OPTIMAL DAG PARTITIONING FOR PARTIALLY INVERTING TRIANGULAR SYSTEMS

ARNO C. N. VAN DUIN*

Abstract. An approach for solving sparse triangular systems of equations on highly parallel computers employs a partitioned representation of the inverse of the triangular matrix so that the solution can be obtained by a series of matrix-vector multiplications. This approach requires a number of global communication steps that is proportional to the number of factors in the partitioning. The problem of finding the minimal number of factors subject to the requirement that these factors do not need more storage space than the original triangular factor has been studied by several authors. We formulate a new related graph problem and give an algorithm to solve this problem. We prove that the partitioning resulting from this algorithm requires less factors than existing partitioning algorithms.

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

1. Introduction. In this paper a graph partitioning algorithm is considered which arises in the solution of sparse triangular systems of equations on highly parallel computers using the partitioned inverse approach. The advantage of this approach over the conventional substitution algorithm is that there is much more parallelism to be exploited in the arising matrix-vector multiplications. The number of these matrix-vector multiplications must be kept to a minimum in order to reduce the amount of global communication steps necessary after each matrix-vector multiplication.

The problem that needs to be solved is:

PROBLEM 1 Given a lower triangular matrix L find matrices S_k such that:

1. $L = \prod_{k=1}^{K} S_k$
2. the sparsity pattern of S_k is equal to the sparsity pattern of S_k^{-1},
3. the sparsity pattern of S_k and $S_i (i \neq k)$ do not overlap outside the diagonal,
4. the sparsity pattern of $\sum_{k=1}^{K} S_k$ is equal to the sparsity pattern of L, and
5. K is minimum for all factorizations that satisfy the first four properties.

The matrices S_k can be stored efficiently in the memory space required for L. In addition, the calculation of the solution vector $\mathbf{x}$ to the system $L\mathbf{x} = \mathbf{b}$ can be done in K steps of parallel matrix-vector multiplications. The number of expensive global communication steps is proportional to the number of factors K in the factorization of L. Therefore, K can be used to predict the time needed for the triangular solution on highly parallel machines. For sake of simplicity we assume L to have a unit diagonal.

In stead of solving problem 1 we formulate a related graph problem, but first we introduce some terminology used in graph theory.

Associate with the matrix L a graph $G = (V, E)$ with vertices $V = \{1, \ldots, n\}$ corresponding to the columns of L and edges $E = \{(j, i) | l_{ij} \neq 0\}$ corresponding to the

* Department of Computer Science, Leiden University, P.O. Box 9512, 2300 RA Leiden, The Netherlands, arno@cs.leidenuniv.nl

nonzeros of L. The edge e_{ij} is directed from the lower numbered vertex i to the higher numbered vertex j. A vertex i is called a *predecessor* of another vertex j in G if there exists a directed path from i to j in G. If i is a predecessor of j then is j a *successor* of i. An *ordering* of G is any bijection from V to the set $\{1, 2, \ldots, n\}$. An ordering is called an *ascending topological ordering* if every node has a lower number than all its successors. The transitive closure of a graph $G = (V, E)$ is a graph $\tilde{G} = (V, \tilde{E})$ with $\tilde{E} = \{(i, j) | \text{there is a path in } G \text{ from } i \text{ to } j\}$. If a graph G is equal to its own transitive closure, G is called transitively closed. The graph associated with L^{-1} is equal to the transitive closure of the graph associated with L [8]. The *induced subgraph* of a subset of vertices $\tilde{V}$ is the graph $\tilde{G} = (V, \tilde{E})$ with $\tilde{E} = \{(i, j) | (i, j) \in E \wedge i \in \tilde{V}\}$.

Others [1, 2, 3, 5, 11] have considered the following node partitioning problem, which is closely related to problem 1:

PROBLEM 2 Given a DAG G, find an ordered partition $R_1 \prec R_2 \prec \cdots \prec R_r$ of its vertices such that

1. for every $v \in V$, if $v \in R_i$ then all predecessors of v belong to $R_1, \ldots, R_i$,
2. the subgraph induced by each R_i is transitively closed, and
3. r is minimum over all partitions that satisfy the first two properties.

A more efficient algorithm exists for the special case where the graph associated with $L + L^T$ is chordal [9], as is the case when L is a Cholesky factor of a symmetric positive definite matrix. Let S_k be the matrix with the v^{th} column equal to that of L for all nodes v in R_k, and let all other columns m be filled with the corresponding unit vector $\mathbf{e}_m$. The first four demands of problem 1 are met, so we have a factorization of L in r terms. The number of factors, r, is an upper bound of K: $K \leq r$.

We try to find a better bound and better partitioning by considering the following edge partitioning problem:

PROBLEM 3 Given a DAG $G = (V, E)$ find an ordered partition $W_1 \prec W_2 \prec \cdots \prec W_t$ of its edges such that

1. for every $e_{ij} \in E$, if $e_{ij} \in W_s$ then all edges $e_{ki} \in E$ belong to $W_1, \ldots, W_s$.
2. the subgraph (V, W_s) is transitively closed.
3. t is minimum over all partitions that satisfy the first two properties.

The next theorem states that the solution to problem 3 results in a partitioning with less factors than the solution to problem 2.

THEOREM 1 $r \geq t$

Proof Let W_i be all edges leaving from the nodes in R_i than condition 1 and 2 of problem 3 are fulfilled, so $t \leq r$. □

Let S_k be the identity matrix with the entry at position (j, i) equal to that of L for all edges e_{ij} in W_k. The first four demands of problem 1 are met, so we have a factorization of L in t terms. The number of factors, t, is an upper bound of K: $K \leq t \leq r$.

It should be noted that problem 1 cannot be fully characterized as a graph problem when the values of the entries are not taken into account. E.g. consider the following triangular matrix and its inverse:

$$\begin{pmatrix} 1 & & & \\ 2 & 1 & & \\ 1 & 2 & 1 & \\ & 3 & 2 & 1 \end{pmatrix} \overset{inv}{\longleftrightarrow} \begin{pmatrix} 1 & & & \\ -2 & 1 & & \\ 3 & -2 & 1 & \\ & 1 & -2 & 1 \end{pmatrix}$$

The solution to problem 1 for this matrix is $K = 1$ although the corresponding graph is not transitively closed [1].

In this paper we introduce an algorithm for solving the edge partitioning problem. The algorithm is discussed in section 2. In section 3 the problem of balancing the factors is briefly discussed. In section 4 several experiments are presented. Some concluding remarks are given in section 5.

2. Optimal Edge-Partition Algorithm. This section describes an algorithm that solves problem 3. It is a greedy algorithm that tries to add to the current factor all edges that start in a node of which all incoming edges are assigned to this or earlier factors. The outgoing edges of such a node are considered for addition to the current factor. When there are no more such nodes we start with a new factor consisting of all edges that were considered for the previous factor but could not be added to that factor. This makes new nodes and new edges available for addition. This process continues until there are no more edges left.

The edge e_{ij} can be added to the current factor W_k if:

CONDITION 1 All edges $e_{\tilde{j}i} \in E$ have been assigned to $W_1, \ldots, W_k$.

CONDITION 2 The graph $(V, W_k \cup \{e_{ij}\})$ is transitively closed.

CONDITION 3 There is no path from i to j in the remaining subgraph.

Note that if an edge could not be added to W_k because either or both of the conditions 2 and 3 were not fulfilled, the edge is (unconditionally) added to W_{k+1}. With only those edges, (V, W_{k+1}) is transitively closed: there are no edges in W_{k+1} to any of the sources of those edges, there are only paths of length one in W_{k+1}, so for all paths from i to j in (V, W_{k+1}) there is an edge e_{ij}, ergo it is transitively closed. Because of condition 2 W_{k+1} remains transitively closed.

[1] the element $(4,1)$ of the inverse is not a *structural* nonzero but a *numerical* nonzero, only structural nonzeros are considered in the relation between the inverse and the transitive closure in [8].

ALGORITHM RPOPT

```
forall v ∈ V do
    count(v) ← indegree(v)
enddo
F ← {e_vw ∈ E | v, w ∈ V, count(v) = 0}
k ← 1
while F ≠ ∅ do
    W_k ← ∅
    H ← ∅
    forall e_vw ∈ F do
        W_k ← W_k ∪ {e_vw}
        count(w) ← count(w) − 1
        if count(w) = 0 then H ← H ∪ {w}; endif
    enddo
    F ← ∅
    while H ≠ ∅ do
        G ← ∅
        take next v from H; H ← H\{v}
        forall e_vw ∈ E do
            if ∀e_uv ∈ W_k ∃e_uw ∈ W_k then G ← G ∪ {e_vw}
            else F ← F ∪ {e_vw} endif
        enddo
        forall e_vw ∈ G do
            if ∀e_vu ∈ F ∄e_uw ∈ E then
                W_k ← W_k ∪ {e_vw}
                count(w) ← count(w) − 1
                if count(w) = 0 then H ← H ∪ {w}; endif
            else F ← F ∪ {e_vw} endif
        enddo
    endwhile
    k ← k + 1
endwhile
```

Condition 3 is necessary because otherwise it might happen that e_{ij} is in W_k and e_{ik} is in W_{k+1}, so e_{kj} cannot be added to W_{k+1} (violation of condition 2). A suboptimal number of factors would be the result. In algorithm RPOPT the checks for this condition are optimized by checking not on any path but just on the existence of an edge $e_{kj} \in E$. If there is no such edge the subgraph induced by the path is not transitively closed, and condition 2 precludes the suboptimal situation.

The condition that prohibited an edge to be added to the previous factor can be used to distinguish three types of edges: (1) $W_k^{(p)}$ is the subset of edges that could not be added to W_{k-1} because of condition 1, i.e. those edges $e_{ij} \in W_k$ for which there is an edge

$e_{\tilde{j}i} \in W_k$, (2) $W_k^{(f)}$ is the subset of edges that could not be added to W_{k-1} because of condition 2, i.e. edges $e_{ij} \in W_k \backslash W_k^{(p)}$ for which $(V, W_{k-1} \cup \{e_{ij}\})$ is not transitively closed, (3) $W_k^{(d)}$ is the subset of edges that could not be added to W_{k-1} because of condition 3, i.e. edges $e_{ij} \in W_k \backslash (W_k^{(f)} \cup W_k^{(p)})$: for each of these edges there is an edge $e_{i\tilde{j}} \in W_k^{(f)}$ and an edge $e_{\tilde{j}j} \in E$.

We have the following relations:

$$W_k = W_k^{(p)} \cup W_k^{(f)} \cup W_k^{(d)} \tag{1}$$

$$W_k^{(p)} \cap W_k^{(f)} = W_k^{(p)} \cap W_k^{(d)} = W_k^{(f)} \cap W_k^{(d)} = \emptyset \tag{2}$$

$$W_k \neq \emptyset \tag{3}$$

Before we can proof that the number $\tilde{K}$ of factors in the partitioning generated by algorithm RPOPT is optimal, i.e. $\tilde{K} = t$, we first proof some lemmas.

LEMMA 1 $W_k^{(f)} \neq \emptyset$ for all k

Proof Suppose $W_k^{(f)} = \emptyset$ for some k. With (1) we have:

$$\left.\begin{array}{c} W_k^{(f)} = \emptyset \Rightarrow W_k^{(d)} = \emptyset \\ W_k^{(f)} = W_k^{(d)} = \emptyset \Rightarrow W_k^{(p)} = \emptyset \end{array}\right\} \Rightarrow W_k = \emptyset$$

This contradicts (3). □

LEMMA 2 For all edges $e_{ij} \in W_k^{(p)}$ there is an edge $e_{\tilde{i}\tilde{j}} \in W_k^{(f)}$ such that there is a path from $\tilde{i}$ to i in (V, W_k).

Proof Since the graph (V, W_k) is acyclic, there is an edge $e_{\tilde{i}\tilde{j}} \in W_k \backslash W_k^{(p)}$ such that there is a path from $\tilde{i}$ to i in (V, W_k). If that edge is in $W_k^{(f)}$ the lemma is true. If it is not in $W_k^{(f)}$ then because of (1) and (2), the edge is in $W_k^{(d)}$. Because of condition 3 there is an edge $e_{\tilde{i}\hat{j}} \in W_k^{(f)}$ such that there is a path from $\tilde{i}$ via $\hat{j}$ to i. Because of condition 1 the edges of this path are all in W_k, and again the lemma is true. □

LEMMA 3 For every edge $e_{ij} \in W_k^{(f)}$ $(k \geq 2)$ there is an edge $e_{\tilde{i}\tilde{j}} \in W_{k-1}^{(f)}$ such that there is a path from $\tilde{i}$ to i in (V, W_{k-1}) and the subgraph induced by the nodes on this path is not transitively closed.

Proof Since $(V, W_{k-1} \cup \{e_{ij}\})$ is not transitively closed there must be at least one edge $e_{\hat{i}\hat{j}} \in W_{k-1}$ such that there is a path from $\hat{i}$ to i in (V, W_{k-1}) but not an edge $e_{\hat{i}i}$. Because of relations (1) and (2) there are three possibilities:

1. If $e_{\hat{i}\hat{j}} \in W_{k-1}^{(f)}$ the path exists.

2. If $e_{\hat{\imath}j} \in W_{k-1}^{(d)}$ then because of condition 3 there is an edge $e_{\hat{\imath}\tilde{\jmath}} \in W_{k-1}^{(f)}$ such that there is a path from $\hat{\imath}$ via $\tilde{\jmath}$ to i. Because of condition 1 and the fact that i does not have any predecessors in (V, W_k) $(e_{ij} \in W_k^{(f)})$, the edges of this path are all in W_{k-1}, and thus the path exists.

3. If $e_{\hat{\imath}\hat{\jmath}} \in W_{k-1}^{(p)}$ then because of lemma 2 the path exists.

There is no edge $e_{\hat{\imath}i}$ so the subgraph induced by the nodes on the path is not transitively closed. □

THEOREM 2 The number of factors $\tilde{K}$ in the edge-partition produced by algorithm RPOPT is optimal.

Proof Due to 3 we have that there exists a path in (V, E) $e_{i_1j_1}, e_{i_2j_2}, \dots, e_{i_mj_m}$ with exactly one edge from each $W_k^{(f)}$. All these edges must be in different (because the subpaths are not transitively closed), consecutive (because of condition 1) factors, so the minimal number of factors t is equal to $\tilde{K}$. □

3. Balancing The Factors. The number of nonzeros in each matrix S_k formed according to the partitioning found by RPOPT can differ greatly among the factors. In order resolve this imbalance some of the edges of a factor can be delayed to the next factor if they leave both factors invertible in place. Identifying all edges that (possibly only in combination with other edges) can be moved to the next factor and finding the optimally balanced partitioning is a hard problem. We show how hard this problem is by considering the sub-problem of finding the optimally balanced partitioning after the edges that can be delayed have been identified. Suppose a simple (fast) scheme is used to find out how many edges can be delayed in each factor and to what factor they can be delayed:

1. determine for the last factor for which nodes it has incoming but no outgoing edges. All edges in previous factors that point to these nodes can be added to this or any of the intermediate factors.

2. determine the same set of nodes for the next to last factor. All nodes in previous factors that point to these nodes or to nodes to which the last factor has incoming nodes can be added to this or any of the intermediate factors.

3. proceed likewise for all factors.

THEOREM 3 The problem of finding optimally balanced factors is NP-complete.

Proof Consider the following NP-complete problem [7, pp 239-240]: Given a task set T with tasks t of length $l(t) = 1$, deadlines $d(t)$, and precedence relations $t \prec \tilde{t}$, determine whether or not there is an m-processor schedule σ for T that obeys the precedence constraints and meets all the deadlines.

Use the following correspondence:

- edge $e_{ij} \leftrightarrow$ task t of length one $l(t) = 1$

- edge set $E \leftrightarrow$ task set T

matrix	n	nnz	matrix	n	nnz
bcspwr01	39	85	sherman4	1104	3786
bcspwr02	49	108	gre_1107	1107	5664
bcspwr03	118	297	pores_2	1224	9613
steam1	240	3762	mahindas	1258	7682
bcspwr04	274	943	bcspwr06	1454	3377
bcspwr05	443	1033	qcgstab1	1600	7840
pores_3	532	3474	bcspwr07	1612	3718
steam2	600	13760	bcspwr09	1723	4117
bp_800	822	4534	orsreg_1	2205	14133
orsirr_2	886	5970	sherman5	3312	20793
sherman1	1000	3750	saylr4	3564	22316
poisson	1024	4992	tfqmr1	3969	19593
orsirr_1	1030	6858	vdvorst3	4096	20224
sherman2	1080	23094	sherman3	5005	20033
gaff1104	1104	16056	bcspwr10	5300	13571

Table 1: The matrices used in the experiments.

- latest factor k to which e_{ij} can be delayed $\leftrightarrow$ deadline $d(t) = k+1$
- the earliest factor for edge $e_{\tilde{i}\tilde{j}}$ is after the latest factor k to which e_{ij} can be delayed $\leftrightarrow$ precedence relation $t \prec \tilde{t}$

Then we have a 1-1 correspondence between the NP-complete problem and answering the question whether or not it is possible to find a partitioning with at most m edges in each factor:

Suppose we have an m-processor schedule. The task with the latest deadline has deadline $\tilde{K}+1$, so all processors must have completed their tasks in $\tilde{K}$ steps. Let the task set T_1 consists of the tasks that are completed in step 1. Similarly define $T_2, \ldots, T_{\tilde{K}}$. Let W_k be the edge subset that contains all edges that correspond to tasks in T_k. Then we have an edge partitioning where each W_k has at most m edges.

Conversely, suppose we have an edge-partitioning $W_1, \ldots, W_{\tilde{K}}$ where each W_k has at most m edges. With each W_k we associate a task subset T_k. Each task $t \in T_k$ must be completed in time step k. Let the tasks of each subset T_k be numbered: $t_k^{(1)}, t_k^{(2)}, \ldots,$ and let σ be the schedule where tasks $t_1^{(i)}, t_2^{(i)}, \ldots$ are assigned to processor i. Since there are at most m tasks in each T_k, σ is an m-processor schedule for T. $\square$

4. Experimental Results. In this section algorithm RPOPT is tested on a set of incomplete factors of matrices from the Harwell-Boeing collection [4]. The matrices used in the experiments are listed in table 1. Matrix `poisson` stems from a standard finite difference discretization of the Poisson-problem on a unit square, matrix `tfqmr1` corresponds to problem 1 in [6], and matrix `vdvorst3` corresponds to problem 3 in [13].

Algorithm RPOPT is used to partition the factors and compared to three other partitioning algorithms:

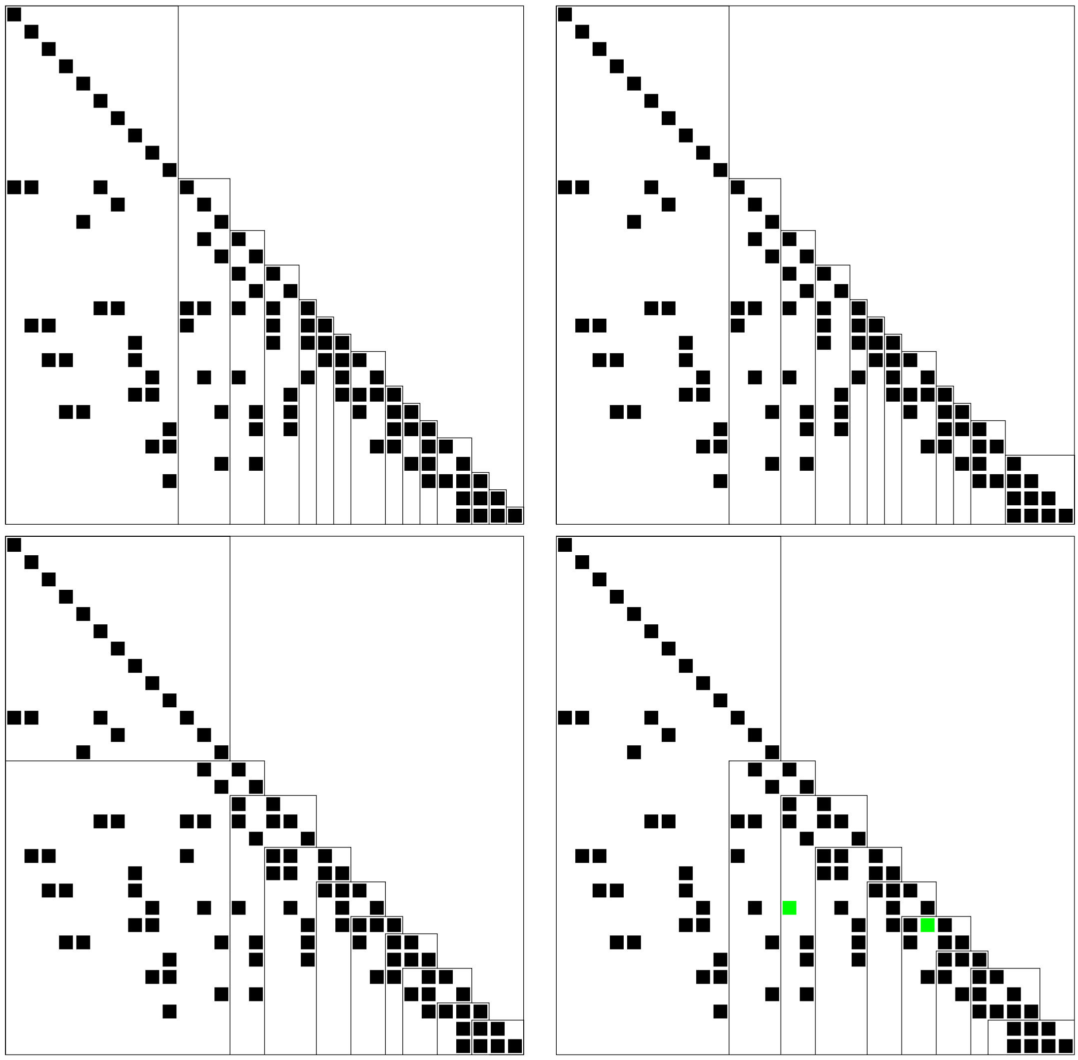

Figure 1: `pores_1`, reordered with 'minimum degree', incomplete factorization with one level of fill, L-matrix, factored using level scheduling: 15 factors, RP2: 12 factors, RPO2: 10 factors, RPOPT: 9 factors

1. *Level scheduling* (see e.g. [12] or [10, pp 346–350]) all nodes are given a level that is equal to the longest path from a root to that node. All nodes of the same level can be eliminated concurrently. A characteristic of level scheduling is that the diagonal blocks are diagonal matrices.

2. *Node partitioning*, a solution to problem 2. The factors are vertical strips in the matrix. We use algorithm RP2 from [2].

3. Γ-*partitioning*, an edge partitioning with usually less factors than in the optimal node partitioning generated by RP2. The factors have the shape of the Greek capital letter. We use algorithm RPO2 from [14].

To illustrate the different partitioning algorithms the lower triangular factor of an

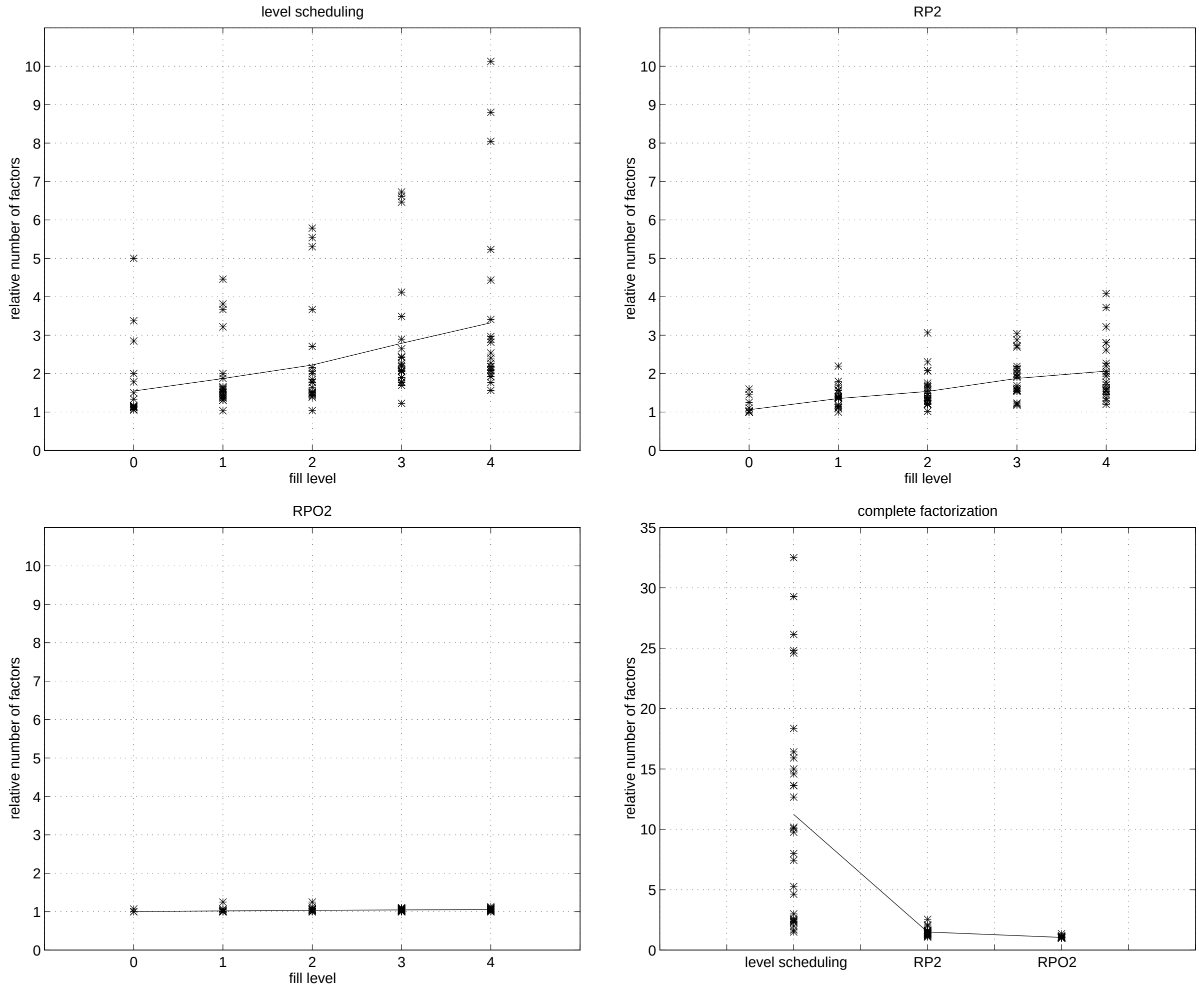

Figure 2: The number of factors compared to the number of factors in the RPOPT partitioning and their averages (the solid lines) for a set of matrices incompletely factored with different levels of fill.

incompletely factored matrix `pores_1` is partitioned and the results are displayed in figure 1. The factors resulting from algorithm RPOPT also have the shape of a Γ, but with possibly extra edges in the next Γ that also belong to this factor. In the figure these edges are colored gray.

In figure 2 the ratio between the number of factors resulting from the other partitioning algorithms and that of RPOPT is presented for the matrices from our test set for several different levels of fill as well as for complete factorization. We see that the number of factors (and thus the number of global communication steps on a massively parallel computer) for level scheduling steadily grows from 1.5 times the number of RPOPT factors on average for zero fill factorizations to 11 times on average for complete factorizations. Compared to RP2 the gain is not as impressive but still more than a factor 2 in quite some cases. RPO2 does quite a fine a job with partitionings that only differ small percentages from RPOPT.

In figure 3 the time needed for RPOPT to partition a triangular matrix is compared to

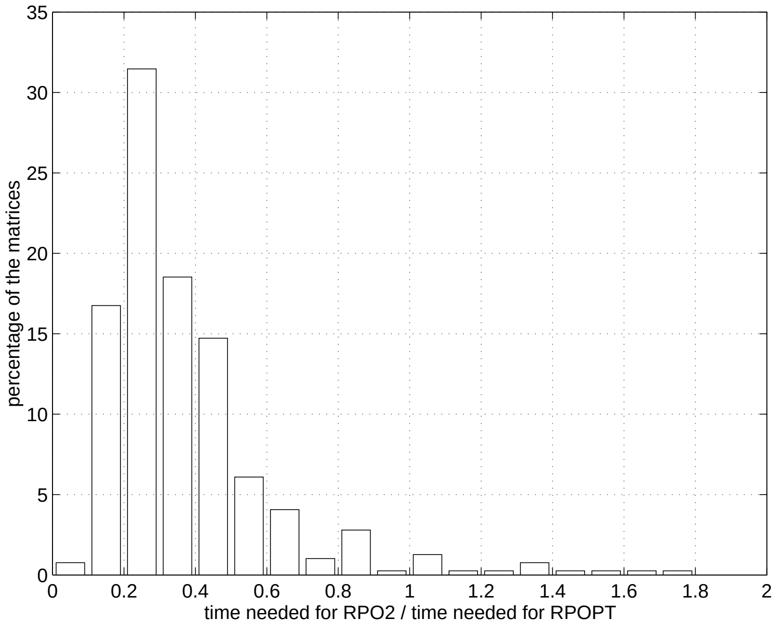

Figure 3: The time needed by RPOPT compared to the time needed by RPO2.

the time needed by algorithm RPO2 for the same matrix. All test matrices were factored using six different levels of fill (no fill, fill level one to four, and a complete factorization). Both the lower and upper triangular matrix were partitioned, resulting in a total test suite of 360 triangular matrices. The timings were done on an HP9000/720 workstation. Since the algorithms are of equal complexity no huge differences in timings are to be expected. Because RPOPT has to look in two directions (checking condition 2 and 3) RPOPT will probably take at least twice as much time as RPO2. These expectations are confirmed by our experiments. On average RPO2 takes 40% of the amount of time needed by RPOPT.

5. Conclusions. In this paper we have presented an algorithm for solving the edge partitioning problem stated as problem 3 in section 1. This problem is closely related to the minimal number of factors problem (problem 1). We have shown that the partitionings resulting from this new algorithm have less (or equal) number of factors as existing partitionings. A number of experiments with the triangular factors from (in)complete factorizations gave an idea about the order of the improvement. These experiments showed that a partitioning with the lowest number of factors can be obtained by using algorithm RPOPT in a time proportional to the time needed by the (less optimal) algorithms RP2 and RPO2.

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.

[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] M.K. Enns, W.F. Tinney, and F.L. Alvarado. Sparse matrix inverse factors. *IEEE Trans. Power Systems*, 5(2):466–473, May 1990.

[6] R.W. Freund. A transpose-free quasi-minimal residual algorithm for non-hermitian linear systems. Technical Report 91.18, RIACS, NASA Ames Research Center, 1991.

[7] M.R. Garey and D.S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, 1979.

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

[9] B.W. Peyton, A. Pothen, and X. Yuan. Partitioning a chordal graph into transitive subgraphs for parallel sparse triangular solution. *Linear Algebra Appl.*, 192:329–353, 1993.

[10] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, Boston, 1995.

[11] A. Sunderland. Parallel solution strategies for triangular systems arising from oil reservoir simulations. In B. Hertzberger and G. Gerazzi, editors, *Proceedings of HPCN Europe 1995, LNCS 919*, pages 148–155, 1995.

[12] H.A. van der Vorst. Large tridiagonal and block tridiagonal linear systems on vector and parallel computers. *Parallel Comput.*, 5:45–54, 1987.

[13] H.A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 13(2):631–644, March 1992.

[14] A.C.N. van Duin. Sparse triangular system partitioning. in preparation, 1997.

A. Source Code. This appendix contains a FORTRAN implementation of RPOPT.

```
      subroutine rpopt ( n, nnz, ka, phgh, la, kat, E, count, a,
     +                   blokaant )
c
c ======================================================================
c
c     Programmer     Arno van Duin
c     Version   1.0  Date 09-06-1997
c
c ***********************************************************************
c
c     KEYWORDS
c
c     sparse
c     triangular matrix
c     reordering
c     decomposition
c     edge-partitioning
c
c ***********************************************************************
c
c     INPUT / OUTPUT PARAMETERS
c
      implicit none
      integer blokaant, n, nnz
      integer ka(nnz+1), phgh(n), kat(nnz+1)
      integer E(n), count(n), la(n)
      double precision a(nnz+1)
c
c     a       io the (reordered) values of the nonzeros of the matrix
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     n        i the dimension of the matrix
c     nnz      i the number of nonzeros in the matrix
c     ka       i the MSC specification of the sparsity pattern
c     la       o pointer per column to first edge of the next factor
c     kat      i the MSR specification of the sparsity pattern
c     phgh     o contains for each factor a pointer to its first column
c
c ***********************************************************************
c
c     LOCAL PARAMETERS
c
      integer itemp(n), kabloc(nnz+1), roots(n), icheck(n), newroots(n)
      integer i, j, jj, iswap, rootaant, newaant, P, fillers, istr,
     +        iend, thiscol, thisrij, thisroot, thisedge, totdeze
      double precision dswap
      logical delay, dezedoen
c
c     delay    if true, current edge causes fill
c     dezedoen false if this edge has been checked already
c     dswap    help real for swapping edges
c     fillers  help variable for keeping track of the fill causing edges
c     i        loop counter
```

```
c     icheck   work array to keep track if this edge has been checked already
c     iend     pointer to first not added edge of this column
c     istr     pointer to last not added edge of this column
c     iswap    help integer for swapping edges
c     itemp    work array for scattering
c     j        loop counter
c     jj       loop counter
c     kabloc   contains for every edge to which factor it belongs
c     newaant  number of columns that have become eligible
c     newroots the numbers of the columns that have become eligible
c     P        current column count
c     rootaant number of columns that can be added to the next factor
c     roots    the numbers of the columns that can be added to the next factor
c     thiscol  a column of this factor
c     thisedge current edge of current column
c     thisrij  the row index of this edge
c     thisroot current column
c     totdeze  pointer to last scattered edge
c
c *********************************************************************
c
c     CALLED SUBROUTINES
c
c     None.
c
c =====================================================================
c
c     --- initializations
c
      blokaant = 0
      rootaant = 0
      newaant = 0
      P = 0
      do i = 1, nnz
         kabloc(i) = 0
      enddo
      phgh(1) = 1

      do j = 1, n
         itemp(j) = 0
         icheck(j) = 0
         count(j) = kat(j+1)-kat(j)
         la(j) = ka(j+1)
         if ( count(j) .eq. 0 ) then
            la(j) = ka(j)
            rootaant = rootaant + 1
            roots(rootaant) = j
            P = P + 1
            E(P) = 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)
c
c          --- add all remaining edges to this factor
c
           do j = la(thisroot), ka(thisroot+1)-1
              thisedge = ka(j)
              kabloc(j) = blokaant
              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
           P = P + 1
           E(P) = thisroot

           do i = kat(thisroot), kat(thisroot+1)-1
              icheck(kat(i)) = 1
           enddo
c
c          --- check fill for all edges starting in this root
c               by checking all columns with edges to this root
c
           do i = kat(thisroot), kat(thisroot+1)-1
c
c             --- scatter the edges in the column that belong to this factor
c
              if ( icheck(kat(i)) .eq. 1 ) then

                 thiscol = kat(i)
                 istr = 1
                 iend = 0
                 if ( la(thiscol) .lt. ka(thiscol+1) .and.
     +                kabloc(la(thiscol)) .eq. blokaant ) then
                    istr = la(thiscol)
                    iend = ka(thiscol+1)-1
                 else if ( kabloc(ka(thiscol)) .eq. blokaant ) then
                    istr = ka(thiscol)
                    iend = la(thiscol)-1
                 endif
```

```
                  dezedoen = .true.
                  totdeze = iend
                  do j = istr, iend
                     itemp(ka(j)) = 1
                     if ( icheck(ka(j)) .eq. 1 ) then
                        dezedoen = .false.
                        totdeze = j
                        goto 10
                     endif
                  enddo
10                continue
c
c                 --- if edge in this col to this root belongs to this factor
c
                  if ( itemp(thisroot) .eq. 1 .and. dezedoen ) then
c
c                    --- for all edges that uptil now did not cause any fill
c
                     jj = ka(thisroot)
                     do j = ka(thisroot), la(thisroot)-1

                        if ( jj .ge. la(thisroot) ) goto 20
c
c                       --- if there is not an edge in this col on the same row
c                           then it will cause fill, swap it to the fill-part
c
                        if ( itemp(ka(jj)) .eq. 0 ) then
                           la(thisroot) = la(thisroot)-1
                           if ( jj .ne. la(thisroot) ) then
                              iswap = ka(la(thisroot))
                              dswap = a(la(thisroot))
                              ka(la(thisroot)) = ka(jj)
                              a(la(thisroot)) = a(jj)
                              ka(jj) = iswap
                              a(jj) = dswap
                           else
                              jj = jj + 1
                           endif
                        else
                           jj = jj + 1
                        endif

                     enddo
20                   continue

                  endif
c
c                 --- reset scatter array
c
                  do j = istr, totdeze
                     itemp(ka(j)) = 0
                  enddo

               endif
            enddo
c
c           --- reset array
```

```
c
          do i = kat(thisroot), kat(thisroot+1)-1
             icheck(kat(i)) = 0
          enddo
c
c         --- check delay if there are fill causing edges
c
          if ( la(thisroot) .lt. ka(thisroot+1) ) then
             fillers = la(thisroot)
c
c            --- for all not fill causing edges, check if there is an edge
c                on this row from a root that is pointed to by any of the
c                fill causing edges, if so: swap to delay part
c
             jj = ka(thisroot)
             do i = ka(thisroot), la(thisroot)-1
                if ( jj .ge. la(thisroot) ) goto 40
                delay = .false.
                thisrij = ka(jj)
                do j = kat(thisrij), kat(thisrij+1)-1
                   if ( kat(j) .gt. thisroot ) then
                      itemp(kat(j)) = 1
                   endif
                enddo
                do j = fillers, ka(thisroot+1)-1
                   if ( itemp(ka(j)) .eq. 1 ) then
                      delay = .true.
                      goto 30
                   endif
                enddo
30              continue
                do j = kat(thisrij), kat(thisrij+1)-1
                   itemp(kat(j)) = 0
                enddo

                if ( delay ) then
                   la(thisroot) = la(thisroot)-1
                   if ( jj .ne. la(thisroot) ) then
                      iswap = ka(la(thisroot))
                      dswap = a(la(thisroot))
                      ka(la(thisroot)) = ka(jj)
                      a(la(thisroot)) = a(jj)
                      ka(jj) = iswap
                      a(jj) = dswap
                   else
                      jj = jj + 1
                   endif
                else
                   jj = jj + 1
                endif
             enddo
40           continue

          endif
c
c         --- add okay edges to factor and update counts
c
```

```
      do j = ka(thisroot), la(thisroot)-1
         thisedge = ka(j)
         kabloc(j) = blokaant
         count(thisedge) = count(thisedge) - 1
         if ( count(thisedge) .eq. 0 ) then
            newaant = newaant + 1
            newroots(newaant) = thisedge
         endif
      enddo

      if ( la(thisroot) .lt. ka(thisroot+1) ) then
         rootaant = rootaant + 1
         roots(rootaant) = thisroot
      endif

   enddo
   phgh(blokaant+1) = P+1

enddo

end
```