

The remainder of this paper is organised as follows. In §2 we present the details of the parallel sparse LU decomposition algorithm. In §3 we review the possible choices of a local data structure for the local part of the sparse matrix and we compare their time complexity and memory use. In §4 we present the results of numerical experiments on square meshes of up to 400 transputers, for a test set of eleven matrices from the Harwell–Boeing sparse matrix collection [13]. In §5 we draw the conclusions.

2. Parallel algorithm.

In this section we present an algorithm for the parallel LU decomposition of sparse matrices on a square processor mesh with distributed memory. The algorithm consists of a number of steps, each of which has three phases: a pivot search, row and column permutations, and an update of the reduced matrix and its row and column nonzero counts.

2.1. Outline of the parallel algorithm.

We may choose to distribute a matrix over the processors and then find a local representation for each matrix part. On the other hand, we may choose to find a representation for the entire matrix and then distribute this representation. We decide to distribute the matrix first. This choice is justified by the following arguments:

- If distribution is accomplished first, the representation details of a matrix are always local. This locality eases the implementation of operations that change the sparsity pattern of the matrix. As an example, the insertion or deletion of an element in a local representation is an operation performed by a single processor.
- If representation is accomplished first, both the matrix itself and its representation details have to be distributed. This dual requirement has the consequence that operations that change the sparsity pattern of the matrix are always of a global nature. Hence, several processors have to cooperate to perform such operations. This cooperation makes these operations unnecessarily inefficient and difficult to implement.

Matrices are distributed over a square mesh of Q^2 processors. Each processor is identified by Cartesian coordinates (s,t), with $0 \leq s,t < Q$; in what follows we shall omit these bounds on s and t for the sake of brevity. We define $grid(s,t)$ as the set of index pairs of an $n \times n$ matrix assigned to processor (s,t) according to the grid distribution (5),

$$grid(s,t) = \{ \ (i,j) \ : \ 0 \leq i,j < n \wedge i \bmod Q = s \wedge j \bmod Q = t\}. \tag{6}$$

We introduce an $n \times n$ matrix variable X, and distribute it according to the grid distribution. The permutation variables π and ρ of (2) are replicated and distributed: it turns out to be convenient to maintain a copy of π_i in all processors $(i \bmod Q, *)$, and a copy of ρ_j in all processors $(*, j \bmod Q)$. Initially $X = A$ and $\pi = \rho = \text{id}$, the identity permutation. At the end of the computation the matrix variable X contains the factor L in its strictly lower triangular part and U in its upper triangular part, and the permutation variables π and ρ contain their final values. Row nonzero counts are maintained in a vector variable R which is replicated and distributed in the same manner as π. The component R_i equals the number of nonzero elements of row i in the *reduced matrix*, which is defined as the $(n-k) \times (n-k)$ submatrix of elements $X_{ij}, k \le i, j < n$, at the start of step k of the algorithm below. (For the purpose of explanation, we assume throughout this paper that there are no accidental zeros in the computations. Therefore, the number of nonzeros in a row equals the number of entries in the sparse representation of that row.) Similarly, column nonzero counts are maintained in a vector variable C which is replicated and distributed in the same manner as ρ. The Appendix presents a formal description of the aim of the algorithm and of the relation between X, π, ρ, R, and C that is maintained throughout the algorithm. An outline of the algorithm follows.

ALGORITHM 1 (parallel sparse LU decomposition).

```
$X := A$;
$\pi := \text{id}$;
$\rho := \text{id}$;
initialise $R$ and $C$;
$k := 0$;
while $k < n$ do begin
    find pivot set $S = \{(i_r, j_r) : 0 \le r < m\}$;
    permute rows $i \in \{i : k \le i < k+m\} \cup \{i_r : 0 \le r < m\}$;
    permute corresponding $\pi_i$ and $R_i$;
    permute columns $j \in \{j : k \le j < k+m\} \cup \{j_r : 0 \le r < m\}$;
    permute corresponding $\rho_j$ and $C_j$;
    update matrix elements $\{X_{ij} : k+m \le i < n \wedge k \le j < k+m\}$;
    update matrix elements $\{X_{ij} : k+m \le i, j < n\}$;
    update row nonzero counts $\{R_i : k+m \le i < n\}$;
    update column nonzero counts $\{C_j : k+m \le j < n\}$;
    $k := k+m$
end.
```

In the notation of this outline, it is implied that each processor (s,t) performs its part of the computations on its own data. The details of the separate parts of the algorithm are presented in §§2.2–2.4. The algorithm is illustrated in Fig. 1.

2.2. Parallel pivot search. A simple and effective pivot search strategy is to choose pivot elements from a limited number of the sparsest rows or columns; see [33]. Since we choose the column-oriented stability criterion (3), it is most convenient to search columns. The pivot search consists of three parts: searching columns to find candidate pivot elements; determining mutual compatibility of candidates; and constructing a pivot set of mutually compatible elements.

2.2.1. Search for candidates. In the first part of the pivot search, columns are inspected in parallel. Processor (s,t) has one column part of each column j with $k \le j < n$ and $j \bmod Q = t$, and it participates in the search of *ncol* of these columns that have lowest nonzero counts C_j. Here, *ncol* is an input parameter. (If less than *ncol* columns remain, these are searched.) The set of columns to be searched by processor (s, t) is denoted by *SearchCols*(t); this set is available in all processors $(*, t)$. Together, the processors $(*, t)$ search the complete columns of *SearchCols*(t). A set *ColCandidates*(t) is formed which includes one optimal pivot candidate per searched column; this set becomes available in all processors $(*, t)$. An outline of the program text for processor column $(*, t)$ follows.

ALGORITHM 2 (find candidate pivot elements).

```
$ColCandidates(t) := \emptyset$;
determine $SearchCols(t)$ from $C$;
**for all** $j \in SearchCols(t)$ **do begin**
    find $r$, $k \le r < n$, such that $|X_{rj}| = \max\{|X_{lj}| : k \le l < n \wedge X_{lj} \ne 0\}$;
    $threshold := u \cdot |X_{rj}|$;
    find $i$, $k \le i < n$, such that
        $M_{ij} = \min\{M_{lj} : k \le l < n \wedge |X_{lj}| \ge threshold \wedge X_{lj} \ne 0\}$;
    $ColCandidates(t) := ColCandidates(t) \cup \{(i, j)\}$
**end**
```

The statements of Algorithm 2 are implemented as follows. The set *SearchCols*(t) is determined by using a local data structure for column nonzero counts (see [12, Chap. 9]). The index r of an element with maximum absolute value in column j is determined by first searching locally in processor (s, t), and then communicating and comparing these local maxima to obtain the index r of the global maximum. This maximum $|X_{rj}|$ is broadcast to all processors $(*, t)$. Markowitz counts M_{ij}, $(i, j) \in grid(s, t)$, are computed from locally available nonzero counts R_i, $i \bmod Q = s$, and C_j, $j \bmod Q = t$. The index i is determined in the same fashion as the index r.

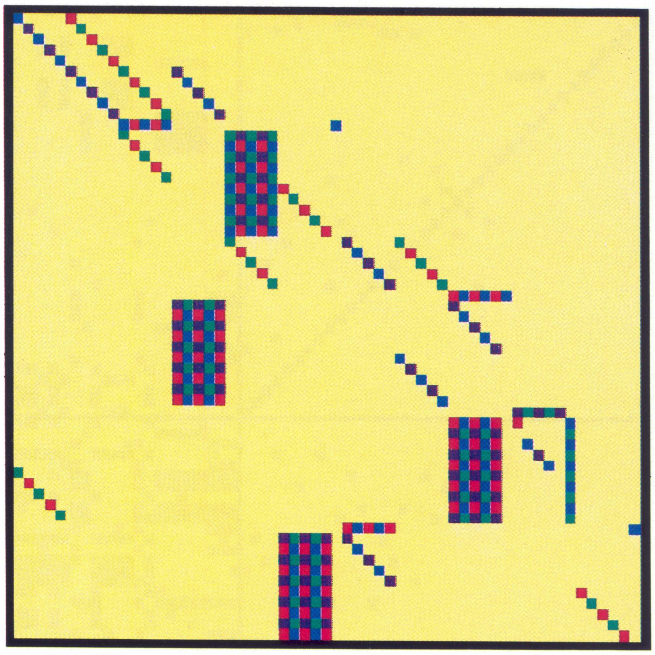
After the sets *ColCandidates*(t) are formed, they are collected into one set $Candidates = \bigcup_{t=0}^{Q-1} ColCandidates(t)$. The total number of candidates is $ncand = \min(Q \cdot ncol, n-k)$. The pivot candidates are sorted according to increasing Markowitz count, $Candidates = \{(i_r, j_r) : 0 \le r < ncand\}$, with $M_{i_r, j_r} \le M_{i_{r'}, j_{r'}}$ if $r < r'$. This ordering is used later on to give preference to candidates with low Markowitz counts. Now, candidates (i, j) that have an unacceptably high Markowitz count, $M_{ij} > a \cdot M_{i_0, j_0}$, are discarded.

The set *Candidates* is sorted during its construction by using a pipeline of processors $(0, *)$ as follows. First, each processor $(0, t)$ sorts its own set *ColCandidates*(t) by increasing Markowitz count. After that, processor $(0, t)$ inserts its candidates at the appropriate places in the ordered stream of candidates that passes by, going from processor $(0, t-1)$ to $(0, t+1)$. Processor $(0, 0)$ starts the pipeline. Processor $(0, Q-1)$ collects the ordered sequence into *Candidates* and broadcasts this set to all other processors.

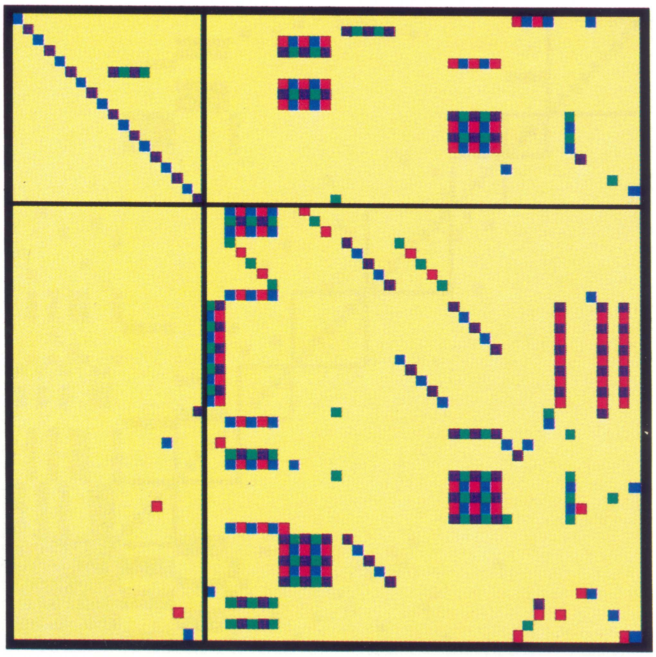
2.2.2. Compatibility of candidates.

In the second part of the pivot search, the compatibility of each pivot candidate with all other candidates is determined. The second part is separated from the third part, the construction of the pivot set, to avoid frequent synchronisation of processors, which would occur if these parts were combined. To determine compatibility, it is sufficient to inspect for each candidate (i, j) the column j, and to check whether there are nonzeros $X_{i'j}$ in rows i' that contain candidates $(i', j') \ne (i, j)$. Pivot candidate (i, j) is marked as incompatible with these candidates (i', j').

The work is distributed by letting each processor search its local part of the sparse

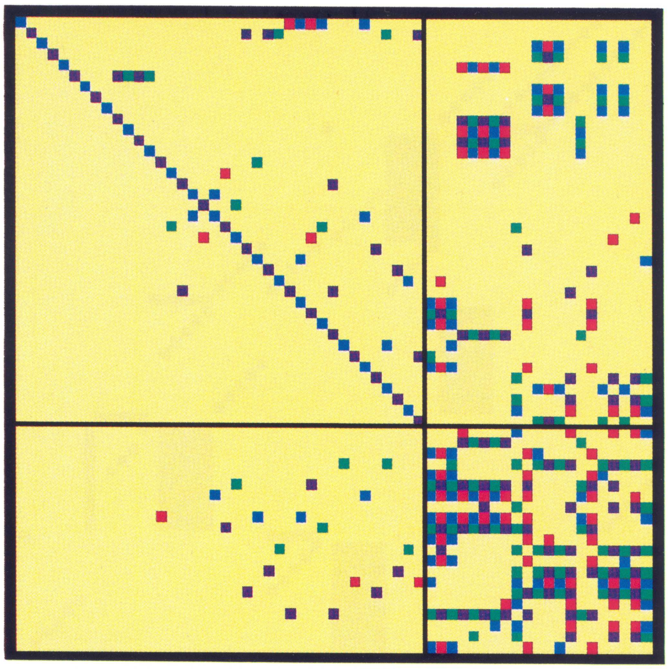


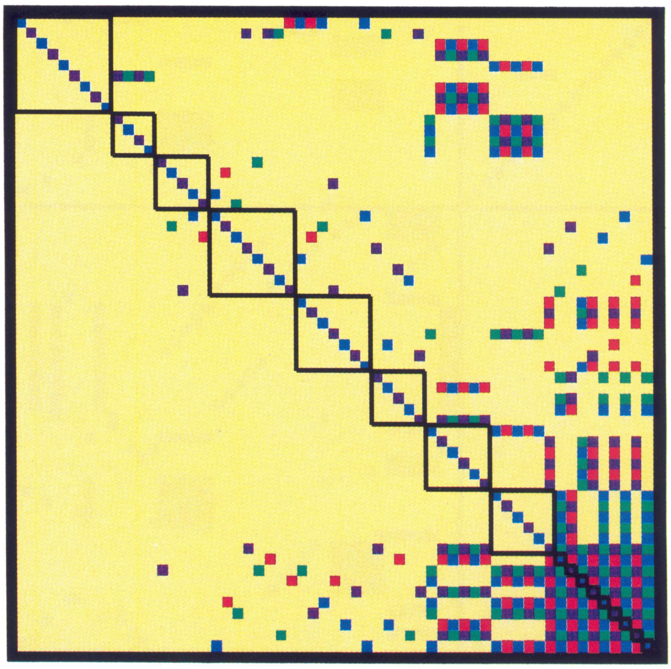
(a) $k = 0$



(b) $k = 18$

FIG. 1. *Snapshots of the sparse matrix* IMPCOL B *at step k of its parallel LU decomposition.*


(c) $k = 38$


(d) $k = 59$

matrix for nonzeros that make two candidates mutually incompatible. Processor (s,t) compares candidates from $\mathit{RowCandidates}(s) = \{(i,j) \in \mathit{Candidates} : i \bmod Q = s\}$ with candidates from $\mathit{ColCandidates}(t)$. The set $\mathit{RowCandidates}(s)$ is available in all processors $(s,*)$. In this way, the processors build a distributed incompatibility set. This set is often small, because the original matrix is sparse. Processor (s,t) stores its incompatibilities in a local set $\mathit{Incompatible}(s,t) = \{(i,j,i',j') : (i,j) \in \mathit{ColCandidates}(t) \wedge (i',j') \in \mathit{RowCandidates}(s) \wedge (i,j) \neq (i',j') \wedge X_{i'j} \neq 0\}$. An outline of the program text for processor (s,t) follows.

ALGORITHM 3 (determine compatibility of candidate pivot elements).

```
$\mathit{RowCandidates}(s) := \{(i,j) \in \mathit{Candidates} : i \bmod Q = s\}$;
$\mathit{Incompatible}(s,t) := \emptyset$;
for all $(i,j) \in \mathit{ColCandidates}(t)$ do
    for all $(i',j') \in \mathit{RowCandidates}(s)$ do
        if $(i,j) \neq (i',j') \wedge X_{i'j} \neq 0$ then
            $\mathit{Incompatible}(s,t) := \mathit{Incompatible}(s,t) \cup \{(i,j,i',j')\}$
```

2.2.3. Construction of the pivot set. In the third part of the pivot search, the pivot set S is constructed by starting with the empty set and successively adding new compatible pivot candidates (i_r, j_r) in order of increasing r. This procedure gives priority to candidates with lower Markowitz counts. The set S is constructed by a pipeline of processors which generates pivot elements, similar to a parallel sieve of Eratosthenes which generates prime numbers. Each processor is responsible for determining the inclusion in S of several pivot candidates. The length L of the pipeline can be chosen freely, between $1 \leq L \leq \min(\mathit{ncand}, Q^2)$. The choice $L = 1$ implies that all the work is done by one processor, giving an upper bound on the time complexity of $\mathcal{O}(Q^2 \cdot \mathit{ncol}^2)$, for a yield of $\mathcal{O}(\mathit{ncand}) = \mathcal{O}(Q \cdot \mathit{ncol})$ pivots. This is because in the worst case each candidate is checked for compatibility with all its predecessors. The best choice (for $\mathit{ncand} \leq Q^2$) is $L = \mathit{ncand}$, so that each processor determines the inclusion of exactly one candidate. The time complexity for this choice is $\mathcal{O}(Q \cdot \mathit{ncol})$. For practical reasons, we choose a length $L = Q$, and implement the pipeline in processor row $(0,*)$; see Algorithm 4 below. This can easily be modified, if desired. The time complexity for $L = Q$ has an upper bound of $\mathcal{O}(Q \cdot \mathit{ncol}^2)$, which is close to optimum for small ncol.

The responsibility for including a candidate (i_r, j_r) in the pivot set S or not is distributed evenly over the processors of the pipeline: each processor $(0,t)$ is responsible for at most ncol *local candidates*, i.e., the set $S(t)$ of candidates $(i_r, j_r), 0 \leq r < \mathit{ncand}$, with $\lfloor r/\mathit{ncol} \rfloor = t$. This set is needed only by processor $(0,t)$. To decide on inclusion in S, a processor $(0,t)$ needs the set $I(t)$ which contains the incompatibilities $(i_r, j_r, i_{r'}, j_{r'})$ and $(i_{r'}, j_{r'}, i_r, j_r)$ of each of the local candidates (i_r, j_r) with all the preceding candidates $(i_{r'}, j_{r'})$, $r' < r$. To provide this information, each $(i_r, j_r, i_{r'}, j_{r'}) \in \mathit{Incompatible}(s,t)$ is sent from processor (s,t) to processor $(0, \lfloor \max(r,r')/\mathit{ncol} \rfloor)$, before the pipeline starts operating.

The pipeline works as follows: processor $(0,t)$ receives a sequence of pivot elements from its neighbour $(0,t-1)$ and sends these elements to its neighbour $(0,t+1)$. If a pivot element from the sequence is incompatible with a local candidate, that local candidate is eliminated. After the pivot elements have passed, processor $(0,t)$ treats the sequence of remaining local candidates in a similar manner. The pipeline is started by processor $(0,0)$, and the m pivot elements of S are collected by processor $(0, Q-1)$ and then broadcast to all processors. An outline of the program text for processor $(0, t)$ follows.

ALGORITHM 4 (construct pivot set).

```
**if** $t = Q - 1$ **then** $S := \emptyset$;
$S(t) := \{(i_r, j_r) : 0 \le r < ncand \wedge \lfloor r/ncol \rfloor = t\}$;
**while** a pivot element $(i, j)$ is received from $(0, t-1)$ **do begin**
    **if** $t < Q - 1$ **then** send $(i, j)$ to $(0, t+1)$
                   **else** $S := S \cup \{(i, j)\}$;
    $S(t) := S(t) \setminus \{(i', j') : (i, j, i', j') \in I(t) \vee (i', j', i, j) \in I(t)\}$
**end**;
**while** $S(t) \ne \emptyset$ **do begin**
    $(i, j) := (i_l, j_l)$ with $l = \min\{r : (i_r, j_r) \in S(t)\}$;
    $S(t) := S(t) \setminus \{(i, j)\}$;
    **if** $t < Q - 1$ **then** send $(i, j)$ to $(0, t+1)$
                   **else** $S := S \cup \{(i, j)\}$;
    $S(t) := S(t) \setminus \{(i', j') : (i, j, i', j') \in I(t) \vee (i', j', i, j) \in I(t)\}$
**end**
```

2.3. Parallel permutations. Rows and columns of the matrix X can be permuted *implicitly* [28], [31] by using the permutations π and ρ to access the matrix indirectly, or *explicitly* [3], [6], [18] by moving rows and columns in the matrix. Chu and George [6] show that explicit permutation leads to a good load balance for parallel dense LU decomposition with a row-wrapped distribution. Numerical experiments of Geist and Romine [18] confirm that the gain in load balance more than offsets the incurred increase in communication time.

For dense LU decomposition with partial pivoting, the grid distribution with explicit permutation leads to an optimal load balance [3], irrespective of the choice of pivots. The load balance for implicit permutation, however, hinges on the randomising effect of the particular pivot sequence. For sparse LU decomposition with the grid distribution, explicit permutation guarantees that the row and column parts of the reduced matrix are evenly distributed over the processors. Therefore, the computational workload is well balanced if the assumption holds that the nonzeros of the matrix are evenly distributed over the row and column parts. Because of these considerations, we decide to permute rows and columns explicitly.

The aim of the row and column permutations in the matrix X is to create an $m \times m$ diagonal submatrix of elements $X_{ij}, k \le i, j < k + m$, with the m elements of the pivot set S on the diagonal; see Fig. 1(d). Because of the compatibility of the pivot elements this can be achieved, for instance, by moving the rows i_r, $0 \le r < m$, into position $k + r$ and moving the rows of $\{i : k \le i < k + m\} \setminus \{i_r : 0 \le r < m\}$ in some arbitrary order into the vacated positions $i_r \ge k + m$. The columns should be treated accordingly. Note that there is some freedom in determining the row permutation, particularly for large m; this may be exploited by a heuristic strategy which keeps row movements local, or even tries to avoid them (if $k \le i_r < k + m$).

The row permutation involves at most the rows of $\{i : k \le i < k + m\} \cup \{i_r : 0 \le r < m\}$; all the other rows remain in place. Note that the intersection of both sets may not be empty. The source and destination indices of the rows to be moved are stored in arrays *Src* and *Dest* of length *ndest*. Source processor (s, t) sends its part of row $Src(r)$ to destination processor $(Dest(r) \bmod Q, t)$, for each index r, $0 \le r < ndest$, with $Src(r) \bmod Q = s$. These communications are most efficiently implemented by operating $2Q$ pipelines in parallel, one upwards and one downwards for each processor column $(*, t)$. Data are injected into the appropriate pipeline by the source processor; they flow downstream until they are extracted by the destination processor. After this communication phase, processor (s, t) performs the following assignments.

```
ALGORITHM 5 (permute rows and row nonzero counts).
  for all $r : 0 \le r < ndest \wedge Dest(r) \bmod Q = s$ do begin
    for all $j : 0 \le j < n \wedge j \bmod Q = t \wedge (X_{Src(r),j} \ne 0 \vee X_{Dest(r),j} \ne 0)$
      do $X_{Dest(r),j} := X_{Src(r),j}$;
    $\pi_{Dest(r)} := \pi_{Src(r)}$;
    $R_{Dest(r)} := R_{Src(r)}$
  end
```

2.4. Parallel updates. An algorithm for processor (s, t), which updates the matrix, follows.

```
ALGORITHM 6 (update matrix).
  broadcast $\{X_{jj} : k \le j < k+m \wedge j \bmod Q = t\}$ from $(t,t)$ to $(*,t)$;
  for all $i, j : k+m \le i < n \wedge k \le j < k+m \wedge (i,j) \in grid(s,t) \wedge X_{ij} \ne 0$
    do $X_{ij} := X_{ij}/X_{jj}$;
  broadcast $\{X_{il} : k+m \le i < n \wedge k \le l < k+m \wedge (i,l) \in grid(s,t) \wedge$
    $X_{il} \ne 0\}$ from $(s,t)$ to $(s,*)$;
  broadcast $\{X_{lj} : k \le l < k+m \wedge k+m \le j < n \wedge (l,j) \in grid(s,t) \wedge$
    $X_{lj} \ne 0\}$ from $(s,t)$ to $(*,t)$;
  for all $i, j, l : k+m \le i, j < n \wedge (i,j) \in grid(s,t) \wedge k \le l < k+m \wedge$
    $X_{il} \ne 0 \wedge X_{lj} \ne 0$ do $X_{ij} := X_{ij} - X_{il}X_{lj}$
```

After the matrix update, the row and column nonzero counts must be adjusted because of the decreasing size of the reduced matrix and the creation of new nonzeros. It is sufficient to adjust the nonzero counts of those rows $i \ge k+m$ that have a nonzero entry X_{il} in a column l, $k \le l < k+m$, and of those columns $j \ge k+m$ that have a nonzero entry X_{lj} in a row l, $k \le l < k+m$. Nonzero counts of the other rows are not affected (for $i \ge k+m$) or are not needed any more (for $i < k+m$), and similarly for the column counts. The local data structure of column nonzero counts that is used to produce $SearchCols(t)$ (see §2.2.1) must be adjusted accordingly.

The bulk of the computing work of the LU decomposition is formed by the main matrix-update loop; see the last statement of Algorithm 6. An efficient implementation strongly depends on the data structure used to represent the sparse matrix. This is the main subject of §3.

3. Local data structures.

In this section we analyse a number of candidate data structures for the local representation of a grid part of a sparse matrix. The candidates we consider are sparse data structures, which store only nonzeros. This leads to efficient use of computing time and memory. In our efficiency considerations, computing time is of primary importance, and memory use is of secondary importance. (Usually there is plenty of memory available on distributed-memory parallel computers.)

3.1. Description of data structures. Grid parts of sparse matrices are sparse matrices themselves. Therefore, the data structure for a grid part of a sparse matrix can be chosen from the large variety of data structures that are used in sequential sparse matrix algorithms. Here, we consider several simple data structures, and describe them by Pascal-like type definitions. (A formal treatment of useful data structures is given in [30].) All data structures enable easy access to rows as well as to columns, which is required for sparse LU decomposition.

We examine data structures for the $\hat{n} \times \hat{n}$ sparse matrix $\hat{A}$ which represents the local grid part of the matrix A. Here, $\hat{n} = n/Q$. To simplify the analysis, we assume that $n \bmod Q = 0$. Local variables and indices are hatted, to distinguish them from global ones. For processor (s,t), the relation between $\hat{A}$ and A is given by

$$\hat{A}_{\hat{\imath}\hat{\jmath}} = A_{\hat{\imath}Q+s,\hat{\jmath}Q+t} \quad \text{for all } \hat{\imath},\hat{\jmath},\ 0 \le \hat{\imath},\hat{\jmath} < \hat{n}. \tag{7}$$

The data structures are:

1. *Gustavson's data structure* [20], *unordered.* The nonzeros of $\hat{A}$ are represented by array *entries*. The maximum number of nonzeros that can be stored is *bnd*. For each column, the nonzeros are stored in a contiguous block of *entries*, in arbitrary order. Between the blocks there can be empty space. For column cj of $\hat{A}$, *colfirst*$[cj]$ gives the start of the block, and *lencol*$[cj]$ its length. For each nonzero of column cj, both its row index $\hat{\imath}$ and its numerical value a are stored. In addition, there is a row data structure which is similar to the column data structure, except that numerical values of nonzeros are not stored.

```
type matrix = record
                rowfirst, colfirst : array [0..$\hat{n}-1$] of 0..$bnd-1$;
                lenrow, lencol     : array [0..$\hat{n}-1$] of 0..$\hat{n}$;
                $\hat{\jmath}$                  : array [0..$bnd-1$] of 0..$\hat{n}-1$;
                entries            : array [0..$bnd-1$] of entry
              end;
     entry  = record
                $\hat{\imath}$ : 0..$\hat{n}-1$;
                a : real
              end;
```

2. *Gustavson's data structure, ordered.* This data structure is the same as data structure 1, except that the nonzeros within each row and column are ordered by increasing index.

3. *Two-dimensional linked-list structure, unordered.* This dynamic data structure links the nonzeros of a row or column into a list, in arbitrary order. The headers of the $\hat{n}$ linked lists that correspond to the rows (columns) are represented by array *rows* (*cols*). Each entry has a pointer in its *next*$\hat{\jmath}$-field to another nonzero in the same row, and a pointer in its *next*$\hat{\imath}$-field to another nonzero in the same column.

```
type matrix       = record
                      rows, cols : array [0..$\hat{n}-1$] of entrypointer
                    end;
     entrypointer = $\uparrow$ entry;
     entry        = record
                      $\hat{\imath},\hat{\jmath}$            : 0..$\hat{n}-1$;
                      a              : real;
                      next$\hat{\imath}$, next$\hat{\jmath}$ : entrypointer
                    end;
```

4. *Two-dimensional linked-list structure, ordered.* This is the same data structure as the previous one, except that the nonzeros within each row and column list are ordered by increasing index. This data structure is similar to the orthogonal

linked list structure of Knuth [23]. The related Curtis–Reid data structure [7] can be obtained by leaving out the $\hat{\imath}$- and $\hat{\jmath}$-index of each entry and labelling the end of each row (column) list with the corresponding row (column) index. The Curtis–Reid data structure saves memory at the expense of an increase in computing time. We do not consider this data structure here, because memory use is not our primary concern.

5. *Two-dimensional doubly linked-list structure, unordered.* The idea of data structure 3 is carried further by representing the nonzeros of a row or column in a doubly linked list. This extension facilitates the deletion of an entry. We obtain data structure 5 by modifying the definition of *entry* in data structure 3:

```
type entry = record
                 $\hat{\imath}, \hat{\jmath}$                          : $0..\hat{n}-1$;
                 $a$                             : real;
                 $prev\hat{\imath}, prev\hat{\jmath}, next\hat{\imath}, next\hat{\jmath}$ : entrypointer
             end;
```

6. *Two-dimensional doubly linked-list structure, ordered.* This is the same data structure as the previous one, except that the nonzeros within each row and column list are ordered by increasing index.

3.2. Computing time. We analyse the time complexity of a number of important computations for all data structures. Since the matrix is distributed over the processors, these computations are also distributed. Generally, a distributed computation includes some computation on the local matrix part by each processor, and some communication between the processors. Communication of rows and columns requires retrieval of these rows and columns from the matrix, followed by send and receive operations. Since the retrieval is similar to a local computation that is discussed below (multiple-row assignment) and since the send and receive operations are independent of the data structure, it is sufficient only to consider local computations. We examine the following local computations:

- *Multiple-row assignment* (see Algorithm 5).
```
for all $r : 0 \le r < ndest \wedge Dest(r) \bmod Q = s$ do
    for all $j : 0 \le j < n \wedge j \bmod Q = t \wedge (X_{Src(r),j} \neq 0 \vee X_{Dest(r),j} \neq 0)$
      do $X_{Dest(r),j} := X_{Src(r),j}$
```
- *Multiple-column division* (see Algorithm 6).
```
for all $i,j : k+m \le i < n \wedge k \le j < k+m \wedge (i,j) \in grid(s,t) \wedge X_{ij} \neq 0$
  do $X_{ij} := X_{ij}/X_{jj}$
```
- *Rank-m update* (see Algorithm 6).
```
for all $i,j,l : k+m \le i,j < n \wedge (i,j) \in grid(s,t) \wedge k \le l < k+m \wedge$
  $X_{il} \neq 0 \wedge X_{lj} \neq 0$ do $X_{ij} := X_{ij} - X_{il}X_{lj}$
```

Each of these local computations is performed once in every step of parallel sparse LU decomposition. Other local matrix computations that are performed within a single step, such as multiple-column assignment or multiple-column pivot search, very much resemble the above computations for all our data structures.

We evaluate the performance of the data structures by counting the number of operations in a local (sequential) computation by processor (s,t). We present a worst-case analysis, which gives upper bounds $\mathcal{O}(x)$ on the number of operations. The operation counts are expressed in global constants m and Q and in local constants $\hat{c}$ and $\hat{m}$; the maximum number of nonzeros in a row or column part is denoted by $\hat{c}$, and

TABLE 1
Operation counts $\mathcal{O}(x)$ for computations on local data structures.

Data structure	Multiple-row assignment	Multiple-column division	Rank-m update
1. Gustavson unordered	$\hat{c}^2\hat{m}$	$\hat{c}m/Q$	$\hat{c}^2m$
2. Gustavson ordered	$\hat{c}^2\hat{m}$	$\hat{c}m/Q$	$\hat{c}^2m(1+\log m)$
3. 2D list unordered	$\hat{c}^2\hat{m}$	$\hat{c}m/Q$	$\hat{c}^2m$
4. 2D list ordered	$\hat{c}^2\hat{m}$	$\hat{c}m/Q$	$\hat{c}^2m(1+\log m)$
5. 2D double list unordered	$\hat{c}\hat{m}$	$\hat{c}m/Q$	$\hat{c}^2m$
6. 2D double list ordered	$\hat{c}^2\hat{m}$	$\hat{c}m/Q$	$\hat{c}^2m(1+\log m)$

the maximum number of row parts $Dest(r)$, $0 \le r < ndest$, with $Dest(r) \bmod Q = s$, is denoted by $\hat{m}$. (It can be shown that $\hat{m} \le m + \lceil m/Q \rceil$, although it is more likely that $\hat{m} \approx 2m/Q$.) The actual performance in the parallel computation will deteriorate due to load imbalance, but in a manner which is independent of the data structure chosen. Therefore, our comparison of data structures remains valid for the parallel case.

Table 1 displays the orders of the operation counts for the different data structures. In the following, we shall briefly explain these results. The multiple-row assignment can be done by deleting the nonzeros of the old rows and inserting the nonzeros of the new rows. This involves at most $2\hat{m}$ row parts of at most $\hat{c}$ nonzeros, and hence the total number of nonzeros involved is $\mathcal{O}(\hat{c}\hat{m})$. The number of operations equals the number of nonzeros for the unordered data structure 5, since matrix elements can be assigned new values in $\mathcal{O}(1)$ operations: old elements are deleted by making use of the double links; new elements are inserted at the headers of the row and column lists. For the corresponding ordered data structure 6, insertion of a new element into a row takes $\mathcal{O}(\hat{c})$ operations, since its column predecessor must be found. This gives a total of $\mathcal{O}(\hat{c}^2\hat{m})$ operations. Similar considerations hold for the other data structures.

The multiple-column division takes $\mathcal{O}(\hat{c}m/Q)$ operations, since at most $\hat{c}$ nonzeros in at most $\lceil m/Q \rceil$ column parts are modified. This is the operation count for all the data structures, because they all allow column-wise access to the numerical values of the nonzeros.

For data structures 3–6, the rank-m update is a sequence of updates of target row parts. Each target row part i is updated by subtracting from it m update row parts l, each multiplied by a scalar X_{il}. This is done by: (i) scattering the nonzeros of the target row part into an array of length $\hat{n}$ that is known to be zero; (ii) scanning the update row parts and multiplying them by a scalar, while accumulating numerical values in the array and building a list of new nonzeros to be created; and (iii) updating the matrix data structure by adjusting numerical values and inserting new nonzeros, while resetting the array to zero. This is the second approach of [12, §2.4]. (In certain cases, such as updates by one row, it may be cheaper to scatter update rows into the array, instead of the target row. This is the first approach of [12, §2.4].) For data structures 1 and 2, the rank-m update is performed by columns, because of the column-wise access to the numerical values of the matrix.

For data structures 3–6, there are m column parts that contain nonzero multipliers X_{il}. Since each column part has at most $\hat{c}$ nonzeros, the total number of nonzero multipliers and hence of update row parts to be scanned is at most $\hat{c}m$. Since each

TABLE 2
Minimum memory requirement of data structures.

Data structure	Memory requirement per processor
1, 2. Gustavson	$4n/Q + 3cn/Q^2$
3, 4. 2D list	$2n/Q + 5cn/Q^2$
5, 6. 2D double list	$2n/Q + 7cn/Q^2$

update row part has at most $\hat{c}$ nonzeros, the total number of operations of (ii) is $\mathcal{O}(\hat{c}^2 m)$. For data structures 1 and 2, the same upper bound holds.

In the unordered case, the total complexity of the rank-m update is $\mathcal{O}(\hat{c}^2 m)$, because in (iii) new elements can be inserted in $\mathcal{O}(1)$ operations. This can be done at the end of row and column blocks for data structure 1, and at the headers of row and column lists for data structures 3 and 5.

In the ordered case, the nonzeros of the updated target row part must be obtained in order of increasing column index. This can be done by a symbolic merge-sort. In the worst case, m updates of a single target row part may cause it to grow from $\hat{c}$ to $\hat{c}(m+1)$ nonzeros. The corresponding merge-sort takes $\mathcal{O}(\hat{c}m \log m)$ operations. (All logarithms in this paper have base two.) In the worst case, this growth occurs for $\hat{c}$ target rows, so that the total number of operations in the merge-sorts is $\mathcal{O}(\hat{c}^2 m \log m)$ and the total number of operations is $\mathcal{O}(\hat{c}^2 m(1 + \log m))$. (On average the differences between the data structures will be less pronounced than the last column of Table 1 suggests, because the growth rate of target rows may be less than the worst-case rate that has been assumed.)

3.3. Memory requirements. Table 2 shows the *minimum memory requirement* per processor for each data structure, i.e., the memory needed when the nonzeros are evenly distributed over the processors. We assume that the amount of memory per integer, real, and pointer is equal to 1. Memory requirements are expressed in terms of global constants n, c, and Q. The number of rows that appear in a grid part of the matrix is n/Q. The number of nonzeros in a row is assumed to be c, and the number of nonzeros in a row part is assumed to be c/Q.

All data structures require $\Omega(n/Q + cn/Q^2)$ memory. In the case of data structures 3–6, the memory needed for the n row headers and the n column headers scales with Q as $\Theta(Q^{-1})$, whereas the memory needed for the cn nonzeros scales as $\Omega(Q^{-2})$. This implies that row and column headers take up more memory in parallel computations than in sequential computations. Note that the grid distribution is still better in this respect than a row or column distribution, since in the latter case each processor would need $\Omega(n)$ memory, irrespective of Q. The scaling behaviour of data structures 1 and 2 is similar to that of the other data structures.

3.4. Discussion. Table 1 indicates that the unordered two-dimensional doubly linked-list structure 5 is superior: it outperforms all other data structures in the multiple-row assignment, and it is one of the three optimal data structures for the rank-m update. This conclusion is specific to *parallel* LU decomposition: in the sequential case explicit row permutations and hence row assignments often do not occur because permuting is done implicitly; furthermore, in the sequential case it is common to use rank-1 updates, and for $m = 1$ and $Q = 1$ the rank-m update takes $\mathcal{O}(c^2)$ operations for all data structures. Therefore, all data structures perform equally well sequentially.

A series of multiple-rank updates based on compatible pivot sets may be of benefit *even in the sequential case.* This gain is derived from updating an initial, small data structure in one large step, instead of updating a growing data structure in many small steps, each time accessing a larger and larger structure, with row length increasing from c to $c(m+1)$ in the worst case. For an unordered data structure, the sequential rank-m update has an operation count of $\mathcal{O}(c^2m)$, whereas a sequence of m rank-1 updates has a count of $\mathcal{O}(c^2m^2)$. Note that in this comparison, c is fixed as the number of nonzeros of a row *at the start* of the rank-m update or sequence of m rank-1 updates.

Table 1 shows that in all cases the time of the rank-m update dominates the time of the other computations. It also dominates the communication time: the number of communications in one step is of the order $\mathcal{O}(\hat{c}m + Q)$, since $\mathcal{O}(m)$ row parts of length $\hat{c}$ have to be communicated in each processor column of length Q, and similarly for column parts. For data structure 5, the communication count is about a factor $\hat{c}$ smaller than the operation count of the rank-m update.

The relative merit of data structures depends not only on operation counts, but also on other, sometimes machine-dependent, parameters. Linked-list data structures perform better on fast scalar processors, and hence on RISC-like architectures such as transputers. In contrast, the Gustavson data structure may perform better on vector processors.

We have chosen the conventional ordered two-dimensional linked-list structure 4 as the data structure for parallel sparse LU decomposition. Together with the grid distribution, data structure 4 has become the standard of all the parallel sparse linear algebra programs of PARPACK, a package developed at Koninklijke/Shell-Laboratorium, Amsterdam. This package includes among others LU decomposition, Cholesky factorisation, and triangular system solution. Our choice was made at a time when we did not recognise the importance of multiple-row assignments and rank-m updates in the context of parallel LU decomposition. Still, our data structure is conceptually simple, and it allows the development of programs with a reasonable amount of effort, so that it serves as an appropriate research vehicle. Also, it performs efficiently for a wide range of other algorithms, so that it is suitable as a compromise standard. As a suggestion for future research, we strongly encourage experiments with data structure 5, which according to our analysis is the most efficient data structure.

4. Experimental results.

The algorithm has been implemented in the parallel programming language occam 2 (for an introduction, see [4]) and experimental results have been obtained on a Parsytec SuperCluster FT-400 parallel computer. This machine consists of a square mesh of 400 INMOS T800-20 transputers, each with a 2 Mbyte memory. According to our measurements, a transputer performs a 64-bit floating point operation (flop), such as addition or multiplication, in $t_{\text{flop}} = 1.9\,\mu\text{s}$, and a 32-bit integer operation in $t_{\text{iop}} = 1.8\,\mu\text{s}$. A transputer sends a 64-bit real number to a neighbouring transputer in the mesh in $t_{\text{comm,real}} = 8.5\,\mu\text{s}$ and it sends a 32-bit integer in $t_{\text{comm,int}} = 6.6\,\mu\text{s}$. In the experiments, all real numbers have a length of 64 bits, and all integers have a length of 32 bits. The machine accuracy for 64-bit floating point operations is 2.2×10^{-16}. All experimental times were measured by an internal timer calibrated with a wall clock.

TABLE 3
Test set of sparse matrices.

Matrix	Origin	n	$nz(A)$	$c(A)$	$c(L\backslash U)$
IMPCOL B	Chemical engineering	59	312	5.3	7.4
WEST0067	Chemical engineering	67	294	4.4	8.7
FS 541 1	Atmospheric pollution	541	4285	7.9	30.7
STEAM2	Oil reservoir simulation	600	13760	22.9	89.0
SHL 400	Linear programming	663	1712	2.6	2.6
BP 1600	Linear programming	822	4841	5.9	8.8
JPWH 991	Electronic circuit simulation	991	6027	6.1	66.3
SHERMAN1	Oil reservoir simulation	1000	3750	3.8	26.6
SHERMAN2	Oil reservoir simulation	1080	23094	21.4	326.2
LNS 3937	Compressible fluid flow	3937	25407	6.5	107.0
GEMAT11	Optimal power flow	4929	33185	6.7	10.8

4.1. Test set of Harwell–Boeing matrices. To investigate the properties of our algorithm, we performed numerical experiments on eleven real unsymmetric assembled (RUA) matrices from the Harwell–Boeing sparse matrix collection [13]. In our test set we included matrices from diverse application fields, with widely varying sizes and nonzero densities. Fig. 1(a) shows the matrix IMPCOL B from the test set. Table 3 presents data on the test matrices: n is the matrix order; $nz(A)$ is the number of nonzeros of the matrix A before LU decomposition; $c(A)$ is the corresponding average number of nonzeros per row; $c(L\backslash U)$ is the average number of nonzeros per row of the matrix $L\backslash U$ that contains the L and U factors of A. These L and U factors were obtained by executing the parallel program on 400 processors.

4.2. Algorithm with standard parallel pivot search strategy. In this subsection we compare the parallel program running on $p = Q^2$ transputers, $1 \leq p \leq 400$, with a sequential program running on one transputer. The parallel program is an implementation of the algorithm of §2. The data structure is the ordered two-dimensional linked-list structure, i.e., data structure 4 of §3. Our standard pivot search procedure is a specific implementation of the parallel algorithm of §2.2 for $ncol = 1$. (Alternative pivot search procedures will be examined in §4.3.) Candidate pivot elements must satisfy (3) with $u = 0.1$, as recommended by Duff, Erisman, and Reid [12, Chap. 7]. Candidates are rejected on sparsity grounds if their Markowitz count is higher than four times the minimum Markowitz count of the candidates, as in the experiments of Davis and Yew [9].

The sequential program is a well-optimised version of the parallel program. It is obtained by simplifying the parallel program, removing all parallel overhead, and wherever possible exploiting the fact that $p = 1$. The parallel pivot search strategy is replaced by the common sequential strategy of searching three matrix columns for numerically acceptable pivot candidates and then choosing one candidate with the lowest Markowitz count.

The sequential and $p = 1$ times for matrices SHERMAN2 and LNS 3937 had to be obtained on a separate transputer with 16 Mbyte memory, because these problems do not fit into the 2 Mbyte memory of a transputer of the FT-400. (The maximum number of nonzeros per processor is about 70,000.) Incidentally, the separate transputer is about 1.13 times faster on sparse LU decomposition problems than a transputer

TABLE 4
Time (in s) of parallel sparse LU decomposition on p transputers.

Matrix	seq	$p = 1$	4	9	16	25	49	100	400
IMPCOL B	0.34	0.38	0.20	0.15	0.13	0.12	0.12	0.11	0.13
WEST0067	0.47	0.53	0.27	0.20	0.18	0.18	0.15	0.13	0.16
FS 541 1	18.4	18.8	6.74	4.47	2.91	2.25	1.79	1.42	1.29
STEAM2	92.9	93.9	28.2	14.1	9.31	7.15	4.82	3.59	2.58
SHL 400	2.31	2.69	1.40	1.05	0.88	0.82	0.73	0.71	0.66
BP 1600	9.97	10.7	4.46	3.30	2.52	2.20	1.94	1.79	1.74
JPWH 991	121.	131.	34.8	20.4	13.7	10.3	7.33	5.50	3.97
SHERMAN1	36.6	41.0	13.0	7.00	6.08	4.33	3.09	2.75	2.37
SHERMAN2	1668.	1592.		196.	108.	87.8	46.2	32.7	15.6
LNS 3937	2119.	2111.		261.	168.	96.7	77.2	37.9	23.7
GEMAT11	75.6	84.3	29.3	18.2	14.1	11.4	9.19	7.73	6.23

of the FT-400. Times measured on the separate transputer were multiplied by 1.13, to obtain comparable table entries. The $p = 4$ times for matrices SHERMAN2 and LNS 3937 could not be obtained, because we did not have a four-transputer network available with sufficient memory.

Table 4 shows the time T_p of parallel LU decomposition on p processors and the time T_{seq} of sequential decomposition. The scaling behaviour of T_p with p can be explained qualitatively by the expression

$$T_p = \mathcal{O}\left(\frac{c^2 n}{p} + \frac{cn}{\sqrt{p}} + \frac{\sqrt{p}n}{m}\right), \tag{8}$$

where c is the average number of nonzeros per row during the LU decomposition and m is the average rank of a matrix update. The first term represents the computing time of n/m rank-m updates, each of time $\mathcal{O}(\hat{c}^2 m(1+\log m)) \approx \mathcal{O}(c^2 m/p)$; see Table 1. (In this rough approximation the $m \log m$ term is neglected.) The second term represents mainly the time needed to communicate row and column parts in n/m steps, each of time $\mathcal{O}(\hat{c}m) \approx \mathcal{O}(cm/\sqrt{p})$; see §3.4. The third term includes the startup time of communication pipelines in n/m steps, each of time $\mathcal{O}(\sqrt{p})$. A few smaller terms have been neglected in the derivation of (8). Usually the first term dominates for small p, the second term for intermediate p, and the third term for large p. The cross-over points between these ranges are problem-dependent.

The table shows that T_p is a monotonically decreasing function of p, with the exception of the increase that occurs in moving from $p = 100$ to $p = 400$ for the matrices IMPCOL B and WEST0067. This increase is hardly surprising, since at the start of the computation there are fewer nonzeros (312 or 294) than processors (400), so there is little to compute per processor and the total time is dominated by the third term of (8), which increases with p. In the other cases either the first or the second term dominates. For example, the gain by a factor of two in computing rate for SHERMAN2 from $p = 100$ to $p = 400$ is probably due to the dominant behaviour of the second term.

The maximum speedup $S_p = T_{\text{seq}}/T_p$ achieved is $S_{400} \approx 107$ for SHERMAN2. Tables 3 and 4 show that the speedup is correlated to $c(L\backslash U)$: speedups increase with increasing $c(L\backslash U)$.

TABLE 5
Number of steps of parallel sparse LU decomposition on p transputers.

Matrix	$p=1$ seq	4	9	16	25	49	100	400
IMPCOL B	59	35	30	25	22	20	14	15
WEST0067	67	40	33	32	27	24	20	18
FS 541 1	541	318	238	212	179	154	127	104
STEAM2	600	404	337	306	257	228	220	181
SHL 400	663	332	223	171	139	105	83	45
BP 1600	822	561	465	399	347	314	268	190
JPWH 991	991	679	558	506	477	398	376	306
SHERMAN1	1000	616	430	405	327	274	250	200
SHERMAN2	1080		911	857	795	803	785	745
LNS 3937	3937		2252	2140	1830	1634	1318	1237
GEMAT11	4929	2589	1799	1389	1152	890	662	386

Table 4 shows that the difference in running time between the sequential program and the parallel program with $p = 1$ is small. The difference is due partly to parallel overhead and partly to differing pivot search strategies: the sequential program searches three columns per step, whereas the parallel program with $p = 1$ searches only $Q \cdot ncol = 1$ column. In most cases, the parallel algorithm is slower due to the parallel overhead and the lower quality (with respect to fill-in reduction) of the pivots. In two cases, SHERMAN2 and LNS 3937, the parallel algorithm is faster because these slowdown effects are more than offset by a faster search for pivots.

Table 5 shows the number of steps of the LU decomposition. This number is at least $n/\sqrt{p}$, because at most $\sqrt{p}$ compatible pivot elements are found in each step, due to our choice of $ncol = 1$. The near-ideal behaviour shown by matrices SHL 400 and GEMAT11 can be explained as follows. For a general $n \times n$ matrix with c nonzeros per row, the probability of an arbitrary element being zero is $1 - c/n$, so that the probability of two arbitrary pivot candidates being compatible is $(1 - c/n)^2$; cf. (4). This probability is even higher for pivot candidates that are chosen from the sparsest columns of the matrix, as in our pivot search strategy. Both SHL 400 and GEMAT11 have a very low nonzero density c/n, before and during LU decomposition. This implies that pivot candidates are usually compatible, so that most candidates become pivots and the number of steps is close to minimum. (Note that the number of steps depends on the ratio c/n and not on c alone.)

For all matrices, the number of steps initially decreases rapidly with increasing p, until a saturation point is reached. This point represents the situation where the algorithm proceeds through the sparse part of the matrix in a few steps of high rank m, and then handles the remaining dense part in steps of rank $m = 1$; see Fig. 1(d).

Table 6 displays the number of nonzeros and floating point operations, and the numerical error of the LU decomposition. The number of nonzeros fluctuates with p, without a clear trend, and without a clear advantage to either the sequential or the parallel program. From these results, we conclude that there is at most a limited penalty in terms of fill-in for relaxing the original Markowitz pivot search strategy.

The number of floating point operations is obtained by incrementing a counter for every flop performed, including the redundant flops that are introduced by parallelising the sequential program. The growth in flop count that can be seen for the very

TABLE 6
Number of nonzeros and floating point operations, and numerical error of parallel sparse LU decomposition on p transputers.

Matrix	Nonzeros $nz(L\backslash U)$			Floating point operations			Numerical error		
	seq	$p=16$	$p=400$	seq	$p=16$	$p=400$	seq	$p=16$	$p=400$
IMPCOL B	412	445	435	8.8e+2	1.9e+3	3.9e+3	3e−12	2e−12	3e−13
WEST0067	544	607	583	1.9e+3	4.0e+3	8.3e+3	7e−15	1e−14	3e−14
FS 541 1	13553	15056	16609	2.2e+5	3.1e+5	5.8e+5	6e−15	7e−15	3e−15
STEAM2	42869	45010	53395	1.7e+6	2.1e+6	4.1e+6	6e−9	6e−9	8e−9
SHL 400	1712	1712	1712	0	4.0e+1	2.2e+3	0	0	0
BP 1600	7424	7398	7229	2.6e+4	3.1e+4	7.2e+4	5e−12	4e−12	7e−12
JPWH 991	68587	69263	65707	6.9e+6	7.7e+6	7.1e+6	9e−12	3e−12	1e−11
SHERMAN1	27089	30989	26602	1.4e+6	1.8e+6	1.5e+6	1e−11	7e−12	2e−11
SHERMAN2	375316	348651	352301	8.0e+7	6.9e+7	8.4e+7	6e−7	8e−6	4e−6
LNS 3937	474800	448929	421090	8.3e+7	7.4e+7	7.4e+7	2e−4	1e−4	2e−3
GEMAT11	53358	54086	53093	4.8e+5	5.8e+5	9.3e+5	2e−10	4e−10	3e−11

sparse matrix SHL 400 is entirely due to such redundant computations. For denser matrices the redundancy is negligible. In most cases, increasing p leads to a limited growth in flop count.

The numerical error is defined as the maximum absolute error in the numerical solution x of a system $Ax = b$, with b chosen such that all components of the exact solution are one. The system has not been scaled. It has been solved in stages 1–5; see §1. The error shows variations of one order of magnitude, without any distinguishable trend. We attribute this behaviour to coincidence (e.g., caused by arbitrary tie breaking in the pivot search) and not to the qualities of any particular pivot strategy. In most cases the accuracy of our program is comparable to that of the programs MA28 [11] and D2 [8], [9], with the notable exception of the cases STEAM2, SHERMAN2, and LNS 3937, for which our program is less accurate (cf. [8, Table 4.3]). The accuracy can be improved by appropriately increasing u, decreasing a, and increasing *ncol*. This has been confirmed by experiments, except for LNS 3937, which could not be solved with a higher accuracy than 10^{-4}, whatever the choice of parameters. (For this matrix, however, accuracy can be improved significantly by scaling.)

4.3. Algorithm with alternative pivot search strategies. To investigate the influence of the chosen pivot strategy, we replaced the pivot search procedure by alternative search procedures. The other parts of the program remained unchanged. The first alternative is to search one column per processor column, and then choose *one* pivot with the lowest Markowitz count from the Q pivot candidates. This leads to a rank-1 update in each step, which is the usual method in sequential algorithms. The second alternative is a general implementation of the pivot search algorithm of §2.2 for arbitrary *ncol*. We present results for the case *ncol* $= 3$. Fig. 1 illustrates the LU decomposition of IMPCOL B for *ncol* $= 5$ and $Q = 2$.

Figure 2 shows the time of parallel sparse LU decomposition for the standard pivot search strategy (*ncol* $= 1$) and the two alternative strategies (rank-1, *ncol* $= 3$). We observe that the standard strategy is clearly superior to the rank-1 strategy, in particular for matrices with low density c/n, such as GEMAT11. This is due to the exploitation of sparsity-based parallelism, which is not used in the rank-1 strategy. The standard strategy is also better than the *ncol* $= 3$ strategy. We attribute this mainly to the simplicity of the pivot search with *ncol* $= 1$, which requires less communication than the search with *ncol* $= 3$. The potential gain in parallelism caused by increasing the number of pivots (which is at most $m_{\max} = \sqrt{p} \cdot ncol$), and hence increasing the rank of the updates, does not compensate for the losses incurred during the more expensive pivot search.

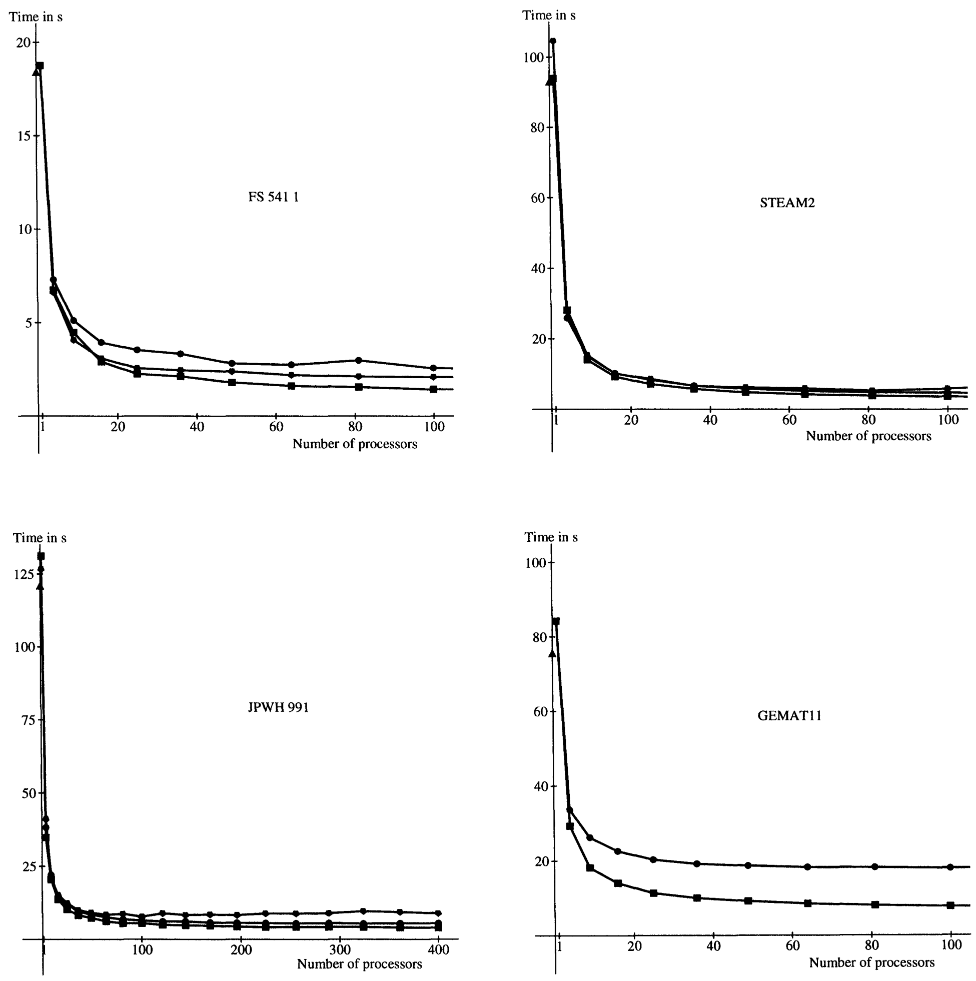


FIG. 2. *Time of parallel sparse LU decomposition for three pivot search strategies. The test matrices are* (a) FS 541 1; (b) STEAM2; (c) JPWH991; *and* (d) GEMAT11. *The squares denote the standard pivot search strategy (with ncol* $= 1$); *the pentagons the strategy with ncol* $= 3$; *and the circles the rank-*1 *strategy. The time of the sequential program is shown by a triangle. For* GEMAT11 *the curves of ncol* $= 1$ *and ncol* $= 3$ *nearly coincide, and only the first curve is shown.*

A possible application of the pivot search strategy with high *ncol* is when several matrices have to be decomposed that have an identical sparsity pattern but slightly different numerical values. In that case, the permutations π and ρ of (2) are determined during the decomposition of the first matrix. The remaining matrices are ordered according to π and ρ and then decomposed without pivot searching or permuting. This saves the parallel pivot search and the parallel permutations which may consume up to 80 per cent of the computing time. This procedure has been shown to lead to significant gains in the sequential case [12, Chap. 5], and also in the parallel case [28]. In our case, an initial investment in determining a high-quality ordering by using the alternative pivot strategy with high *ncol* (and low a and high u) would pay off handsomely in subsequent decompositions.

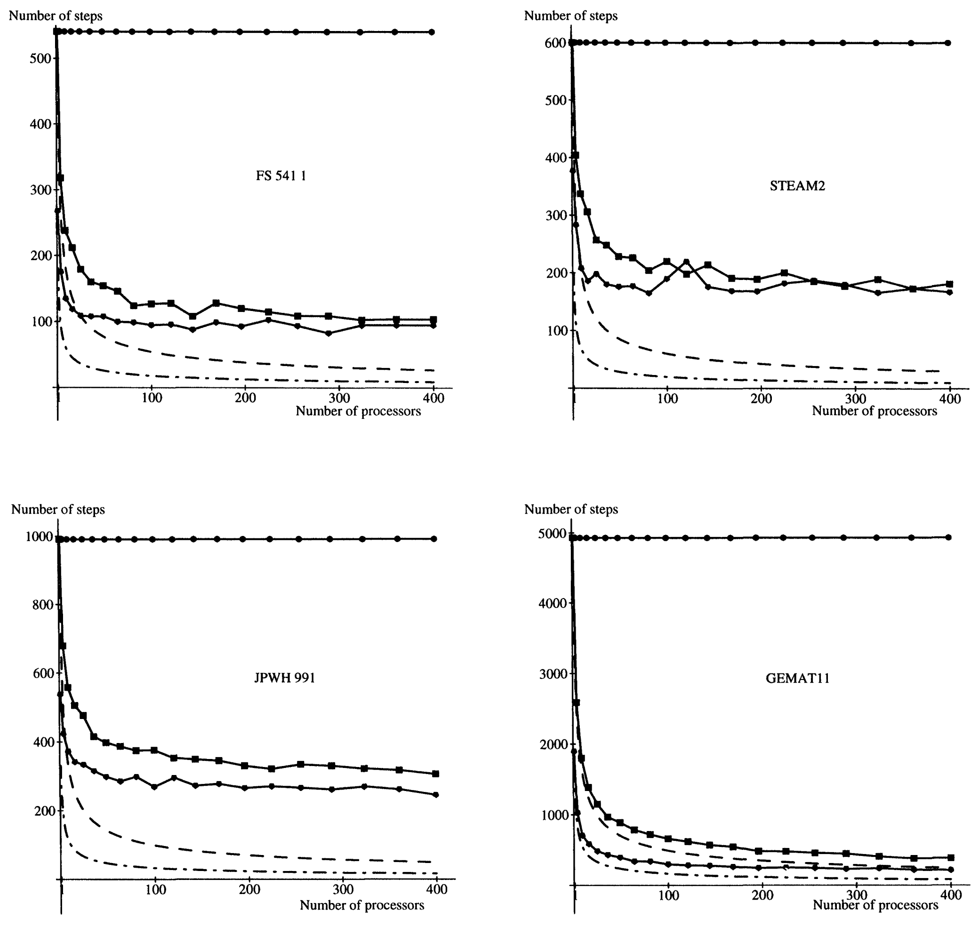


FIG. 3. *Number of steps of parallel sparse LU decomposition for three pivot search strategies. The test matrices and the marker symbols are the same as in Fig. 2. The dashed line denotes the minimum number of steps, $n/\sqrt{p}$, for the standard strategy; and the dashed-dotted line, the minimum number, $n/3\sqrt{p}$, for the ncol = 3 strategy.*

Figure 3 shows the number of steps of parallel sparse LU decomposition for the standard strategy and the two alternatives. The rank-1 strategy gives n steps in all cases. The *ncol* $= 3$ strategy gives a smaller number of steps than the standard strategy, in particular for a small number of processors. For a larger number of processors the difference between these two strategies is relatively small. Note that the curves of the matrix GEMAT11 are close to the ideal curves.

The results with respect to number of nonzeros, number of floating point operations, and numerical error of the alternative pivot strategies are similar to those of the standard strategy.

5. Conclusions.

In this paper, we have presented a scalable parallel sparse LU decomposition algorithm which is based on the grid distribution and the ordered two-dimensional linked-list data structure. The algorithm scales reasonably well with the number of processors, and it achieves a speedup of up to 107 on 400 processors for large problems, i.e., problems with a large matrix order n and a high average number c of nonzeros per row. The potential of the algorithm is best exploited in the solution of large problems, such as the larger problems in the Harwell–Boeing collection.

Our general-purpose algorithm exploits both density-based and sparsity-based parallelism. In a way, these two kinds of parallelism supplement each other: matrices with high c have many potentially simultaneous operations in each rank-1 update; matrices with low nonzero density c/n have many potentially simultaneous rank-1 updates. Matrices with high c and low c/n benefit from both kinds of parallelism. Matrices with low c but high c/n (and hence small n) offer little hope for parallelism.

The timing results of our sparse LU decomposition program on 400 transputers show that a distributed-memory parallel computer can successfully compete in the field of sparse matrix computations with today's fastest uniprocessor supercomputers. As an example, the problem SHERMAN2 is solved in 15.6 s by our program running on the FT-400, and in 34.4 s by MA28 running on one processor of the CRAY YMP/832 [27]. (This is only an indication of relative speeds, as computing speeds are obviously problem-dependent: for SHERMAN1 the CRAY YMP/832 is four times faster than the FT-400.)

Future research may lead to significant improvement of the algorithm of this paper. First, the data structure can be changed into the unordered two-dimensional doubly linked-list structure 5, which theoretically has the lowest time complexity; see §3. Second, the distributed-memory parallel sparse algorithm can be combined with a parallel dense algorithm [3] which is invoked as soon as the nonzero density of the reduced matrix exceeds a certain value and sufficient memory is available to store the reduced matrix as a dense matrix. Such a switch from a sparse to a dense program is performed in the shared-memory parallel algorithm D2 [9] (when $c/n \geq 0.2$), and in the shared-memory parallel algorithms Y12M1, Y12M2, and Y12M3 [17] (when $c/n \geq 0.1$), with often large gains. This switch prevents, for instance, extensive searching for compatible pivots when only few compatible pivots exist due to the high density of the reduced matrix. Third, several pivot search strategies can be combined: searching for large pivot sets in the first few steps of the algorithm ($ncol > 1$), then searching for smaller sets ($ncol = 1$), and after that searching for single pivots, and finally switching to a dense algorithm. (A similar combination of strategies is proposed in [17].) We expect future hybrid algorithms with appropriate switch-over criteria to be considerably faster than the current algorithm.

Appendix. Postcondition and invariant.

The aim of the algorithm for processor (s, t) is to establish the *postcondition* [10] $R[s, t]$, which is the following logical expression:

$$\begin{array}{lll} R[s,t]: & \forall_{i,j\,:\,i\le j\,\wedge\,(i,j)\in grid(s,t)} & X_{ij} = U_{ij} = A_{\pi_i,\rho_j} - \sum_{h=0}^{i-1} L_{ih}U_{hj} \\ \wedge & \forall_{i,j\,:\,i>j\,\wedge\,(i,j)\in grid(s,t)} & X_{ij} = L_{ij} = \left(A_{\pi_i,\rho_j} - \sum_{h=0}^{j-1} L_{ih}U_{hj} \right) / U_{jj}. \end{array}$$

At the start of step k of the algorithm, processor (s,t) guarantees the truth of the following *invariant* [10]:

$$\begin{array}{lll} P[s,t]: & \forall_{i,j\,:\,i,j\ge k\,\wedge\,(i,j)\in grid(s,t)} & X_{ij} = A_{\pi_i,\rho_j} - \sum_{h=0}^{k-1} X_{ih}X_{hj} \\ \wedge & \forall_{i,j\,:\,i<k\,\wedge\,i\le j\,\wedge\,(i,j)\in grid(s,t)} & X_{ij} = A_{\pi_i,\rho_j} - \sum_{h=0}^{i-1} X_{ih}X_{hj} \\ \wedge & \forall_{i,j\,:\,j<k\,\wedge\,i>j\,\wedge\,(i,j)\in grid(s,t)} & X_{ij} = \left(A_{\pi_i,\rho_j} - \sum_{h=0}^{j-1} X_{ih}X_{hj} \right) / X_{jj} \\ \wedge & \forall_{i\,:\,i\ge k\,\wedge\,i \bmod Q = s} & R_i = |\{j : k \le j < n \wedge X_{ij} \ne 0\}| \\ \wedge & \forall_{j\,:\,j\ge k\,\wedge\,j \bmod Q = t} & C_j = |\{i : k \le i < n \wedge X_{ij} \ne 0\}|. \end{array}$$

The first three subexpressions of the invariant state which partial sums have been accumulated so far. The last two subexpressions describe the nonzero counts. All invariants $P[s,t]$ hold trivially for $k = 0$, after the initialisation $X = A$ and $\pi = \rho =$ id and the appropriate initialisation of the nonzero count vectors R and C. At the end of the computation, for $k = n$, all components have their final values, so that the postcondition $R[s,t]$ is established. The algorithms works towards $R[s,t]$ by repeatedly incrementing k from 0 to n, while keeping the invariants $P[s,t]$ valid.

Acknowledgments. We are grateful to Daniël Loyens and John Somers for many stimulating discussions, and also for providing us with the Interactive Linear Algebra System (ILIAS), which greatly facilitated our numerical experiments. We thank Rob Miesen for help in plotting the figures.

REFERENCES

[1] G. Alaghband, *Parallel pivoting combined with parallel reduction and fill-in control*, Parallel Comput., 11 (1989), pp. 201–221.

[2] R. H. Bisseling and J. G. G. van de Vorst, *Parallel triangular system solving on a mesh network of transputers*, SIAM J. Sci. Statist. Comput., 12 (1991), pp. 787–799.

[3] ———, *Parallel* LU *decomposition on a transputer network*, in Lecture Notes in Computer Science 384, Springer-Verlag, New York, 1989, pp. 61–77.

[4] A. Burns, *Programming in occam* 2, Addison-Wesley, Wokingham, U.K., 1988.

[5] D. A. Calahan, *Parallel solution of sparse simultaneous linear equations*, in Proc. 11th Annual Allerton Conf. on Circuits and System Theory, 1973, pp. 729–735.

[6] E. Chu and A. George, *Gaussian elimination with partial pivoting and load balancing on a multiprocessor*, Parallel Comput., 5 (1987), pp. 65–74.

[7] A. R. Curtis and J. K. Reid, *The solution of large sparse unsymmetric systems of linear equations*, J. Inst. Math. Appl., 8 (1971), pp. 344–353.

[8] T. A. Davis, *A parallel algorithm for sparse unsymmetric* LU *factorization*, Ph. D. thesis, Tech. Rep. 907, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana, IL, Sept. 1989.

[9] T. A. Davis and P.-C. Yew, *A nondeterministic parallel algorithm for general unsymmetric sparse* LU *factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 383–402.

[10] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[11] I. S. DUFF AND J. K. REID, *Some design features of a sparse matrix code*, ACM Trans. Math. Software, 5 (1979), pp. 18–35.

[12] I. S. DUFF, A. M. ERISMAN, AND J. K. REID, *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford, U.K., 1986.

[13] I. S. DUFF, R. G. GRIMES, AND J. G. LEWIS, *Sparse matrix test problems*, ACM Trans. Math. Software, 15 (1989), pp. 1–14.

[14] A. M. ERISMAN, R. G. GRIMES, J. G. LEWIS, W. G. POOLE, JR., AND H. D. SIMON, *Evaluation of orderings for unsymmetric sparse matrices*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 600–624.

[15] G. C. FOX, M. A. JOHNSON, G. A. LYZENGA, S. W. OTTO, J. K. SALMON, AND D. W. WALKER, *Solving Problems on Concurrent Processors*, Vol. 1, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[16] K. A. GALLIVAN, B. A. MARSOLF, AND H. A. G. WIJSHOFF, MCSPARSE: *A parallel sparse unsymmetric linear system solver*, Tech. Rep. 1142, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana, IL, Aug. 1991.

[17] K. GALLIVAN, A. SAMEH, AND Z. ZLATEV, *Parallel direct method codes for general sparse matrices*, Tech. Rep. 1143, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana, IL, Oct. 1991.

[18] G. A. GEIST AND C. H. ROMINE, LU *factorization algorithms on distributed-memory multiprocessor architectures*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 639–649.

[19] G. H. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, 2nd ed., The Johns Hopkins University Press, Baltimore, MD, 1989.

[20] F. G. GUSTAVSON, *Some basic techniques for solving sparse systems of linear equations*, in Sparse Matrices and Their Applications, D. J. Rose and R. A. Willoughby, eds., Plenum Press, New York, 1972, pp. 41–52.

[21] M. T. HEATH, E. NG, AND B. W. PEYTON, *Parallel algorithms for sparse linear systems*, SIAM Rev., 33 (1991), pp. 420–460.

[22] S. L. JOHNSSON, *Communication efficient basic linear algebra computations on hypercube architectures*, J. Parallel Distrib. Comput., 4 (1987), pp. 133–172.

[23] D. E. KNUTH, *The Art of Computer Programming*, Vol. 1, 2nd ed., Addison-Wesley, Reading, MA, 1973.

[24] H. M. MARKOWITZ, *The elimination form of the inverse and its application to linear programming*, Management Sci., 3 (1957), pp. 255–269.

[25] D. M. NICOL AND J. H. SALTZ, *An analysis of scatter decomposition*, IEEE Trans. Comput., 39 (1990), pp. 1337–1345.

[26] P. SADAYAPPAN AND S. K. RAO, *Communication reduction for distributed sparse matrix factorization on a processor mesh*, in Proc. Supercomputing '89, ACM Press, New York, 1989, pp. 371–379.

[27] M. K. SEAGER, *A* SLAP *for the masses*, in Parallel Supercomputing: Methods, Algorithms and Applications, G. F. Carey, ed., John Wiley, Chichester, U.K., 1989, pp. 135–155.

[28] A. SKJELLUM, *Concurrent dynamic simulation: Multicomputer algorithms research applied to ordinary differential-algebraic process systems in chemical engineering*, Ph. D. thesis, California Institute of Technology, Pasadena, CA, May 1990.

[29] D. SMART AND J. WHITE, *Reducing the parallel solution time of sparse circuit matrices using reordered Gaussian elimination and relaxation*, in Proc. IEEE Internat. Symp. Circuits and Systems, 1988, pp. 627–630.

[30] A. F. VAN DER STAPPEN, *Distributed data structures for sparse linear algebra*, Master's thesis, Eindhoven Univ. of Technology, the Netherlands, June 1988.

[31] E. F. VAN DE VELDE, *Experiments with multicomputer* LU*-decomposition*, Concurrency: Practice and Experience, 2 (1990), pp. 1–26.

[32] J. G. G. VAN DE VORST, *The formal development of a parallel program performing* LU*-decomposition*, Acta Inform., 26 (1988), pp. 1–17.

[33] Z. ZLATEV, *On some pivotal strategies in Gaussian elimination by sparse technique*, SIAM J. Numer. Anal., 17 (1980), pp. 18–30.

[34] ———, *Computational Methods for General Sparse Matrices*, Mathematics and Its Applications 65, Kluwer Academic Publishers, Dordrecht, the Netherlands, 1991.

[35] Z. ZLATEV, J. WASNIEWSKI, AND K. SCHAUMBURG, Y12M—*Solution of Large and Sparse Systems of Linear Algebraic Equations*, Lecture Notes in Computer Science 121, Springer-Verlag, Berlin, 1981.