SPARSE TRIANGULAR SYSTEM PARTITIONING

ARNO C. N. VAN DUIN

Abstract. The parallel solution of a sparse triangular system can be a serious bottleneck in parallel computation. An improvement on the parallel efficiency can be achieved by partially inverting the triangular system. To this end, the triangular matrix is represented by the product of a (small) number of sparse factors. These sparse factors are constructed in such a way that their inverses have the same sparsity pattern. A new method for finding these factors is presented. This method factors the triangular system with a smaller number of factors than optimal column methods do.

Key words. parallel computation, triangular system, sparse matrix, matrix inverse

1. Introduction.

Finding the solution vector $\mathbf{x}$ of a triangular system

$$L\mathbf{x} = \mathbf{b}$$

where L is a triangular $N \times N$ matrix and $\mathbf{b}$ an N-dimensional vector, is a problem that arises in many application areas. For instance, when a system of equations is solved using Gaussian elimination or an iterative method with an incomplete factorization type of preconditioner. In many applications, such as finite element applications, solution of initial value problems by implicit methods, and Newton-like methods for the solution of non-linear equations, the system needs to be solved for multiple right-hand sides. In order to be able to solve complete systems efficiently on a parallel computer, not only the factorization needs to be done in parallel but also the triangular solves.

Instead of the problem of solving $\mathbf{x}$ from $L\mathbf{x} = \mathbf{b}$ we consider the equivalent problem of calculating $\mathbf{x}$ from $\mathbf{x} = L^{-1}\mathbf{b}$ because the matrix-vector multiplication in the latter has more potential for parallelism.

Without loss of generality only lower triangular matrices with a unit diagonal are considered. Each of these systems L can be written as:

$$L = I + \sum_{j=1}^{N-1} \sum_{i=j+1}^{N} l_{ij}\mathbf{e}_i\mathbf{e}_j^T \tag{1}$$

where l_{ij} denotes the entry of matrix L at position (i, j).

L can also be expressed in product form[1]. Since $\mathbf{e}_i^T\mathbf{e}_j = \delta_{ij}$ (Kronecker delta), L reads:

$$L = \prod_{j=1}^{N-1} \prod_{i=j+1}^{N} \left(I + l_{ij}\mathbf{e}_i\mathbf{e}_j^T\right) \tag{2}$$

[1]Because of the non-commutativity of matrix-multiplication, we assign a specific order of operands to the product symbol $\prod$:

$$\prod_{i=1}^{3} A_i = A_1A_2A_3 \text{ and } \prod_{i=-=3}^{1} A_i = A_3A_2A_1$$

The inverses of the terms in equation (2) are easy to construct: $\left(I + l_{ij}\mathbf{e}_i\mathbf{e}_j^T\right)^{-1} = \left(I - l_{ij}\mathbf{e}_i\mathbf{e}_j^T\right)$, since $(i > j)$. And thus:

$$L^{-1} = \prod_{j=-N-1}^{1} \prod_{i=-N}^{j+1} \left(I - l_{ij}\mathbf{e}_i\mathbf{e}_j^T\right) \tag{3}$$

Define $V_{i,j} = \left(I - l_{ij}\mathbf{e}_i\mathbf{e}_j^T\right)$. L^{-1} can now be written as:

$$L^{-1} = \prod_{j=-N-1}^{1} \prod_{i=-N}^{j+1} V_{i,j} \tag{4}$$

Although matrix-matrix products are not commutative in general, a lot of the $V_{i,j}$'s are.

When a group of V-matrices is taken together a new product form of L^{-1} is obtained. Let the number of factors in this product be M and let W_m be one of these factors, then (4) is rewritten to:

$$L^{-1} = \prod_{i=-M}^{1} W_m \tag{5}$$

Alvarado et al. [1, 2, 3] discuss the situation where all V matrices with nonzero off-diagonal entries in one column are part of the same W matrix. In this paper we extend their notion of partitions. We introduce a new type of partition, the Γ-partition in section 3. Some definitions used to describe this partition are presented in section 2.

2. Definitions. We associate with a lower triangular matrix L a *directed acyclic graph* $G(L)$ with vertices $V = \{1, 2, \ldots, n\}$ and edges $E(L) = \{(i,j) \| L_{ji} \neq 0\}$. If $(i,j) \in E$ then i is called a predecessor of j and j is a successor of i. An ordering of the vertices is called an *ascending topological order* if all nodes are numbered before their successors. The *transitive closure* of a graph $G = (V, E)$ is the graph $G' = (V, E')$ where $E' = \{(i,j) \|$ there is a path from i to j in $G\}$. The graph $G(L^{-1})$ is equal to the transitive closure of $G(L)$ [6].

DEFINITION 1 The *induced subgraph* $(\mathrm{isg}(\tilde{V}))$ of a set of nodes $\tilde{V} \subset V$ from G is a graph $\tilde{G} = (\tilde{V}, \tilde{E})$, where $\tilde{E} = \{(i,j) \| (i,j) \in E \wedge i \in \tilde{V} \wedge j \in \tilde{V}\}$.

DEFINITION 2 A partition in factors like Equation 5 is called an *edge partition* of G.

DEFINITION 3 An edge partition where all edges leaving from the same node are in the same factor is called a *column partition.*

DEFINITION 4 An edge partition where all edges ending in the same node are in the same factor is called a *row partition.*

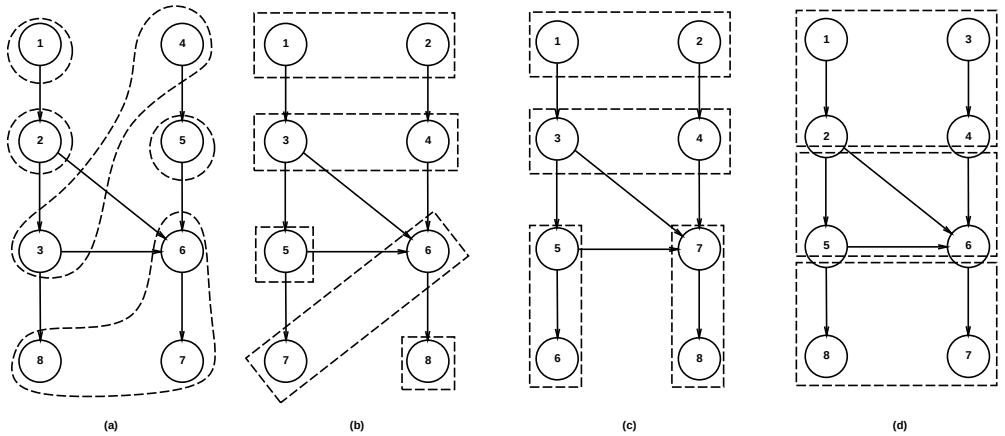

Figure 1: Best no-fill partitions for a graph: (a) column partition without reordering, five factors are required, (b) level scheduling, five levels, (c) column partition with reordering, four factors are required, (d) Γ-partition with reordering, three factors are required.

DEFINITION 5 A matrix A is *invertible in place* if its inverse has the same sparsity pattern as A.

DEFINITION 6 If all factors of an edge partition are invertible in place, it is called a *no-fill edge partition.*

DEFINITION 7 A no-fill edge partition with the smallest possible number of factors is called a *best no-fill edge partition.*

DEFINITION 8 An edge partition is called *non-overlapping* if for all $i, j \in V$ there is only one factor with a path from i to j.

DEFINITION 9 A node is said to belong to a factor if that factor is the last factor with an edge ending in this node, or (if there is no such factor) the first factor with an edge leaving from this node.

DEFINITION 10 The *diagonal block* of a factor is that part of the matrix that is represented by the induced subgraph of the nodes that belong to this factor.

DEFINITION 11 An *off-diagonal edge* is an edge that is not part of the induced subgraph of any of the diagonal blocks.

DEFINITION 12 In an *off-diagonal block* there is no edge starting in a node if there is also an edge ending in that node.

DEFINITION 13 A *zero node* of a factor in an edge partition is a root in the induced subgraph of the nodes of this factor.

In figure 1 some partitions for a small graph are shown.

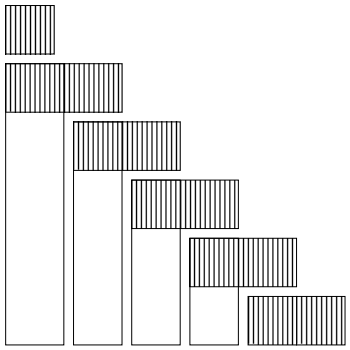

Figure 2: The Γ-partition of a lower triangular matrix

3. Γ-partition. In this paper we are concerned with non-overlapping no-fill partitions. The column partitions used by Alvarado et al. [3] give rise to a block decomposition. In this block decomposition each diagonal block combined with the corresponding off-diagonal block must be invertible in place. Instead of taking the diagonal blocks and the off-diagonal blocks together, a different partition strategy is to leave them apart. In that case there are no constraints on the off-diagonal blocks, but the diagonal blocks must still be invertible in place. Note that the number of factors in this partition is not necessarily twice as much as the column partition: the no-fill constraint only applies to the diagonal blocks. These blocks will be larger (in general) than with column partitions.

The partition we propose is to combine each diagonal block not with the corresponding off-diagonal block but with the off-diagonal block that corresponds to the previous diagonal block.

DEFINITION 14 A Γ-*partition* is a non-overlapping no-fill edge partition, where nodes are assigned to a factor together with the edges of the induced subgraph as well as all off-diagonal edges from the nodes of the previous factor.

The factors in this Γ-partition consist of a diagonal block and the off-diagonal block that is below the previous diagonal block. Hence, it has the form of a Γ. This is graphically shown in figure 2. The no-fill constraint applies to the striped areas in the figure. The edges in this area are also the edges that possibly change in value when the factor is inverted.

A Γ-partition can be specified by the nodes in the diagonal blocks. We use the following notation to specify a particular Γ-partition: $P = \{\{v_{1_1}, v_{1_2}, \dots\}, \dots, \{v_{k_1}, v_{k_2}, \dots\}\}$. When the nodes are not in their original order it is a *reordered* Γ-*partition*, e.g. {{1,2,4,5}, {3,6}, {7,8}} is the reordered Γ-partition of the graph in Figure 1. Note that the same notation can be used to specify row-partitions, column-partitions, and level scheduling.

In section 4 an algorithm that finds a best no-fill Γ-partition is presented, and an algorithm for finding a best no-fill reordered Γ-partition is given in section 5.

4. Best no-fill Γ-partition. Algorithm PO1 is an adaptation of the row-wise version of algorithm P1 proposed by Alvarado et al. [3]. It is a greedy algorithm, that tries to add the next row to the current block if this will leave it invertible in place, i.e. it leaves the corresponding graph transitively closed. Since the current block (without the next

row) is transitively closed only paths to the node associated with the next row need to be checked. This is equal to checking the following condition:

CONDITION 1 In the current block there is no predecessor of the next node that has a predecessor that (a) is in the current or the previous block and (b) is not a predecessor of the next node.

or in matrix terms:

CONDITION 2 For all rows in the current block for which the row under consideration has a nonzero in the corresponding column, the set of column indices of the nonzeros that are on these rows and have a column index that corresponds to a row of either this block or the previous block must be a subset of the set of column indices of the nonzeros of the row under consideration.

Since all rows are considered before their successors are, none of the successors of this row can already be in the current block, so only fill in this row needs to be checked.

ALGORITHM PO1

$i \leftarrow 1; k \leftarrow 1$
while $i \leq n$ **do**
  $r \leftarrow i$
  while all predecessors $\in S_{k-1} \cup S_k$ of every predecessor $\in S_k$ of r are predecessors of r **do**
    $r \leftarrow r + 1$
  od
  $S_k \leftarrow \{i, \ldots, r\}$
  $k \leftarrow k + 1; i \leftarrow r + 1$
od

The difference with the condition in algorithm P1 is the part (a) of 1. This extra check does not add to the complexity of the algorithm, making algorithms P1 and PO1 of equal complexity. A small optimization is possible in checking on condition 2 by using the fact that if v_i and v_j are predecessors of the current node and v_i is a predecessor of v_j and v_j does not cause fill, then v_i will not either, otherwise v_i would have caused fill to v_j as well.

The following theorem is needed to prove the optimality of the reordering produced by PO1.

THEOREM 1 In the Γ-partition generated by PO1, all nodes are in the earliest possible block.

Proof This clearly is true for the first block. Suppose this is true for all blocks up to k. For the first node in block k there is a path P through block $k-1$ to a node j of block $k-2$ such that $\mathrm{isg}(P)$ is not transitively closed. Since node j and the other nodes on the

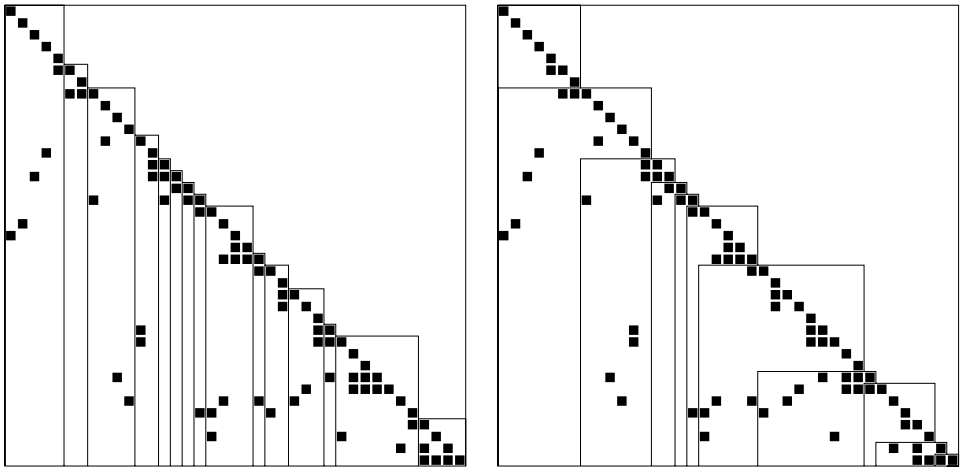

Figure 3: A best no-fill column partition generated by P1 (15 factors) and a best no-fill Γ-partition generated by algorithm PO1 (11 factors) of the incomplete Cholesky factor with one level of fill of minimum degree reordered BCSPWR01

path are in the earliest possible block the first node of block k must be in a block $> k-1$. So, the first node in block k is in the earliest possible block. Since all other nodes of block k must be in a block not earlier than where the first node is in (no reordering), they too are in the earliest possible block. Therefore all nodes of block k are in the earliest possible block. □

This leads to the following theorem:

THEOREM 2 Algorithm PO1 finds a best no-fill Γ-partition.

Proof Suppose PO1 finds a Γ-partition with k blocks. Since each node in the k^{th} block must be in block $\geq k$ (Theorem 1), k is the minimal number of blocks. □

In figure 3 the result of algorithm P1 from [2] and that of algorithm PO1 on the incomplete Cholesky factor with one level of fill of the minimum degree reordered matrix BCSPWR01 from the Harwell-Boeing collection [4] are presented. The number of levels (for level scheduling) for this matrix is 11, just like the number of factors in the PO1. But level scheduling does reorder the matrix, so we proceed with trying to find a best *reordered* Γ-partition.

5. Best reordered Γ-partition. In this section an algorithm similar to Alvarado and Schreiber's algorithm RP2 is presented. An ascending topological ordering of an acyclic digraph G is constructed, such that the reordered graph has a best no-fill Γ-partition with the smallest possible number of factors. This also is a greedy algorithm, but in stead of considering only the next row (like in algorithm PO1), all rows with no unnumbered predecessors (denoted by the set E) can be added to the current block if condition 1 is fulfilled.

ALGORITHM RPO2

```
forall v ∈ V do
    count(v) ← indegree(v)
od
E ← {v ∈ V | count(v) = 0}
i ← 0; k ← 1
while i < n do
    S_k ← ∅
    H ← ∅
    while E ≠ ∅ do
        take next v from E; E ← E\{v}
        if all predecessors ∈ S_{k-1} ∪ S_k of every predecessor ∈ S_k of v
                are predecessors of v then
            i ← i + 1; α(v) ← i;
            S_k ← S_k ∪ {v}
            for every successor w of v do
                count(w) ← count(w) − 1
                if count(w) = 0 then E ← E ∪ {w}; fi
            od
        else
            H ← H ∪ {v}
        fi
    od
    k ← k + 1
    E ← H
od
```

The proof of the optimality of algorithm RPO2 goes along the same lines as for algorithm PO1.

THEOREM 3 In the reordered Γ-partition generated by RPO2, all nodes are in the earliest possible block.

Proof This clearly is true for the first block. Suppose this is true for all blocks up to k. For each zero node in block k there is a path P through block $k-1$ to a zero node j of block $k-2$ such that $\mathrm{isg}(P)$ is not transitively closed (otherwise condition 1 would have been fulfilled). Since node j and the other nodes on the path are in the earliest possible block each zero node of block k must be in a block $> k-1$. So, each zero node in block k is in the earliest possible block. Since all other nodes are successors of (at least one of) these nodes, they too must be in a block $\geq k$ (ascending topological ordering). Therefore all nodes of block k are in the earliest possible block. □

THEOREM 4 Algorithm RPO2 finds a best no-fill reordered Γ-partition.

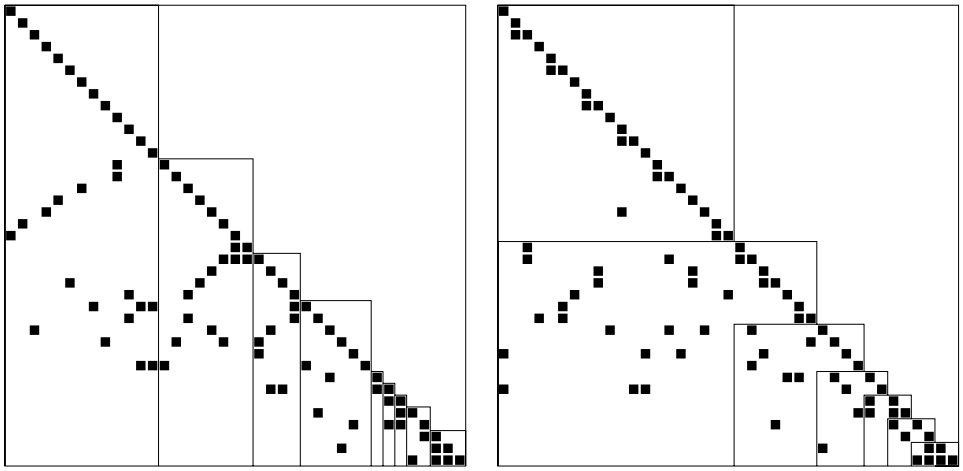

Figure 4: A best no-fill reordered row partition generated by RP2 (9 factors) and a best no-fill reordered Γ-partition generated by algorithm RPO2 (7 factors) of the incomplete Cholesky factor with one level of fill of minimum degree reordered BCSPWR01

Proof Suppose RPO2 finds a Γ-partition with k blocks. Since each node in the k^{th} block must be in block $\geq k$ (Theorem 3), k is the minimal number of blocks. □

In figure 4 the result of algorithm RP2 from [2] and that of algorithm RPO2 on the incomplete Cholesky factor with one level of fill of the minimum degree reordered matrix BCSPWR01 from the Harwell-Boeing collection [4] are presented.

LEMMA 1 Level scheduling is a no-fill reordered Γ-partition.

Proof For all nodes in each diagonal block, condition 1 is fulfilled. □

For the RP2 algorithm a row wise version (RP2R) can be formulated by changing

if all predecessors $\in S_{k-1} \cup S_k$ of every predecessor $\in S_k$ of v
are predecessors of v **then**

to

if all predecessors of every predecessor $\in S_k$ of v
are predecessors of v **then**

This difference between RPO2 and RP2R in the condition for adding a row to a block causes that in RP2R the complete row can cause fill, but in RPO2 only part of the row needs to be considered. This observation leads to the following theorem:

THEOREM 5 Every no-fill row-partition is also a Γ-partition.

Proof For all nodes in each diagonal block, condition 1 is fulfilled. □

As a consequence:

THEOREM 6 RPO2 generates a partition with a less or equal number of factors than RP2R.

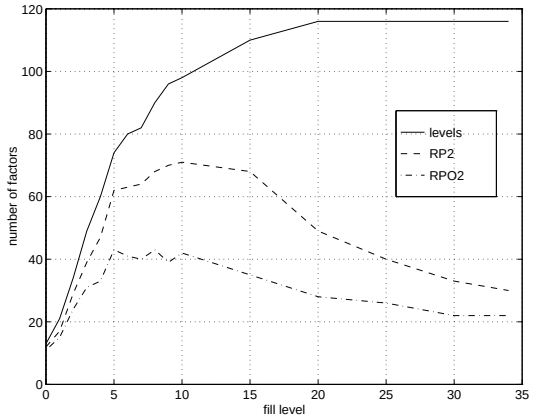

Figure 5: Effect of fill on the number of factors for matrix BCSPWR10

Proof Since RPO2 generates an optimal Γ-partition, the number of factors in the Γ-partition generated by RP2R must be equal or larger. □

Likewise an algorithm RPO2C, a column version of RPO2, can be constructed. The number of factors in the partition generated by RPO2C is always less or equal to the number of factors in the partition generated by RP2.

Note that there is no direct relation between the number of factors in the partition generated by RPO2 and RP2, but as will be shown in the next section RPO2 usually generates better partitionings then RP2.

6. Experimental Results. In this section the performance of the algorithms presented in the previous section is tested on a set of matrices from the Harwell-Boeing collection [4].

In Figure 5 for matrix BCSPWR10 the influence of the amount of fill that is allowed in the factorization, on the number of factors generated by RP2 and RPO2, and the number of levels as used by level scheduling is shown. Of course, the number of levels is a strictly non-decreasing function of the fill-level. This is not necessarily so for the number of factors in the partitioning generated by RP2 and RPO2. More fill can mean that condition 1 for RPO2 or the corresponding condition for RP2 is fulfilled more often, making the factors larger and the number of factors less, which is the case for this particular matrix. From Figure 5 and Table 1 we see that the turning point for the number of factors in the Γ-partitioning lies beyond fill level 4. So large gains like the factor 20 obtained in [2] are not to be expected for triangular matrices arising from incomplete factorizations.

In Table 1 the results of the partitioning algorithms for some matrices from the Harwell-Boeing set are given. Some of the matrices are reordered with H0 [5] prior to the incomplete factorization in order to get a zero free diagonal. The column headed 'fill' gives the number of fill levels that are allowed in the factorization, 'nnz' denotes the number of nonzeros in the L and U, 'lvlsL' and 'lvlsU' denote the number of levels in L and U, the columns headed by 'P1', 'PO1', 'RP2', and 'RPO2' give the number of factors in the partitioning using the specified algorithms. For all the matrices RPO2 gives the partitioning with the least number of factors. The gain over level-scheduling ranges from no gain for some zero

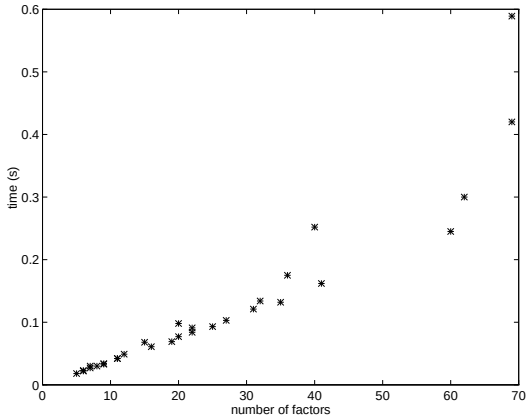

Figure 6: The time needed to solve partitioned triangular systems on a CM-5 as a function of the number of factors.

fill incomplete factors to a factor six for matrix GAFF1104 with four levels of fill.

The solution process has been tested on a CM-5, using the built-in sparse matrix-vector multiplication provided by the CMSSL library package. Triangular matrices were generated randomly and RPO2 was used to partition the matrices. The size varied from 100 to 3200 with 199 to 54532 nonzeros. The timings for the different matrices as a function of the number of factors in the Γ-partitioning are presented in Figure 6 . These timings show that the time required to solve a triangular system using a Γ-partitioning is roughly proportional to the number of factors. Thus the number of factors seems to be a good measure for the quality of the partitioning.

7. Conclusions. A new type of partitioning for lower-triangular matrices has been introduced and an algorithm to generate such a partitioning has been presented. The optimality of this algorithm is proven.

A number of experiments are presented. These experiments show that the gain over level scheduling with respect to the number of factors is best for higher levels of fill. Experiments on the Connection Machines show the solution time to be proportional to the number of factors. Using the new partitioning method results in a smaller (or worst case equal) number of factors than the standard column partitionings, so that this method is highly valuable for applications such as iterative solvers with triangular preconditioners, structural analysis, or power systems applications.

8. Acknowledgments. The author would like to thank Harry Wijshoff for his constructive criticism.

REFERENCES

[1] F.L. Alvarado, A. Pothen, and R. Schreiber. Highly parallel sparse triangular solution. In A. George, J.R. Gilbert, and J.W.H. Liu, editors, *Graph Theory and Sparse Matrix Computation, The IMA Volumes in Mathematics and its Applications 56*, pages 141–157. Springer-Verlag, 1993.

[2] F.L. Alvarado and R. Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM J. Sci. Comput.*, 14(2):446–460, March 1993.

matrix	fill	nnz	lvlsL	P1	PO1	RP2	RPO2	lvlsU	P1	PO1	RP2	RPO2
bp800†	0	4534	28	89	64	22	21	11	75	31	10	9
	1	5531	39	104	70	30	27	15	84	35	14	11
	2	6177	43	109	77	35	29	27	90	37	25	19
	3	6815	45	116	79	36	29	36	99	44	33	24
	4	7271	48	119	82	39	29	40	110	45	37	21
gaff1104†	0	16056	132	365	270	103	76	136	340	266	102	80
	1	28982	253	531	198	180	83	262	519	201	177	81
	2	39987	319	499	148	198	70	323	480	152	189	75
	3	49580	371	497	142	217	73	382	480	148	208	73
	4	58127	404	490	125	227	66	408	477	128	220	67
gre1107	0	5664	41	40	40	40	40	21	118	17	19	11
	1	9692	41	40	40	40	40	29	118	23	27	19
	2	16848	41	40	40	40	40	57	123	36	53	36
	3	27985	71	470	72	70	53	94	258	127	90	86
	4	41082	426	624	176	351	160	117	383	351	115	100
orsirr1	0	6858	39	242	158	38	38	39	242	158	38	38
	1	12202	259	716	504	253	208	259	716	504	253	208
	2	20024	383	719	490	358	245	383	719	490	358	245
	3	30158	456	739	350	425	212	456	739	350	425	212
	4	41608	602	775	322	547	245	602	775	322	547	245
orsirr2	0	5970	41	265	185	40	38	41	265	185	40	38
	1	10592	235	600	413	227	184	235	600	413	227	184
	2	17432	360	642	413	335	230	360	642	413	335	230
	3	26758	429	652	294	392	192	429	652	294	392	192
	4	36996	544	675	261	481	203	544	675	261	481	203
orsreg1	0	14133	45	44	44	44	44	45	44	44	44	44
	1	24853	105	1988	1819	103	103	105	1988	1819	103	103
	2	40093	189	1967	1819	184	160	189	1967	1819	184	160
	3	56971	267	1975	1160	260	206	267	1975	1160	260	206
	4	75519	385	1957	1160	375	286	385	1957	1160	375	286
pores2	0	9613	37	127	80	26	25	28	95	78	25	25
	1	13371	116	462	326	92	72	97	433	344	89	69
	2	17939	192	477	337	154	114	157	450	348	145	104
	3	23527	295	500	253	230	151	252	482	284	227	130
	4	29178	351	506	245	277	161	305	491	275	275	143
pores3	0	3474	78	114	114	43	43	77	176	111	42	42
	1	4616	151	225	222	82	79	148	258	188	146	94
	2	5473	188	227	222	119	97	221	291	189	183	115
	3	6509	190	227	151	119	80	286	291	216	217	176
	4	7187	228	229	152	157	81	292	291	183	220	145
saylr4	0	22316	55	54	54	54	54	55	54	54	54	54
	1	38814	257	2324	1589	239	193	257	2324	1589	239	193
	2	62020	382	2298	1589	368	284	382	2298	1589	368	284
	3	88938	719	2771	1266	684	393	719	2771	1266	684	393
	4	123226	865	2747	1266	838	462	865	2747	1266	838	462
sherman1	0	3750	28	27	27	27	27	28	27	27	27	27
	1	5356	74	359	286	70	58	74	359	286	70	58
	2	6912	106	345	286	99	83	106	345	286	99	83
	3	8832	164	390	210	154	107	164	390	210	154	107
	4	11300	205	394	212	192	123	205	394	212	192	123
sherman2	0	23094	74	336	38	47	26	69	308	64	45	26
	1	41615	208	360	146	119	59	210	423	158	115	65
	2	67448	328	337	133	168	76	328	389	142	145	82
	3	90517	386	351	113	191	73	388	400	115	163	76
	4	117031	516	334	109	231	85	518	350	108	189	86

Table 1: results for various matrices from the Harwell-Boeing set (the matrices marked with † are pre-reordered with H0 in order to get a zero-free diagonal)

[3] F.L. Alvarado, D.C. Yu, and R. Betancourt. Partitioned sparse A^{-1} methods. *IEEE Trans. Power Systems*, 5(2):452–459, May 1990.

[4] I.S. Duff, R.G. Grimes, and J.C. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

[5] K.A. Gallivan, B.A. Marsolf, and H.A.G. Wijshoff. Solving large nonsymmetric sparse linear systems using MCSPARSE. *Parallel Comput.*, 22(10):1291–1333, December 1996.

[6] J.R. Gilbert. Predicting structure in sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, 15(1):62–79, Jan. 1994.

A. Source Code. This appendix contains a FORTRAN implementation of the algorithms.

A.1. Algorithm PO1.

```
      subroutine po1 ( n, nnz, ka, phgh, itemp, ibloc )
c
c ======================================================================
c
c     Programmer     Arno van Duin
c     Version   1.0  Date 09-04-1997
c
c **********************************************************************
c
c     KEYWORDS
c
c     sparse
c     triangular matrix
c     decomposition
c
c **********************************************************************
c
c     INPUT / OUTPUT PARAMETERS
c
      implicit none
      integer n, nnz
      integer ka(nnz+1), phgh(-1:n-2), itemp(n), ibloc(n)
c
c     ibloc    o contains for each row to which factor it belongs
c     itemp    - work array used for scattering rows
c     n        i the dimension of the matrix
c     nnz      i the number of nonzeros in the matrix
c     ka       i the MSR specification of the sparsity pattern
c     phgh     o contains for each factor a pointer to its first row
c
c **********************************************************************
c
c     LOCAL PARAMETERS
c
      integer i, k, r
      logical check
c
c     check   if true, current row does not cause fill
c     i       loop counter
c     k       number of factors
c     r       current row
c
c **********************************************************************
c
c     CALLED SUBROUTINES
c
      logical nofill
c
c     NOFILL  Function that determines whether the specified row will
c             cause fill if added to the current factor
c
c ======================================================================
c
```

```
c     --- initializations
c
      do i = 1, n
         itemp(i) = 0
         ibloc(i) = 0
      enddo

      phgh(-1) = 0
      phgh(0)  = 0
      ibloc(1) = 1
      i        = 2
      k        = 1
c
c     --- while not all rows added
c
      do while ( i .le. n )
         ibloc(i) = k
         r = i+1
         check = .true.
c
c        --- while this row can be added
c
         do while ( r .le. n .and. check )
c
c           --- take next row and check if its addition will not
c               cause any fill
c
            check = nofill ( n, nnz, r, ka, itemp, ibloc, k )
            ibloc(r) = k
            r = r + 1
         enddo
         if ( .not. check ) r = r-1
         phgh(k) = r-1
         k = k + 1
         i = r
      enddo
      end

      function nofill ( n, nnz, r, ka, itemp, ibloc, this )
      logical nofill
c
c ======================================================================
c
c     Programmer     Arno van Duin
c     Version   1.0  Date 09-04-1997
c
c **********************************************************************
c
c     KEYWORDS
c
c     sparse
c     triangular matrix
c     reordering
c     decomposition
c
c **********************************************************************
c
```

```
c     INPUT / OUTPUT PARAMETERS
c
      implicit none
      integer n, nnz, r, this
      integer ka(nnz+1), itemp(n), ibloc(n)
c
c     ibloc    i contains for each row to which factor it belongs
c     itemp    - work array used for scattering rows
c     n        i the dimension of the matrix
c     nnz      i the number of nonzeros in the matrix
c     ka       i the MSR specification of the sparsity pattern
c     r        i current row
c     this     i current factor
c
c ********************************************************************
c
c     LOCAL PARAMETERS
c
      integer i, j, k, l
c
c     i       loop counter
c     j       loop counter
c     k       loop counter
c     l       loop counter
c
c ********************************************************************
c
c     CALLED SUBROUTINES
c
c     None.
c
c ====================================================================
c
c     --- scatter current row in workspace
c
      itemp(r) = 1
      do j = ka(r), ka(r+1)-1
         itemp(ka(j)) = 1
      enddo
      nflop = nflop + ka(r+1)-ka(r)
c
c     --- check if any of the other rows in this factor would cause fill
c         only if there is an edge from this row to the other row it
c         needs to be considered
c
      nofill = .true.
      do i = ka(r), ka(r+1)-1
         j = ka(i)
         if ( ibloc(j) .ge. this .and. itemp(j) .eq. 1 ) then
c
c           --- check that row
c
            do k = ka(j), ka(j+1)-1
               nflop = nflop + 1
               l = ka(k)
               if ( ibloc(l) .ge. this-1 ) then
                  if ( itemp(l) .eq. 0 ) then
```

```
c
c                    --- if fill then goto exit
c
                     nofill = .false.
                     go to 10
                  else
c
c                    --- small optimization entries need to be considered
c                        only once (otherwise a previous row would not
c                        have been added)
c
                     itemp(l) = 2
                  endif
               endif
            enddo
         endif
      enddo
10    continue
c
c     --- reset workspace
c
      itemp(r) = 0
      do j = ka(r), ka(r+1)-1
         itemp(ka(j)) = 0
      enddo
      end
```

A.2. Algorithm RPO2.

```
      subroutine rpo2 ( n, nnz, ka, phgh, itemp, bloc, kat, E, count,
     +                  blokaant )
c
c ======================================================================
c
c     Programmer     Arno van Duin
c     Version   1.0  Date 15-04-1997
c
c **********************************************************************
c
c     KEYWORDS
c
c     sparse
c     triangular matrix
c     reordering
c     decomposition
c
c **********************************************************************
c
c     INPUT / OUTPUT PARAMETERS
c
      implicit none
      integer blokaant, n, nnz
      integer ka(nnz+1), phgh(n), itemp(n), kat(nnz+1),
     +        E(n), count(n), bloc(n)
c
c     bloc     o contains for each row to which factor it belongs
c     blokaant o number of factors
c     count    - workarray to keep track which rows are eligible
```

```
c     E        o contains the original row numbers E(3)=4 means that
c                the original row 4 is now the third row
c     itemp    - work array used for scattering rows
c     n        i the dimension of the matrix
c     nnz      i the number of nonzeros in the matrix
c     ka       i the MSR specification of the sparsity pattern
c     kat      i the MSR specification of the transposed sparsity pattern
c     phgh     o contains for each factor a pointer to its first row
c
c ***********************************************************************
c
c     LOCAL PARAMETERS
c
      integer i, j, k, l, rootaant, newaant, P, thisroot, thisedge
      integer newroots(n), roots(n)
      logical delay
c
c     delay    if true, current row causes fill
c     i        loop counter
c     j        loop counter
c     k        loop counter
c     l        loop counter
c     newaant  number of rows that have become eligible
c     newroots the numbers of the rows that have become eligible
c     P        current row count
c     rootaant number of rows that can be added to the next factor
c     roots    the numbers of the rows that can be added to the next factor
c     thisedge current edge of current row
c     thisroot current row
c
c ***********************************************************************
c
c     CALLED SUBROUTINES
c
c     None.
c
c =======================================================================
c
c     --- initializations
c
      blokaant = 0
      rootaant = 0
      newaant = 0
      P = 0
      phgh(1) = 1

      do j = 1, n
         itemp(j) = 0
         bloc(j) = 0
         count(j) = ka(j+1)-ka(j)
         if ( count(j) .eq. 0 ) then
            rootaant = rootaant + 1
            roots(rootaant) = j
         endif
      enddo
c
c     --- determine reordering
```

```
c
      do while ( rootaant .gt. 0 )

         blokaant = blokaant + 1
c
c        --- add all roots to this factor
c
         do i = 1, rootaant
            thisroot = roots(i)
            bloc(thisroot) = blokaant
            P = P + 1
            E(P) = thisroot
c
c           --- add all edges to this factor
c
            do j = kat(thisroot), kat(thisroot+1)-1
               thisedge = kat(j)
               count(thisedge) = count(thisedge) - 1
               if ( count(thisedge) .eq. 0 ) then
c
c                 --- register new roots
c
                  newaant = newaant+1
                  newroots(newaant) = thisedge
               endif
            enddo
         enddo

         rootaant = 0
c
c        --- do for all new roots
c
         do while ( newaant .gt. 0 )
            thisroot = newroots(newaant)
            newaant = newaant - 1
            delay = .false.
c
c           --- check fill
c
            do j = ka(thisroot), ka(thisroot+1)-1
               itemp(ka(j)) = 1
            enddo
            do i = ka(thisroot), ka(thisroot+1)-1
               j = ka(i)
               if ( bloc(j) .ge. blokaant .and. itemp(j) .eq. 1 ) then
                  do k = ka(j), ka(j+1)-1
                     l = ka(k)
                     if ( bloc(l) .ge. blokaant-1 ) then
                        if ( itemp(l) .eq. 0 ) then
                           delay = .true.
                           go to 10
                        else
                           itemp(l) = 2
                        endif
                     endif
                  enddo
               endif
```

```
            enddo
10          continue
            do j = ka(thisroot), ka(thisroot+1)-1
               itemp(ka(j)) = 0
            enddo
c
c           --- add okay edges to factor and update counts
c
            if ( delay ) then
               rootaant = rootaant + 1
               roots(rootaant) = thisroot
            else
               P = P + 1
               E(P) = thisroot
               bloc(thisroot) = blokaant
               do j = kat(thisroot), kat(thisroot+1)-1
                  thisedge = kat(j)
                  count(thisedge) = count(thisedge) - 1
                  if ( count(thisedge) .eq. 0 ) then
                     newaant = newaant + 1
                     newroots(newaant) = thisedge
                  endif
               enddo
            endif

         enddo
         phgh(blokaant+1) = P+1

      enddo

      end
```