



# Preconditioners based on strong subgraphs

IS Duff, K Kaya

April 2013

Submitted for publication in Electronic Transactions on Numerical Analysis

RAL Library  
STFC Rutherford Appleton Laboratory  
R61  
Harwell Oxford  
Didcot  
OX11 0QX

Tel: +44(0)1235 445384  
Fax: +44(0)1235 446403  
email: [libraryral@stfc.ac.uk](mailto:libraryral@stfc.ac.uk)

Science and Technology Facilities Council preprints are available online  
at: <http://epubs.stfc.ac.uk>

**ISSN 1361- 4762**

Neither the Council nor the Laboratory accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigations.

# Preconditioners Based on Strong Subgraphs

Iain S. Duff<sup>1,2</sup> and Kamer Kaya<sup>2,3</sup>

## ABSTRACT

This paper proposes an approach for obtaining block diagonal and block triangular preconditioners that can be used for solving a linear system  $\mathbf{Ax} = \mathbf{b}$ , where  $\mathbf{A}$  is a large, nonsingular, real,  $n \times n$  sparse matrix. The proposed approach uses Tarjan's algorithm for hierarchically decomposing a digraph into its strong subgraphs. To the best of our knowledge, this is the first work that uses this algorithm for preconditioning purposes. We describe the method, analyse its performance, and compare it with preconditioners from the literature such as ILUT and XPABLO and show that it is highly competitive with them in terms of both memory and iteration count. In addition, our approach shares with XPABLO the benefit of being able to exploit parallelism through a version that uses a block diagonal preconditioner.

**Keywords:** sparse matrices, strong subgraphs, strong components, preconditioners

**AMS(MOS) subject classifications:** 05C50, 05C70, 65F50

---

This report is available through the URL <http://www.stfc.ac.uk/CSE/36276.aspx>. It is a preprint of a paper that will appear in the journal Electronic Transactions in Numerical Analysis (ETNA).

<sup>1</sup>R 18, Rutherford Appleton Laboratory, Oxfordshire, OX11 0QX, UK ([iain.duff@stfc.ac.uk](mailto:iain.duff@stfc.ac.uk)). The research of this author was supported in part by the EPSRC Grant EP/I013067/1.

<sup>2</sup>CERFACS, 42 Avenue Gaspard Coriolis, 31057, Toulouse, France ([{duff,kaya}@cerfacs.fr](mailto:{duff,kaya}@cerfacs.fr)).

<sup>3</sup>Current address: Department of Biomedical Informatics, The Ohio State University, Columbus, OH, USA

Scientific Computing Department  
R 18  
Rutherford Appleton Laboratory  
Oxon OX11 0QX

April 12, 2013

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>1</b>
<b>3</b>	<b>A strong subgraph based preconditioner</b>	<b>3</b>
3.1	SCPRE: Obtaining the block diagonal preconditioner . . . . .	4
3.1.1	Tarjan’s algorithm for hierarchical clustering . . . . .	4
3.1.2	HDPRE: Obtaining the initial block structure . . . . .	8
3.1.3	Combining the blocks . . . . .	12
3.2	SCPRE: Extending to a BTF preconditioner . . . . .	12
<b>4</b>	<b>Using SCPRE with an iterative solver</b>	<b>13</b>
4.1	Robustness . . . . .	15
<b>5</b>	<b>Experiments</b>	<b>15</b>
5.1	Experiments with block Gauss-Seidel iterations . . . . .	18
5.1.1	Memory usage . . . . .	19
5.2	Experiments with block Jacobi iterations . . . . .	21
5.3	Cost of generating the preconditioner . . . . .	23
<b>6</b>	<b>Conclusions and future work</b>	<b>23</b>



# 1 Introduction

Given a linear system

$$\mathbf{Ax} = \mathbf{b}, \quad (1.1)$$

where  $\mathbf{A}$  is a real, large, sparse square matrix of order  $n$ , we propose a method to construct a preconditioning matrix  $\mathbf{M}$  to accelerate the solution of the system when using Krylov methods. The proposed method is based on a hierarchical decomposition of the associated digraph into its strong subgraphs. This decomposition can be used to find a permutation of the matrix to produce a block form that can be used to build either a block diagonal matrix for use as a block Jacobi preconditioner or a block tridiagonal matrix for use as a block Gauss Seidel preconditioner.

The algorithm we use to create the blocks on the diagonal of  $\mathbf{M}$  is a modified version of Tarjan's algorithm HD that decomposes a digraph into its strong subgraphs hierarchically (Tarjan 1983). Tarjan assumed that the edges of the digraph are weighted and HD uses this weight information to create the hierarchical decomposition. However, HD requires distinct edge weights if it is implemented as given in Tarjan (1983). In this paper, we propose a slight modification of HD which allows us to handle digraphs whose edge weights are not necessarily distinct. We make further modifications to the algorithm to use it for preconditioning purposes. The strong subgraphs formed by the modified version of HD correspond to the blocks on the diagonal of  $\mathbf{M}$ . To the best of our knowledge, this is the first work that uses Tarjan's hierarchical decomposition algorithm for preconditioning purposes. We call our modified version HDPRE.

We should emphasize at this point that this algorithm of Tarjan is different from the much better known algorithm for obtaining the strong components of a reducible matrix. This earlier algorithm (Tarjan 1972), which we call SCC, is used widely in the solution of reducible systems and is also called by HD and HDPRE which can be viewed as extending the earlier work to irreducible matrices. We use the output from HDPRE to determine our preconditioners. This is done by SCPRE that can generate a block diagonal preconditioner or a block upper-triangular one.

We have conducted several experiments to see the efficiency of the SCPRE algorithm. We compare the number of iterations for convergence and the memory requirement of the GMRES (Saad and Schultz 1986) iterative solver when the proposed approach and a set of ILUT preconditioners (Saad 1994, Saad 2003) are used. We are aware that block based preconditioning techniques have been studied before and successful preconditioners such as PABLO and its derivatives have been proposed (Fritzsche 2010, Fritzsche, Frommer and Szyld 2007). These preconditioners were successfully used for several matrices (Benzi, Choi and Szyld 1997, Choi and Szyld 1996, Dayar and Stewart 2000). In this paper, we compare our results also with XPABLO (Fritzsche 2010, Fritzsche et al. 2007).

Section 2 gives the notation used in the paper and background on Tarjan's algorithm HD. The proposed algorithm is described in Section 3 and the implementation details are given in Section 4. Section 5 gives the experimental results and Section 6 concludes the paper.

## 2 Background

Let  $\mathbf{A}$  be a large, nonsingular,  $n \times n$  sparse matrix with  $m$  off-diagonal nonzeros. The digraph  $G = (V, E)$ , associated with  $\mathbf{A}$ , has  $n$  vertices,  $v_i, i = 1, n$ , in its vertex set  $V$  where  $v_i$  corresponds to the  $i$ th row/column of  $\mathbf{A}$  for  $1 \leq i \leq n$  and  $v_i v_j$  is in the edge set  $E$  iff  $\mathbf{A}_{ij}$  is nonzero, for

$1 \leq i \neq j \leq n$ . Note that we do not consider self-loops of the form  $v_i v_i$  corresponding to diagonal entries in the matrix. Figure 2.1 shows a simple  $6 \times 6$  matrix with 13 nonzeros and its associated digraph.

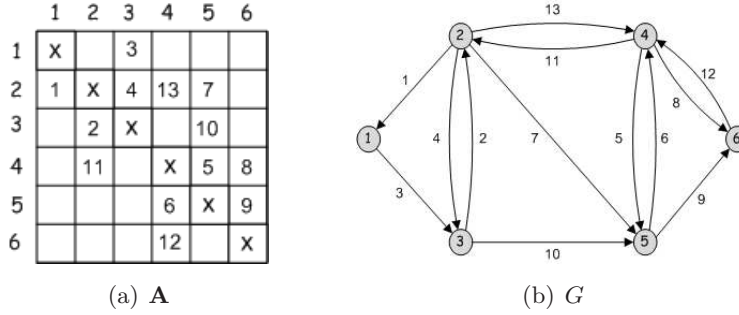


Figure 2.1: A  $6 \times 6$  matrix  $\mathbf{A}$  with 13 off-diagonal nonzeros on the left and its associated digraph  $G$  on the right. The nonzeros on the diagonal of  $\mathbf{A}$  are shown with  $\times$ . Except for these entries, there is an edge in the associated digraph  $G$  for each nonzero of  $\mathbf{A}$ .

A *path* is sequence of vertices such that there exists an edge between every two consecutive vertices. A path is called *closed* if its first and last vertex are the same. A vertex  $u$  is *connected* to  $v$  if there is a path from  $u$  to  $v$  in  $G$ . A directed graph  $G$  is *strongly connected* if  $u$  is connected to  $v$  for all  $u, v \in V$ . Note that a digraph with a single vertex  $u$  is strongly connected. A digraph  $G' = (V', E')$  is a subgraph of  $G$  if  $V' \subset V$  and  $E' \subseteq E \cap (V' \times V')$ . If  $G'$  is strongly connected it is called a strong subgraph (or a strongly connected subgraph) of  $G$ . Furthermore, if  $G'$  is *maximally* strongly connected, i.e., if there is no strong subgraph  $G''$  of  $G$  such that  $G'$  is a subgraph of  $G''$ , it is called a strong component (or a strongly connected component) of  $G$ . If the matrix  $\mathbf{A}$  cannot be permuted into a block triangular form (BTF) by simultaneous row and column permutations, i.e., if the associated digraph is strongly connected, we say that  $\mathbf{A}$  is irreducible. Otherwise, we call it reducible.

Let  $G = (V, E)$  be a digraph and  $\mathcal{P}(V) = \{V_1, V_2, \dots, V_k\}$  define a partition of  $V$  into disjoint sets, i.e.,  $V_i \cap V_j = \emptyset$  for  $i \neq j$  and  $\cup_{i=1}^k V_i = V$ . Let  $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2\}$  be a set of two vertex partitions such that  $\mathcal{V}_1 = \mathcal{P}(V)$  and

$$\mathcal{V}_2 = \bigcup_{V_i \in \mathcal{V}_1} \mathcal{P}(V_i),$$

i.e.,  $\mathcal{V}_2$  is a finer partition obtained from partitioning the parts in  $\mathcal{V}_1$ . Hence, if  $\mathcal{V}_1 = \{\{1, 2, 3\}, \{4, 5, 6\}\}$  then  $\mathcal{V}_2$  can be  $\{\{1\}, \{2, 3\}, \{4, 5\}, \{6\}\}$  but cannot be  $\{\{1, 2\}, \{3, 4\}, \{5, 6\}\}$ . Let  $\text{no}_1(v)$  and  $\text{no}_2(v)$  denote the index of the part containing vertex  $v \in V$  for  $\mathcal{V}_1$  and  $\mathcal{V}_2$ , respectively.

Let **condense** be an operation which takes  $G$  and  $\mathcal{V}$  as inputs and returns a condensed digraph  $\text{condense}(G, \mathcal{V}) = G^\mathcal{V} = (V^\mathcal{V}_2, E^\mathcal{V}_1)$  where each vertex set  $V_i \in \mathcal{V}_1$  is condensed into a single vertex  $\nu_i \in V^\mathcal{V}_2$ . For all  $uv \in E$ , with  $\text{no}_2(u) = i$  and  $\text{no}_2(v) = j$  there exists  $\nu_i \nu_j \in E^\mathcal{V}_1$  if and only if  $\text{no}_1(u) \neq \text{no}_1(v)$ , i.e.,  $u$  and  $v$  are in different coarse parts. Note that even though  $G$  is a simple digraph,  $G^\mathcal{V}$  can be a directed multigraph, i.e., there can be multiple edges between two vertices. The definitions of connectivity and strong connectivity in directed multigraphs are the same as those in digraphs. An example for the **condense** operation is given in Figure 2.2.

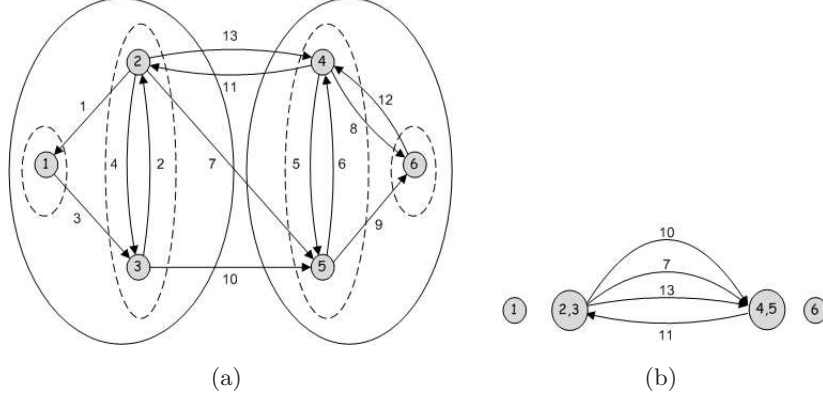


Figure 2.2: An example for the **condense** operation on the digraph in Figure 1(b). The partitions  $\mathcal{V}_1 = \{\{1, 2, 3\}, \{4, 5, 6\}\}$  and  $\mathcal{V}_2 = \{\{1\}, \{2, 3\}, \{4, 5\}, \{6\}\}$  are shown in (a). The condensed graph is shown in (b).

### 3 A strong subgraph based preconditioner

Our proposed algorithm, **SCPRE**, generates a preconditioner  $\mathbf{M}$  with a block diagonal or block upper-triangular structure where the size of each block is smaller than a requested maximum block size  $mbs$ . For the experiments, we scale and permute  $\mathbf{A}$  from (1.1) by Duff and Koster's **MC64** with the option that uses the *maximum product transversal* (Duff and Koster 2001). The idea used by **MC64** is due to Olschowka and Neumaier (1996) who propose an algorithm which permutes and scales the matrix in such a way that the magnitudes of the diagonal entries are one and the magnitudes of the off-diagonal entries are all less than or equal to one. Such a matrix is called an *I-matrix*. For direct methods, it has been observed that the more dominant the diagonal of a matrix, the higher the chance that diagonal entries are stable enough to serve as pivots for elimination. For iterative methods, as previous experiments have shown, such a scaling is also of interest (Benzi, Haws and Tuma 2000, Duff and Koster 2001). We observed a similar behaviour in our preliminary experiments and used **MC64** for scaling and permutation of the original matrix. From now on, we will assume that the diagonal of  $\mathbf{A}$  is nonzero since this is the case after this permutation.

**SCPRE** uses the block structure from **HDPRE** to determine the diagonal blocks of the preconditioner  $\mathbf{M}$ . We then combine some of these blocks if the combination has fewer than  $mbs$  rows/columns and the combination is not block diagonal. The diagonal blocks of the resulting matrix can then be used to precondition the iterative solver using the block Jacobi algorithm and can exploit parallel architectures as the blocks are independent. If we require a block diagonal preconditioner, then we are finished. Otherwise, **SCPRE** permutes the blocks and builds a block upper-triangular preconditioner.

If  $\mathbf{A}$  is reducible and the maximum block size in the BTF of  $\mathbf{A}$  is less than or equal to  $mbs$ , then **SCPRE** will find this form or will return the diagonal blocks of it if a block diagonal preconditioner is desired. The permutation of a matrix into its block triangular form is a well known technique which has been recently and successfully used by direct and iterative solvers for circuit simulation matrices (Davis and Natarajan 2010, Thornquist, Keiter, Hoekstra, Day and Boman 2009), which they state can often be permuted to a non-trivial BTF. For some applications, such as DC operating point analysis, the block triangular form has many but small

blocks (Thornquist et al. 2009). Such a matrix is usually easy to factorize if we initially permute it to BTF, so that a direct solver like KLU (Davis and Natarajan 2010) only needs to factorize the diagonal blocks. Note that Tarjan’s SCC algorithm that has linear complexity in matrix order and number of nonzeros has been widely and successfully used by the computational linear algebra community for obtaining a BTF that is then exploited by subsequent solvers. A code implementing this algorithm is available as MC13 from HSL (HSL 2011) and is also an algorithm in ACM TOMS (Duff and Reid 1978a, Duff and Reid 1978b).

However, when the matrix is irreducible, the SCC algorithm is not applicable. Furthermore, even if the matrix is reducible, we may have little gain from using the BTF because this form may have one or more very large blocks. This is the case for applications like transient simulation or for circuit matrices with feedbacks. For this reason we propose using Tarjan’s HD algorithm (Tarjan 1983) as an additional tool to SCC. SCPRE uses HDPRE and further decomposes blocks larger than  $mbs$  to make the resulting preconditioner practical. For these reasons, in our experiments, we only use matrices that are irreducible or have a large block in their BTF. The details of SCPRE and the algorithms it uses are given in the next section. Note that since we use a combinatorial algorithm from graph theory for preconditioning purposes, we will use terms from graph theory in the following text so that *row/column* and *vertex* are used interchangeably, as well as *nonzero* and *edge*.

### 3.1 SCPRE: Obtaining the block diagonal preconditioner

To obtain a block diagonal preconditioner, SCPRE uses HDPRE and then combines some of these blocks if the size of the combined block is at most  $mbs$  and the combined block is not block diagonal. In this section, we give the details of these algorithms. First, we describe Tarjan’s hierarchical decomposition algorithm in detail.

#### 3.1.1 Tarjan’s algorithm for hierarchical clustering

Let  $G = (V, E)$  be the digraph associated with  $\mathbf{A}$ . The weight of an edge  $uv \in E$  is denoted by  $w(uv)$  and is set to the absolute value of the corresponding off-diagonal nonzero. Hence, there are  $m$  edges and all of the edges have positive weights. A hierarchical decomposition of  $G$  into its strong subgraphs can be defined in the following way. Let  $\sigma_0$  be a permutation of the edges. For  $1 \leq i \leq m$ , let  $\sigma_0(i)$  be the  $i$ th edge in  $\sigma_0$  and  $\sigma_0^{-1}(uv)$  be the index of the edge  $uv$  in the permutation for all  $uv \in E$ . Let  $G_0 = (V, \emptyset)$  be the graph obtained by removing all the edges from  $G$ . We then add edges one by one to  $G_0$  in the order determined by  $\sigma_0$ . Let  $G_i = (V, \{\sigma(j) : 1 \leq j \leq i\})$  be the digraph obtained after the addition of the first  $i$  edges. Initially in  $G_0$ , there are  $n$  strong components, one for each vertex, and during the edge addition process, the strong components gradually coalesce until there is only one, as we are assuming that  $\mathbf{A}$  is irreducible. Note that if this is not the case, the algorithm will be used for the large irreducible blocks in  $\mathbf{A}$ . The hierarchical decomposition of  $G$  into its strong subgraphs with respect to the edge permutation  $\sigma_0$  shows which strong components are formed in this process hierarchically. Note that a strong component formed in this edge addition process is indeed a strong component of some digraph  $G_i$  but not of  $G$ . For  $G$  all except the last are just strong subgraphs.

A hierarchical decomposition can be represented by a hierarchical decomposition tree  $T$ ,

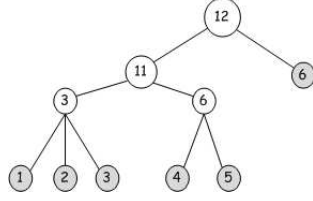


Figure 3.3: The hierarchical decomposition tree for the digraph  $G$  and the permutation given by the edge ordering in Figure 1(b).

whose leaf nodes correspond to the vertices in  $V$ , non-leaf nodes correspond to edges in  $E$ , and subtrees correspond to the decomposition trees of the strong components that form as the process proceeds. Note that only the edges that create strong components during the process have corresponding internal nodes in  $T$ . If  $\sigma_0$  is the ordering determined by the edge numbers, the hierarchical decomposition tree for the digraph in Figure 1(b) is given in Figure 3.3. As the figure shows, during the edge addition process, after the addition of the 3rd and 6th edges in  $\sigma_0$ , the sets of vertices  $\{1, 2, 3\}$  and  $\{4, 5\}$  form a strong component of  $G_3$  and  $G_6$ , respectively. These strong components are then combined and form a larger one after the addition of the 11th edge. In Figure 3.3, the root of the tree is labelled with 12. Hence the first 12 edges in  $\sigma_0$  are sufficient to construct a strongly connected digraph. For the figures in this paper, we use the labels of the corresponding vertices and the  $\sigma_0^{-1}$  values of the corresponding edges to label each leaf and non-leaf node of a hierarchical decomposition tree, respectively.

Given a digraph  $G = (V, E)$  with  $n$  vertices and  $m$  edges, and a permutation  $\sigma_0$ , the hierarchical decomposition tree  $T$  can be obtained by first constructing  $G_0$  and executing SCC for each internal digraph  $G_i$  obtained during the edge addition process. Note that this is an  $\mathcal{O}(mn + m^2)$  algorithm since  $1 \leq i \leq m$  and the cost of SCC is  $\mathcal{O}(n + m)$  due to the strong component algorithm of Tarjan (1972). To obtain  $T$  in a more efficient way, Tarjan first proposed an  $\mathcal{O}(m \log^2 n)$  recursive algorithm (Tarjan 1982) and later improved his algorithm and reduced the complexity to  $\mathcal{O}(m \log n)$  (Tarjan 1983). He assumed that the weights of the edges in the digraph are distinct, i.e.,  $w(uv) \neq w(u'v')$  for two distinct edges  $uv$  and  $u'v'$ . Here we modify the description of the algorithm so that it also works for the case when some edges have equal weights. Note that the connectivity of the digraph is purely structural and is independent of the edge weights. The only role that they play in Tarjan's algorithm HD, is in the preprocessing step that defines a permutation  $\sigma_0$  of the edges and in determining the ordering of the edges during the course of the algorithm. We eliminate the necessity of this latter use by avoiding numerical comparisons through just using the indices of the edges with respect to  $\sigma_0$ . With this slight modification, the algorithm remains correct even when some edges have the same weight which is very important as many matrices have several or many nonzeros with the same numerical value.

HD uses a recursive approach and for every recursive call, it gets a digraph  $G = (V, E)$ , a permutation  $\sigma$  of the edges, and a parameter  $i$  as inputs such that  $G$  is strongly connected and  $G_i$  is known to be acyclic, i.e., every vertex is a separate strong component (Tarjan 1983). For the initial call,  $i$  is set to 0 and the initial permutation is set to  $\sigma_0$  which is a permutation of all the edges in the original digraph.

For a call of  $\text{HD}(G = (V, E), \sigma, i)$ , the *size* of the subproblem is set to  $|E| - i$ , the number of edges that remain to be investigated (Tarjan used the term *rank* to denote the size of a subproblem). Note that in the first step, HD knows that  $G_i$  (that is  $G_0$ ) is acyclic, that is there are  $|V|$  strong

components of  $G_0$ , one for each vertex. If the subproblem size is one, since  $G$  is strongly connected and  $G_i$  is acyclic, the vertices in  $V$  are combined with the addition of the  $|E|$ th edge in  $\sigma$ . Hence HD returns a tree  $T$  having a root labelled with  $\sigma_0^{-1}(\sigma(|E|))$  and  $|V|$  leaves. If the subproblem size is not one, HD performs a binary chop and checks if  $G_j$ ,  $j = \lceil (i + |E|)/2 \rceil$  is strongly connected. If  $G_j$  is strongly connected, then all of the strong components will be combined before the addition of the  $(j + 1)$ th edge. Hence the algorithm calls  $\text{HD}(G_j, \sigma, i)$ . Otherwise, a recursive call is made for each strong component of size larger than one. A detailed pseudo-code of HD is given in Algorithm 1.

---

**Algorithm 1**  $T = \text{HD}(G = (V, E), \sigma, i)$  . For the initial call,  $\sigma = \sigma_0$  and  $i = 0$ .

---

```

1: if  $|E| - i = 1$  then
2:   Let  $T$  be a tree with  $V$  leaves. Root is labelled with  $\sigma_0^{-1}(\sigma(|E|))$ 
3:   return  $T$ 
4: end if
5:  $j = \lceil (i + |E|)/2 \rceil$ 
6: if  $G_j = (V, \{\sigma(k) : 1 \leq k \leq j\})$  is strongly connected then
7:   return  $T = \text{HD}(G_j, \sigma, i)$ 
8: else
9:   for each strong component  $SC_\ell = (V_\ell, E_\ell)$  of  $G_j$  do
10:    if  $|V_\ell| > 1$  then
11:       $\sigma_\ell =$  the permutation of  $E_\ell$  ordered with respect to  $\sigma$ 
12:      if  $i = 0$  or  $(\sigma^{-1}(uv) > i, \forall uv \in E_\ell)$  then
13:         $i_\ell = 0$ 
14:      else
15:         $i_\ell = \max\{k : \sigma^{-1}(\sigma_\ell(k)) \leq i\}$ 
16:      end if
17:       $T_\ell = \text{HD}(SC_\ell, \sigma_\ell, i_\ell)$ 
18:    else
19:       $T_\ell = (V_\ell, \emptyset)$ 
20:    end if
21:  end for
22:   $\mathcal{V}_1 = \mathcal{V}_2 = \{V_\ell : SC_\ell \text{ is a strong component of } G_j\}$ 
23:   $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2\}$ 
24:   $G^\mathcal{V} = \text{condense}(G, \mathcal{V}) = (V^{\mathcal{V}_2}, E^{\mathcal{V}_1})$ 
25:   $\sigma^\mathcal{V} =$  the permutation of  $E^{\mathcal{V}_1}$  ordered with respect to  $\sigma$ 
26:  if  $(\sigma^{-1}(uv) > j, \forall uv \in E^{\mathcal{V}_1})$  then
27:     $i^\mathcal{V} = 0$ 
28:  else
29:     $i^\mathcal{V} = \max\{k : \sigma^{-1}(\sigma^\mathcal{V}(k)) \leq j\}$ 
30:  end if
31:   $T^\mathcal{V} = \text{HD}(G^\mathcal{V}, \sigma^\mathcal{V}, i^\mathcal{V})$ 
32:  replace the leaves of  $T^\mathcal{V}$  with the corresponding trees  $T_\ell$ 
33:  return  $T^\mathcal{V}$ 
34: end if

```

---



By the definition of  $i$ ,  $G_i$ , the subgraph containing the first  $i$  edges of  $G$  in  $\sigma$ , is known to be acyclic. Let  $i_\ell$  be the number of these edges in the  $\ell$ th strong subgraph  $SC_\ell = (V_\ell, E_\ell)$  of  $G_j$ , i.e.,  $i_\ell = |\{uv \in E_\ell : \sigma^{-1}(uv) \leq i\}|$ . Since  $SC_\ell$  is a subgraph of  $G_i$ ,  $G_i$  being acyclic implies that the subgraph of  $SC_\ell$  containing only these  $i_\ell$  edges is also acyclic. In Algorithm 1, lines 12–16 find  $i_\ell$  for each strong component  $SC_\ell$ . This value is then used in the recursive call for  $SC_\ell$  at line 17.

Since  $G_j$  has more than one strong component and  $G$  is known to be strongly connected, with the addition of some edge(s) after the  $j$ th one at least two strong components of  $G_j$  will be combined. To find this edge, another recursive call,  $\text{HD}(G^\mathcal{V}, \sigma^\mathcal{V}, i^\mathcal{V})$ , is made for the condensed graph  $G^\mathcal{V} = (V^\mathcal{V}_2, E^\mathcal{V}_1)$ . Since each strong component of  $G_j$  is reduced to one vertex in  $G^\mathcal{V}$ , a subgraph of the condensed graph which contains only the edges from  $G_j$  must be acyclic. Hence we can find the  $i^\mathcal{V}$  value in a similar fashion to  $i_\ell$ . But this time instead of  $i$  we use  $j$  and set  $i^\mathcal{V} = |\{uv \in E^\mathcal{V}_1 : \sigma^{-1}(uv) \leq j\}|$  for the corresponding recursive call at line 31.

We investigate the size of each new subproblem for the complexity analysis of  $\text{HD}$ . At line 7 of Algorithm 1, the size of the subproblem becomes at most  $j - i$  and for lines 17 and 31, there will be smaller subproblems with size at most  $j - i$  and  $|E| - j$ , respectively. By definition of  $j$ , every subproblem has a size at most  $\frac{2}{3}$  of the original problem size (consider the case when  $i = 0$  and  $|E| = 3$ ). Note that every edge in the original problem corresponds to an edge in at most one subproblem and, if we do not count the recursive calls, the rest of the algorithm takes  $\mathcal{O}(|E|)$ . Let  $|E| = m$ ,  $\mathfrak{t}(m, r)$  be the total complexity of a problem with  $m$  edges and  $r$  problem size, and  $k$  be the number of recursive calls. Then

$$\mathfrak{t}(m, r) = \mathcal{O}(m) + \sum_{i=1}^k \mathfrak{t}(m_i, r_i).$$

Since  $\sum_{i=1}^k m_i \leq m$  and  $r_i \leq 2r/3$  for  $1 \leq i \leq k$ , an easy induction shows that  $\mathfrak{t}(m, r) = \mathcal{O}(m \log r)$ . Hence the total complexity of the algorithm is  $\mathcal{O}(m \log m)$  which is actually  $\mathcal{O}(m \log n)$  since the original graph is a simple digraph (not a directed multigraph).

Let us sketch the algorithm for the digraph  $G = (V, E)$  in Figure 1(b). Assume that  $\sigma_0$  is the ordering described in that figure. In the initial call, line 5 of Algorithm 1 computes  $j = 7$  and checks if  $G_7$  is strongly connected. As Figure 3.4 shows,  $G_7$  has three strong components where the first and second are the new subproblems solved recursively. Since the third strong component contains only one vertex,  $\text{HD}$  does not make a recursive call for it. An additional recursive call is made for the condensed graph. Figure 3.5 shows the graphs for the recursive calls and the returned trees. The number of edges in Figs. 5(a), 5(b) and 5(c) are 4, 2 and 7, whereas the corresponding problem sizes are 4, 2 and 6 respectively. Note that  $i_1$  and  $i_2$  are 0 for the first two calls and  $i^\mathcal{V} = 1$  for the last one with  $G^\mathcal{V}$  since  $G_1^\mathcal{V}$  is known to be acyclic because  $j = 7$  and  $\sigma^\mathcal{V}(1) = \sigma(7)$ .

Because of the multiple edges between two vertices, the condensed graph in Figure 5(c) has 7 edges. However, the algorithm still works if we sparsify the edges of  $G^\mathcal{V} = (V^\mathcal{V}_2, E^\mathcal{V}_1)$  and obtain a simple digraph as follows: For a  $uv \in E$  such that  $u \in V_i$  and  $v \in V_j$  and  $i \neq j$ , there exists  $v_i v_j \in E^\mathcal{V}_1$  if no other  $u'v' \in E$  exists such that  $u' \in V_i$  and  $v' \in V_j$  and  $\sigma^{-1}(u'v') < \sigma^{-1}(uv)$ . That is, for multiple edges between  $u$  and  $v$ , we delete all but the first in the permutation  $\sigma$ . In Figure 5(c), these edges,  $\sigma(7)$  and  $\sigma(8)$ , are shown in bold. In (Tarjan 1983), Tarjan states that although having less edges in the condensed graphs with this modification is desirable, in

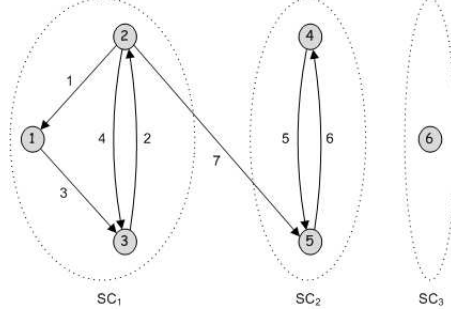


Figure 3.4: Strong components of  $G_7$  for the digraph  $G$  given in Figure 1(b)

practice the added simplicity does not compensate for the cost of the reduction of multigraphs to simple digraphs. This is also validated by our preliminary experiments.

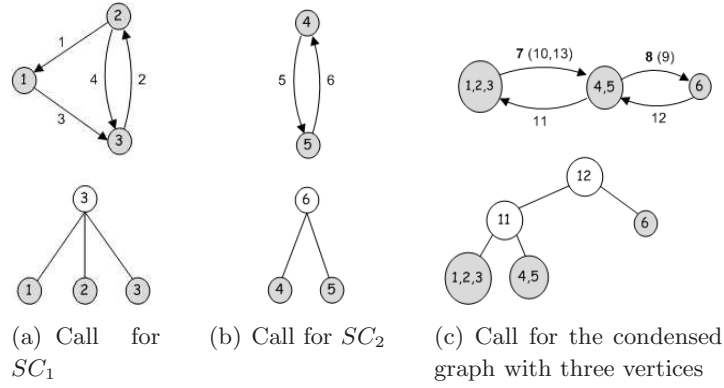


Figure 3.5: Three recursive calls for the digraph  $G$  and  $\sigma_0$  in Figure 1(b). Internal nodes in trees are labelled with the  $\sigma_0^{-1}$  value of the corresponding edge. Note that the overall hierarchical decomposition tree is already given in Figure 3.3.

### 3.1.2 HDPRE: Obtaining the initial block structure

As mentioned in Section 3.1.1, Tarjan proposed HD for hierarchical clustering purposes and sorted the edges with respect to increasing edge weights. Thus, if  $\sigma_0$  is the permutation used for hierarchical clustering  $w(\sigma_0(i)) \leq w(\sigma_0(j))$ , for  $i < j$ . In this work, we propose using two different approaches to obtain the permutation: the first solely depends on the weights of the edges and sorts them in the order of decreasing edge weights, i.e., we define the permutation  $\sigma$  such as  $w(\sigma(i)) \geq w(\sigma(j))$  if  $i < j$ . The second uses the sparsity pattern of the matrix. The reverse Cuthill-McKee (RCM) ordering (Cuthill and McKee 1969) is used to find a symmetric row/column permutation. Then the edges are ordered in a natural, row-wise order. That is, an edge  $ij$  always comes before  $k\ell$  if  $i < k$  or,  $i = k$  and  $j < \ell$ .

The decomposition tree  $T$ , as output from Tarjan's HD algorithm, could be used for preconditioning without modification, but we postprocess this tree to ensure that all leaf nodes are as large as they can be but still have fewer than  $mbs$  nodes. For the decomposition tree  $T$  in Figure 3.3, the cases for  $mbs = 2$  and  $mbs = 3$  are given in Figure 3.6. In  $T$ , for the case  $mbs = 2$ , vertices 1, 2, and 3 cannot be combined since the number of vertices in the combined component will be 3, which is greater than  $mbs$ . Hence, there will be 5 blocks after this phase. However,



vertices 1, 2, and 3 can be combined for the case  $mbs = 3$  and the number of blocks will be 3. Note that for preconditioning, we do not need to construct the whole tree of HD. We only need to continue hierarchically decomposing the blocks until they contain at most  $mbs$  vertices. Hence, for efficiency we modify line 10 of HD to check if the current strong component has more than  $mbs$  vertices (instead of a single vertex). Hence the modified algorithm will make a recursive call for a strong component if and only if the component has more than  $mbs$  vertices.

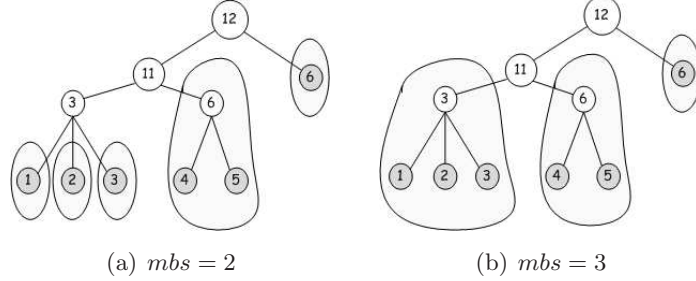


Figure 3.6: Using the output of HD algorithm. Two cases,  $mbs = 2$  and  $mbs = 3$ , are shown for the decomposition tree in Figure 3.3.

To obtain denser and larger blocks, we incorporate some more modifications to HD as follows: first, we modify the definition of  $\mathcal{V}$ . Note that  $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2\}$  for HD, where the parts in  $\mathcal{V}_1 = \mathcal{V}_2$  are the vertex sets of the strong components of  $G_j$ . For preconditioning, we keep the definition of  $\mathcal{V}_1$  but use a finer partition  $\mathcal{V}_2$  that contains the vertex sets of strong components obtained by hierarchically decomposing the strong components of size larger than  $mbs$ . For example, in Figure 3.4, we have 3 strong components of sizes 3, 2 and 1, respectively. Hence,  $\mathcal{V}_1 = \{\{1, 2, 3\}, \{4, 5\}, \{6\}\}$ . If  $mbs = 2$ ,  $SC_1$  will be further divided so  $\mathcal{V}_2 = \{\{1\}, \{2\}, \{3\}, \{4, 5\}, \{6\}\}$ . However, if  $mbs = 3$  no more decomposition will occur and  $\mathcal{V}_1$  will be equal to  $\mathcal{V}_2$ . With this modification, the algorithm will try to combine the smaller strong components and obtain larger ones with at most  $mbs$  vertices. Note that setting  $\mathcal{V} = \{\mathcal{V}_2, \mathcal{V}_2\}$  tries to do the same but will fail since the only components that can be formed by this approach will be the same as those in  $\mathcal{V}_1$ . Hence, by deleting the edges within the vertex sets in  $\mathcal{V}_1$ , we eliminate the possibility of obtaining the same components.

A second modification is applied to the **condense** operation by deleting the edges between two vertices  $\nu_i, \nu_j \in V^{\mathcal{V}_1}$  in the condensed graph  $G^{\mathcal{V}}$ , if the total size of the corresponding parts  $V_i, V_j \in \mathcal{V}_2$  is larger than  $mbs$ . Note that if we were to retain these edges, they would only be used to form blocks of size more than  $mbs$ . We call this modified **condense** operation **pcondense**. An example of the difference between **condense** and **pcondense** is given in Figure 3.7.

As Figure 3.7 shows, with this last modification, some of the graphs for the recursive calls may not be strongly connected. Hence, instead of a whole decomposition tree, we may obtain a forest such that each tree in the forest, which corresponds to a strong subgraph in the hierarchical decomposition, has less than  $mbs$  leaves. The modified algorithm **HDPRE**, described in Algorithm 2, also handles digraphs which are not strongly connected. Note that, for preconditioning, the only information we need is the block structure information. That is, we need to know which vertex is in which tree in the forest after the modified hierarchical decomposition algorithm is performed. Instead of a tree (or a forest), **HDPRE** returns this information in the *scomp* array.

The structure of the algorithm **HDPRE** is similar to that of **HD**. In addition to  $G$ ,  $\sigma$  and  $i$ , **HDPRE**

---

**Algorithm 2**  $scomp = \text{HDPRE}(G = (V, E), \sigma, i, vsize)$  ( $mbs$  is global,  $i = 0$  for the initial call).

---

```

1: if  $|E| - i = 1$  then
2:   find strong components of  $G$ 
3:   for each strong component  $SC_\ell = (V_\ell, E_\ell)$  of  $G$  do
4:     if  $\sum_{v \in V_\ell} vsize(v) > mbs$  then
5:       consider each  $v \in V_\ell$  as a strong component
6:     else
7:        $\forall v \in V_\ell, scomp(v) = \ell$ 
8:     end if
9:   end for
10:  return  $scomp$ 
11: end if
12:  $j = \lceil (i + |E|)/2 \rceil$ 
13: if  $G_j = (V, \{\sigma(k) : 1 \leq k \leq j\})$  is strongly connected then
14:  return  $scomp = \text{HDPRE}(G_j, \sigma, i, vsize)$ 
15: else
16:  for each strong component  $SC_\ell = (V_\ell, E_\ell)$  of  $G_j$  do
17:    if  $\sum_{v \in V_\ell} vsize(v) > mbs$  then
18:       $\sigma_\ell =$  the permutation of  $E_\ell$  ordered with respect to  $\sigma$ 
19:      compute  $i_\ell$  as in Algorithm 1
20:       $vsize_\ell(v) = vsize(v), \forall v \in V_\ell$ 
21:       $scomp_\ell = \text{HDPRE}(SC_\ell, \sigma_\ell, i_\ell, vsize_\ell)$ 
22:      update  $scomp$  according to  $scomp_\ell$ 
23:    end if
24:  end for
25:   $\mathcal{V}_1 = \{V_\ell : SC_\ell \text{ is a strong component of } G_j\}$ 
26:   $\mathcal{V}_2 = \{V_{\ell'} : SC_{\ell'} = (V_{\ell'}, E_{\ell'}) \text{ is a strong component in } scomp\}$ 
27:   $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2\}$ 
28:   $G^\mathcal{V} = \text{pcondense}(G, \mathcal{V}, mbs) = (V^{\mathcal{V}_2}, E^{\mathcal{V}_1})$ 
29:   $\sigma^\mathcal{V} =$  the permutation of  $E^{\mathcal{V}_1}$  ordered with respect to  $\sigma$ 
30:  compute  $i^\mathcal{V}$  as in Algorithm 1
31:  if  $i^\mathcal{V} \neq |E^{\mathcal{V}_1}|$  then
32:     $vsize^\mathcal{V}(v_{\ell'}) = \sum_{v \in V_{\ell'}} vsize(v), \forall V_{\ell'} \in \mathcal{V}_2$ 
33:     $scomp^\mathcal{V} = \text{HDPRE}(G^\mathcal{V}, \sigma^\mathcal{V}, i^\mathcal{V}, vsize^\mathcal{V})$ 
34:    update  $scomp$  with respect to  $scomp^\mathcal{V}$ 
35:  end if
36:  return  $scomp$ 
37: end if

```

---

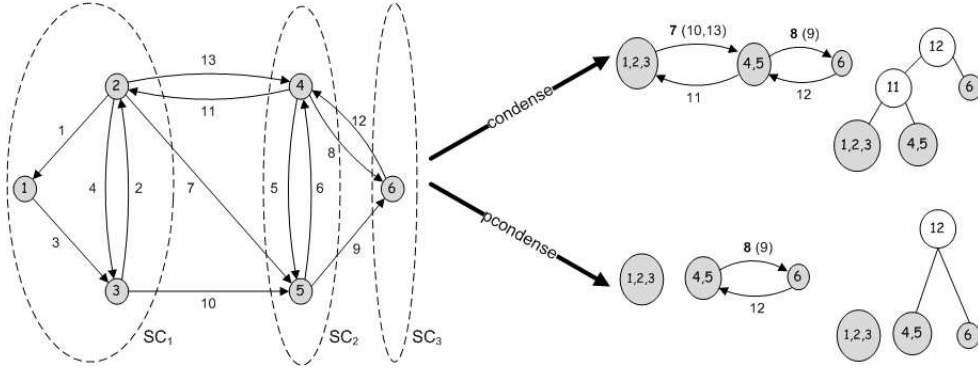


Figure 3.7: Difference between **condense** and **pcondense** operations for the strong components of  $G_7$  given in Figure 3.4. Let  $mbs = 3$  so all of the components have a desired number of vertices and  $\mathcal{V}_1 = \mathcal{V}_2 = \{\{1, 2, 3\}, \{4, 5\}, \{6\}\}$ . Note that the condensed graphs obtained by **condense** and **pcondense** are the same except that the latter does not have some of the edges that the former has. For this example, the edges 7, 10, 11 and 13 are missing since the total size of  $SC_1$  and  $SC_2$  is 5, which is greater than  $mbs$ . As a result, for the **condense** graph, we obtain 3 blocks of sizes 3, 2 and 1, respectively; whereas, for the **pcondense** graph, we have 2 blocks of size 3.

requires an additional input array *vsiz*e which stores the number of vertices condensed into each vertex of  $V$ . For the initial call with  $G = (V, E)$ , *vsiz*e is an array containing  $|V|$  ones. On the other hand, for the condensed vertices, this value will be equal to the sum of the *vsiz*e values condensed into that vertex. For the condensed digraph in Figure 5(c), *vsiz*e =  $\{3, 2, 1\}$  when its vertices are ordered from left to right. To be precise, for a recursive call with  $G = (V, E)$ , the total number of simple vertices is  $\sum_{v \in V} vsiz(e)(v)$  and this number is larger than  $mbs$  for all recursive calls because of the size check in line 17 of Algorithm 2.

For each call, HDPRE checks if the problem size  $|E| - i$  is equal to one. If this is the case, it finds the strong components  $SC_\ell = (V_\ell, E_\ell)$  of  $G$ . If a strong component  $SC_\ell$  has  $\sum_{v \in V_\ell} vsiz(e)(v) > mbs$  vertices then HDPRE considers each vertex in  $V_\ell$  as a different strong component. Otherwise, i.e., if the size of a strong component is less than or equal to  $mbs$ , that component is considered as a whole. Following this logic, HDPRE constructs the *scomp* array and returns. If the problem size,  $|E| - i$  is greater than 1, as was done for HD, HDPRE constructs  $G_j$  for  $j = \lceil (i + |E|)/2 \rceil$  and, if it is strongly connected, the search for the combining edge among the first  $j$  edges starts with the call  $\text{HDPRE}(G_j, \sigma, i, vsiz(e))$ . If not, for every strong component  $SC_\ell = (V_\ell, E_\ell)$  of  $G_j$  such that  $\sum_{v \in V_\ell} vsiz(e)(v) > mbs$ , it makes a recursive call  $\text{HDPRE}(SC_\ell, \sigma_\ell, i_\ell, vsiz(e)_\ell)$  and updates the strong component information for the vertices in  $V_\ell$ . This update operation can be considered as further dividing the strong component  $SC_\ell$  hierarchically until all of the strong components obtained during this process contain at most  $mbs$  vertices.

Similarly to HD, at line 33, HDPRE makes one more recursive call for the condensed graph  $G^\mathcal{V}$  where the definition of the vertex partition  $\mathcal{V}$  (in line 27) is modified as in Fig. 3.7. In HD, each vertex in the condensed graph corresponds to a strong component of  $G_j$  which defines a partition  $\mathcal{V}_1$ . In HDPRE, these components are further divided until all of them have a size no larger than  $mbs$ . A second partition,  $\mathcal{V}_2$ , is obtained from these smaller strong components and  $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2\}$  is defined. After obtaining the condensed graph  $G^\mathcal{V}$ , HDPRE checks if  $G^\mathcal{V}$  is acyclic. Note that if  $i^\mathcal{V} = |E^\mathcal{V}|$ , no strong component with two or more vertices exists in  $G^\mathcal{V}$  and hence it is acyclic.

If  $i^{\mathcal{V}} \neq |E^{\mathcal{V}_1}|$ , after obtaining  $scomp^{\mathcal{V}}$ , HDPRE updates  $scomp$  if a larger strong component is obtained.

For the matrix given in Figure 1(a), HDPRE generates the blocks for the cases  $mbs = 2$  and  $mbs = 3$  as shown in Figure 8(a) and Figure 8(b), respectively. For  $mbs = 2$ , the condensed graph has 5 vertices and no edges hence no combination will occur. For  $mbs = 3$ , as shown in Figure 8(b), the condensed graph has 3 vertices where 2 of them will combine with the 12th edge in  $\sigma_0$ .

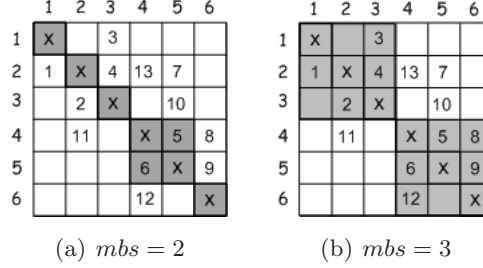


Figure 3.8: Initial block structure of the preconditioner after HDPRE algorithm. Two cases,  $mbs = 2$  and  $mbs = 3$ , are investigated for the matrix in Figure 1(a).

### 3.1.3 Combining the blocks

After HDPRE obtains a block diagonal partition, SCPRE performs a loop on the nonzeros which are not contained in a block on the diagonal to see if it is possible to put more into the block diagonal by combining original blocks. To do this, SCPRE first constructs a condensed simple graph  $H$  where the vertices of  $H$  correspond to the diagonal blocks and inter-block edges of  $G$  in both directions are combined as a single edge with a weight that is the sum of the weights of the combined edges.

After  $H$  is obtained, its edges are visited in an order corresponding to a permutation  $\sigma_H$ . This permutation is consistent with the original permutation  $\sigma_0$ . That is, if the edges of the original digraph are sorted in descending order with respect to the edge weights,  $\sigma_H$  permutes the edges of  $H$  with respect to descending edge weights. On the other hand, if the initial permutation is based on the RCM ordering we compute the RCM ordering of  $H$ , relabel the vertices of  $H$  accordingly, and order the edges with respect to this RCM ordering. Let  $usize(u)$  be the number of rows/columns in a block corresponding to the vertex  $u$ .

Assume that SCPRE constructs  $\sigma_0$  by sorting the edges with respect to decreasing weights. For the matrix given in Figure 8(a), if  $w(2) + w(4) > w(1)$  then vertices 2 and 3 are combined or if  $w(2) + w(4) \leq w(1)$  then vertices 1 and 2 are combined. Since  $mbs = 2$  and there is no edge between vertices 1 and 6, no further combinations are performed.

## 3.2 SCPRE: Extending to a BTF preconditioner

If the desired structure of  $\mathbf{M}$  is block diagonal SCPRE stops. Otherwise, while preserving the blocks, it tries to extend the block diagonal preconditioner to a block upper-triangular one. Note that in this case the order of the blocks is important since it changes which nonzeros are in the upper-triangular part of  $\mathbf{M}$ . By permuting the blocks, SCPRE tries to put entries that are larger in magnitude into the block upper-triangular part. Our preliminary experiments confirmed that

having larger and more nonzeros in a **SCPRE** preconditioner increases its effectiveness. Since the nonzeros in the diagonal blocks stay same while extending a block diagonal preconditioner to a block triangular one, we focus on improving the nonzeros in the block upper-triangular part.

Let  $G = (V, E)$  be the digraph associated with the matrix and  $k$  be the number of diagonal blocks. Let  $\mathcal{V}_1 = \{V_1, V_2, \dots, V_k\}$  be a partition of  $V$  such that the vertices in  $V_i$  correspond to the rows/columns of  $i$ th block. Let  $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_1\}$  and  $G^\mathcal{V} = \text{condense}(G, \mathcal{V})$  be the condensed multigraph. Note that if  $G^\mathcal{V}$  is acyclic, a topological sort in  $G^\mathcal{V}$  gives a symmetric block permutation such that all of the nonzeros in the matrix will be in the upper-triangular part of the permuted matrix. However, this only happens for a reducible matrix with blocks having no more than  $mbs$  rows/columns.

The problem of finding a good block permutation, which maximizes the number of nonzeros in the upper part of  $\mathbf{M}$ , can be reduced to the problem of finding the smallest edge set  $E'$  such that  $\overline{G}^\mathcal{V} = (V^\mathcal{V}_1, E^\mathcal{V}_1 \setminus E')$  is acyclic. For the weighted version of the problem, i.e., to maximize the total magnitude in the upper part, we need to find an edge set  $E'$  where  $\overline{G}^\mathcal{V}$  is acyclic and the sum  $\sum_{uv \in E'} |w(uv)|$  is minimum. In the literature, the first problem is called the *directed feedback arc set* problem and the second one is called the *directed weighted feedback arc set* problem. Both problems are NP-complete (Garey and Johnson 1979, Gavril 1977).

Our simple heuristic proposed for this problem is a greedy algorithm: we first choose the block row with the largest entries in the off-diagonal blocks and remove the corresponding rows/columns in this block. We then do the same thing with the remaining block matrix to obtain the second block row and continue in this way until a single block remains. More formally, we let  $G^\mathcal{V}$  be the condensed graph described above. For each vertex  $u \in V^\mathcal{V}$ , let  $tw\text{eight}(u) = \sum_{uv \in E^\mathcal{V}} w(uv)$ . The main body of the algorithm is a **for** loop where at the  $i$ th iteration, it chooses the vertex  $u$  with maximum  $tw\text{eight}$  and assigns it as the  $i$ th vertex in the permutation. It then removes  $u$  from  $V^\mathcal{V}$ , its edges from  $E^\mathcal{V}$ , and continues with the next iteration. After permuting the matrix with **SCPRE**, we expect that nonzeros with larger magnitudes are mostly placed in the diagonal blocks and some in the upper-triangular part. We show in Figures 9(a) and 9(b) the matrix *ckt11752\_tr\_0* after scaling using **MC64** and after the reordering from **SCPRE**, respectively. In the reordered matrix of Figure 9(b) it is clear that the larger entries are in the diagonal blocks.

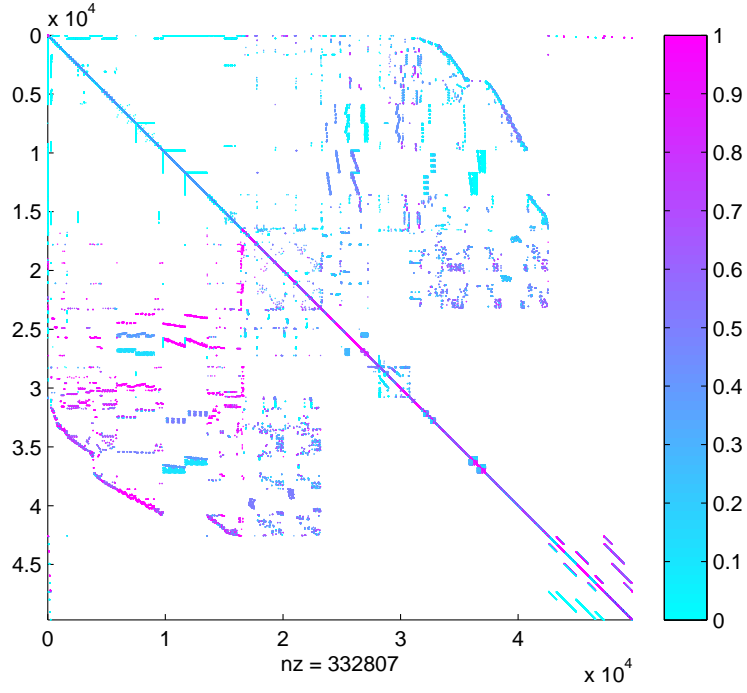
## 4 Using SCPRE with an iterative solver

The iterative solver we use in our experiments is the right-preconditioned **GMRES** (Saad and Schultz 1986) with restarts. A template for this can be found in Barrett, Berry, Chan, Demmel, Donato, Dongarra, Eijkhout, Pozo, Romine and van der Vorst (1994). Let  $\mathbf{A} = \mathbf{D} + \mathbf{U} + \mathbf{L}$  be the scaled and permuted matrix such that  $\mathbf{D}$ ,  $\mathbf{U}$ , and  $\mathbf{L}$  are the block diagonal, upper, and lower parts, respectively.

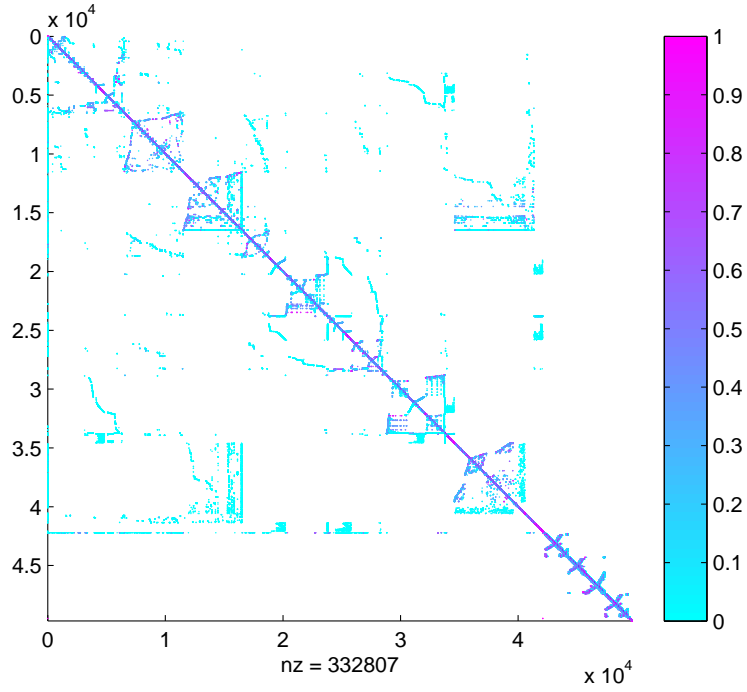
If the desired structure is block diagonal, which is suitable for exploitation of parallelism,  $\mathbf{M} = \mathbf{D}$  is the preconditioner. If this is not the case,  $\mathbf{M} = \mathbf{D} + \mathbf{U}$  is the preconditioner for  $\mathbf{A}$ . For the latter case, the computation  $\mathbf{A}\mathbf{M}^{-1}\mathbf{x}$  becomes

$$\mathbf{A}\mathbf{M}^{-1}\mathbf{x} = (\mathbf{D} + \mathbf{U} + \mathbf{L})(\mathbf{D} + \mathbf{U})^{-1}\mathbf{x} = \mathbf{x} + \mathbf{L}((\mathbf{D} + \mathbf{U})^{-1}\mathbf{x}).$$

Note that **SCPRE** tries to maximize the total magnitude in  $\mathbf{D}$  and  $\mathbf{U}$ . As a consequence and as in experiments not included here show,  $\mathbf{L}$  usually contains many fewer nonzeros than  $\mathbf{A}$ . Hence



(a) After MC64, before  $\text{SCPREDec}$



(b) After  $\text{SCPREDec}$

Figure 3.9: The matrix  $\text{ckt11752\_tr\_0}$  after scaling (a) and after  $\text{SCPREDec}$  (b), respectively. The nonzeros are coloured w.r.t. their magnitudes.  $\text{mbs}$  is set to 5000.

computing the vector  $\mathbf{z} = \mathbf{L}\mathbf{y}$  usually takes very little time and the main operation is to compute  $\mathbf{y} = (\mathbf{D} + \mathbf{U})^{-1}\mathbf{x} = \mathbf{M}^{-1}\mathbf{x}$ . In our implementation, in addition to  $\mathbf{A}$ , we store the  $\mathbf{LU}$  factors of the diagonal blocks, i.e., the factors  $\mathbf{L}_i$  and  $\mathbf{U}_i$  such that  $\mathbf{D}_i = \mathbf{L}_i\mathbf{U}_i$  where  $\mathbf{D}_i$  is the  $i$ th diagonal block. We reduce the memory requirements for these factors by ordering the blocks using the approximate minimum degree (AMD) heuristic (Amestoy, Davis and Duff 1994, Davis 2006) before using the MATLAB sparse factorization. We then solve the upper block triangular system  $\mathbf{M}\mathbf{y} = \mathbf{x}$  using these factors, starting with the last block, so that the off-diagonal part  $\mathbf{U}$  is only used to multiply vectors.

#### 4.1 Robustness

The use of  $I$ -matrix scaling via MC64 helps to reduce the possibility of a singular preconditioner  $\mathbf{M}$  obtained by SCPRE because all the submatrices on the diagonal will also be  $I$ -matrices. But, although it is very rare, these  $I$ -matrices can be singular and we still find cases in which some of the blocks on the diagonal of  $\mathbf{M}$  are singular.

When using the MATLAB factorization, we guard against this potential problem by using the simple and cheap stability check proposed and used by XPABLO (Fritzsche 2010, Fritzsche et al. 2007). That is, if  $n_i$  is the dimension of  $\mathbf{D}_i$ , after computing  $\mathbf{L}_i$  and  $\mathbf{U}_i$ , we check whether

$$\left| 1 - \frac{\|\mathbf{U}_i^{-1}\mathbf{L}_i^{-1}\mathbf{x}\|}{\|\mathbf{e}\|} \right| < \sqrt{\epsilon_M}, \quad (4.2)$$

where  $\mathbf{e} = (1, \dots, 1)^T$  is an  $n_i \times 1$  column vector,  $\mathbf{x} = \mathbf{D}_i\mathbf{e}$ , and  $\epsilon_M$  is machine epsilon. If a block does not satisfy (4.2), XPABLO replaces  $\mathbf{D}_i$  either by  $\mathbf{U}_i$  or  $\mathbf{L}_i$  according to whether it is solving a block upper- or lower-triangular system, respectively. For SCPRE, we use the same test as XPABLO but always use the factor having the largest Frobenius norm to replace  $\mathbf{D}_i$ , where the Frobenius norm of an  $n \times n$  matrix  $\mathbf{B}$  is given by

$$\|\mathbf{B}\|_F = \sqrt{\sum_{1 \leq i, j \leq n} |\mathbf{B}_{ij}|^2}.$$

## 5 Experiments

All of the experiments are conducted on an Intel 2.4Ghz Quad Core computer, equipped with 24GB RAM with a Fedora Linux operating system. For the experiments, we use matrices from the University of Florida Sparse Matrix Collection (Davis and Hu 2011). The matrices we use come from circuit simulation problems (CSP), semiconductor device problems (SDP), electromagnetics problems (EMP), and optimization problems (OPT). We run three sets of comparisons using these matrices. The first set contains 45 matrices with  $m \leq 2 \times 10^6$  nonzeros. For this set, we use  $mbs = 2000$  in the experiments. The second set contains 13 relatively large matrices with  $m \geq 2 \times 10^6$  nonzeros. For this set, we use  $mbs = 5000$  since they are larger. The third set contains 12 average-size optimization matrices with  $10^6 \leq m \leq 2.5 \times 10^6$  nonzeros. In constructing the sets, we do not use matrices whose largest blocks in their BTF form have less than  $mbs$  rows/columns. We also exclude, from the tables, any matrices on which none of our preconditioned iterative solvers converged. The lists of the remaining 37 matrices in the first set, 12 matrices in the second set, and 6 matrices in the third set are given in Table 5.1.



Table 5.1: Properties of the matrices used for the experiments.  $n$  is the dimension of the matrix,  $m$  is the number of nonzeros, and  $n_1$  and  $n_2$  are the size of the largest and second largest blocks in the BTF form. Note that  $n_2 = 0$  means that the matrix is irreducible, i.e.,  $n_1 = n$ . The column Type shows the application from which the matrix arises. The sets are sorted first according to the type of the problem and then their  $n_1$  values.

	Matrix	Group	$n$	$m$	$n_1$	$n_2$	Type
SET 1	<i>Hamrle2</i>	Hamrle	5952	22162	5952	0	CSP
	<i>rajat03</i>	Rajat	7602	32653	7500	1	
	<i>circuit_3</i>	Bomhof	12127	48137	7607	1	
	<i>coupled</i>	IBM_Austin	11341	97193	11293	1	
	<i>memplus</i>	Hamm	17758	99147	17736	1	
	<i>rajat22</i>	Rajat	39899	195429	26316	7672	
	<i>onetone2</i>	ATandT	36057	222596	32211	2	
	<i>onetone1</i>	ATandT	36057	335552	32211	2	
	<i>rajat15</i>	Rajat	37261	443573	37243	1	
	<i>ckt11752_tr_0</i>	IBM_EDA	49702	332807	49371	44	
	<i>circuit_4</i>	Bomhof	80209	307604	52005	7	
	<i>bcircuit</i>	Hamm	68902	375558	68902	0	
	<i>rajat18</i>	Rajat	94294	479151	84507	52	
	<i>hcircuit</i>	Hamm	105676	513072	92144	4927	
	<i>ASIC_100ks</i>	Sandia	99190	578890	98843	2	
	<i>ASIC_100k</i>	Sandia	99340	940621	98843	2	
	<i>ASIC_680ks</i>	Sandia	682712	1693767	98843	2	
	<i>rajat23</i>	Rajat	110355	555441	103024	216	
	<i>twotone</i>	ATandT	120750	1206265	105740	6	
	<i>trans5</i>	IBM_EDA	116835	749800	116817	1	
	<i>dc2</i>	IBM_EDA	116835	766396	116817	1	
	<i>G2_circuit</i>	AMD	150102	726674	150102	0	
	<i>scircuit</i>	Hamm	170998	958936	170493	216	
	<i>transient</i>	Freescal	178866	961368	178823	11	
	<i>Raj1</i>	Rajat	263743	1300261	263571	5	
	<i>ASIC_320ks</i>	Sandia	321671	1316085	320926	6	
	<i>ASIC_320k</i>	Sandia	321821	1931828	320926	6	
	<i>utm5940</i>	TOKAMAK	5940	83842	5794	1	EMP
	<i>dw4096</i>	Bai	8192	41746	8192	0	
	<i>Zhao1</i>	Zhao	33861	166453	33861	0	
	<i>igbt3</i>	Schenk_ISEI	10938	130500	10938	0	SDP
	<i>wang3</i>	Wang	26064	177168	26064	0	
	<i>wang4</i>	Wang	26068	177196	26068	0	
	<i>ecl32</i>	Sanghavi	51993	380415	42341	1	
	<i>ibm_matrix_2</i>	Schenk_IBMSDS	51448	537038	44822	1	
	<i>matrix-new_3</i>	Schenk_IBMSDS	125329	893984	78672	1	
	<i>matrix_9</i>	Schenk_IBMSDS	103430	1205518	99372	1	
SET 2	<i>ASIC_680k</i>	Sandia	682862	2638997	98843	2	CSP
	<i>G3_circuit</i>	AMD	1585478	7660826	181343	0	
	<i>rajat29</i>	Rajat	643994	3760246	629328	71	
	<i>rajat30</i>	Rajat	643994	6175244	632151	0	
	<i>Hamrle3</i>	Hamrle	1447360	5514242	1447360	0	
	<i>memchip</i>	Freescal	2707524	13343948	2706851	0	EMP
	<i>offshore</i>	Um	259789	4242673	259789	0	
	<i>tmt_sym</i>	CEMW	726713	5080961	726713	0	
	<i>t2em</i>	CEMW	921632	4590832	917300	1	
	<i>tmt_unsym</i>	CEMW	917825	4584801	917825	0	
	<i>para-4</i>	Schenk_ISEI	153226	2930882	153226	0	SDP
	<i>ohne2</i>	Schenk_ISEI	181343	6869939	181343	0	
SET 3	<i>ex_data1</i>	GHS_indef	6001	2269500	6001	0	OPT
	<i>boyd1</i>	GHS_indef	93279	1211231	93279	0	
	<i>majorbasis</i>	QLi	160000	1750416	160000	0	
	<i>c-73b</i>	Schenk_IBMNA	169422	1279274	169422	0	
	<i>c-big</i>	Schenk_IBMNA	345241	2340859	345089	2	
	<i>boyd2</i>	GHS_indef	466316	1500397	466316	0	



In our experiments, we restarted **GMRES** (Saad and Schultz 1986) after every 50 iterations. The desired error tolerance for **GMRES**(50) is set to  $\epsilon = 10^{-8}$  and the stopping criterion we use for **GMRES** is

$$\frac{\|\mathbf{A}\mathbf{M}^{-1}\bar{\mathbf{z}} - \mathbf{b}\|}{\|\mathbf{b}\|} < \epsilon$$

where  $\bar{\mathbf{z}} = \mathbf{M}\bar{\mathbf{x}}$ , with  $\bar{\mathbf{z}}$  the computed solution of the preconditioned system and  $\bar{\mathbf{x}}$  the computed solution of the original system. After obtaining the solution  $\bar{\mathbf{x}}$  to the original system, we compute the relative error  $\|\mathbf{A}\bar{\mathbf{x}} - \mathbf{b}\|/\|\mathbf{b}\|$  to the unpreconditioned system. For all cases, this error is smaller than  $10^{-7}$  and indeed, for most of the cases it is also smaller than  $\epsilon$ .

The maximum number of outer iterations is set to 20, hence the maximum number of inner iterations is 1000. In the tables, we give the inner iteration counts when this criterion is satisfied. Otherwise, if the criterion is not satisfied, we put “–” in the table to denote that **GMRES** did not converge. Also, we put the lowest iteration count for each matrix in bold font.

To compare the efficiency of the preconditioner, we used a generic preconditioner, **ILUT** (Saad 1994, Saad 2003), from MATLAB 7.11 with two drop tolerances,  $dtol = 10^{-3}$  and  $10^{-4}$ . In addition to **ILUT**, we also compared our results with those of **XPABLO** (Fritzsche 2010, Fritzsche et al. 2007). For all of the preconditioners, we use **MC64** and obtain a maximum product transversal by scaling and permuting the matrix as a preprocessing step.

In the MATLAB implementation of **ILUT**, for the  $j$ th column of the incomplete **L** and **U**, entries smaller in magnitude than  $dtol \times \|\mathbf{A}_{*j}\|$  are deleted from the factor where  $\|\mathbf{A}_{*j}\|$  is the norm of the  $j$ th column of **A**. However, the diagonal entries of **U** are always kept to avoid a singular factor. For the **ILUT** based preconditioners, we use **AMD** before computing the incomplete factorization of the matrix. For **XPABLO** preconditioners, we use the **J** variant for the block Jacobi iterations, and **LX** and **UX** variants for the forward and backward block Gauss-Seidel iterations, respectively, with the parameters given in Fritzsche et al. (2007). For the maximum block size of **XPABLO**, we used the same *mbs* as for **SCPRE**. We note that the authors of **XPABLO** recommend a value for *mbs* of 1000 (Fritzsche 2010) but, in our experiments, we found the value 2000 to work better and found that it was necessary for our larger problems to avoid failure in **XPABLO**.

**SCPRE** will automatically find the BTF for a reducible matrix. To be fair to the other algorithms that do not detect this form, we use this reducibility information also for the **ILUT** and **XPABLO** preconditioners. That is, when using **ILUT** (**XPABLO**) for reducible matrices, we first compute the BTF form and apply **ILUT** (**XPABLO**) only to the blocks on the diagonal. For smaller blocks, we compute the complete factors. We then use these complete and incomplete factors together while computing a matrix vector product using  $\mathbf{M}^{-1}$ . Our experiments show that this approach is almost always better than using **ILUT** (**XPABLO**) in a straightforward manner in terms of iteration count. We also tried this approach while using the **J** variant of the **XPABLO** preconditioner. Surprisingly, even for the block Jacobi case, this approach helps to reduce the iteration counts slightly for most of the reducible matrices. We call this variant **J-red** in the tables below. Note that for the block Gauss-Seidel case, when we apply **XPABLO** (or **ILUT**) only to the large blocks in the BTF form of a reducible matrix, we keep all of the nonzeros in the preconditioner from the off-diagonal blocks. However, for block Jacobi iterations, we automatically drop them from the preconditioning matrix **M** since its desired structure is block diagonal, not block triangular.

In addition to the number of iterations required for convergence, we compare the performance of the preconditioners according to the relative memory requirement with respect to the number

of nonzeros in  $\mathbf{A}$ . Let  $nz(\mathbf{B})$  be the number of nonzeros in a matrix  $\mathbf{B}$ . For ILUT, the relative memory requirement is equal to

$$mem_{\text{ILUT}} = \frac{nz(\mathbf{L}) + nz(\mathbf{U})}{nz(\mathbf{A})},$$

where  $\mathbf{L}$  and  $\mathbf{U}$  are the incomplete triangular factors of  $\mathbf{A}$ . On the other hand, the relative memory requirement for SCPRE and XPABLO is equal to

$$mem_{\text{SCPRE}} = mem_{\text{XPABLO}} = \frac{\sum_{i=1}^k (nz(\mathbf{L}_i) + nz(\mathbf{U}_i))}{nz(\mathbf{A})},$$

where  $k$  is the number of blocks in the block diagonal  $\mathbf{D}$  and  $\mathbf{L}_i$  and  $\mathbf{U}_i$  are the lower- and upper-triangular factors of the  $LU$  factorization of the  $i$ th block in  $\mathbf{D}$ . Note that the relative memory requirements of the preconditioners can give an idea for the cost of computing  $\mathbf{M}^{-1}\mathbf{x}$ . Assuming  $x$  is a dense vector, a preconditioned GMRES iteration will require approximately  $nz(\mathbf{A})(1+mem_{\mathbf{x}})$  operations for the preconditioner generated by the algorithm  $\mathbf{X}$ .

There are two parameters for the proposed algorithm: the first is the maximum block size,  $mbs$ , the second is the permutation for the nonzeros, denoted by  $\sigma_0$ . As expected, our experiments (not reported here) show that increasing  $mbs$  usually reduces the iteration counts and increases the relative memory requirements of the solver.

We conduct some experiments to show the effect of our choice of  $\sigma_0$  on the performance of our algorithm. Note that in HD the edges are sorted in increasing order with respect to their weights. In our implementation, we define the weight of an edge as the magnitude of the corresponding nonzero and sort the edges in decreasing order. We test our decision by comparing its effect with that of a random permutation. As Table 5.2 shows, our decision to sort the edges in decreasing order with respect to the edge weights makes the solver converge more quickly.

Table 5.2: Effect of the permutation  $\sigma_0$  on the number of iterations. Two options are compared: decreasing order with respect to the edge weights and a random order. Maximum block size for SCPRE is set to 2000 where the structure of  $\mathbf{M}$  is block upper-triangular. For each case, the ratio of the total magnitude in  $\mathbf{M}$  to the total magnitude in  $\mathbf{A}$ , the relative memory requirement, and the number of inner iterations for preconditioned GMRES are given.

Matrix	Decreasing			Random		
	$\frac{\sum  \mathbf{M}_{ij} }{\sum  \mathbf{A}_{ij} }$	$mem_{\text{SCPRE}}$	$iters$	$\frac{\sum  \mathbf{M}_{ij} }{\sum  \mathbf{A}_{ij} }$	$mem_{\text{SCPRE}}$	$iters$
<i>Hamrle2</i>	0.998	2.03	<b>16</b>	0.993	2.05	157
<i>rajat03</i>	0.999	1.07	<b>2</b>	0.997	1.02	5
<i>circuit_3</i>	0.996	1.45	<b>9</b>	0.987	1.23	445
<i>coupled</i>	0.998	1.57	<b>11</b>	0.992	1.58	34
<i>memplus</i>	0.999	1.03	<b>5</b>	0.998	1.03	7
<i>rajat22</i>	0.973	1.20	<b>21</b>	0.962	1.15	-

## 5.1 Experiments with block Gauss-Seidel iterations

Table 5.3 shows the performance of SCPRE and XPABLO for block Gauss-Seidel iterations and their comparison with ILUT. Note that both SCPRE(*dec*) and SCPRE(RCM) are robust; that is, the solvers converge for most of the matrices. Although there are a few matrices for which the SCPRE(RCM) preconditioned solver converges more quickly than that preconditioned with SCPRE(*dec*) (such as *ASIC\_680k*) and, amongst all preconditioners, only SCPRE(RCM) converges for matrices *onetone1* and *onetone2*, SCPRE(*dec*) is almost always better and is our preferred preconditioner.

In general, all the preconditioners work well for the matrices in the first set. However, **SCPRE**(*dec*) is the most robust since the preconditioned solver fails to converge only for 3 out of 37 matrices whereas the next best result is 9 by **XPABLO** variants. Thus **SCPRE**(*dec*) is the best block preconditioner on this set of matrices. When comparing **SCPRE**(*dec*) to **ILUT**( $10^{-4}$ ) on this set we see that they are comparable in terms of the number of best performances but **ILUT**( $10^{-4}$ ) is less robust failing to converge for 10 matrices in this set and requiring more memory than **SCPRE**(*dec*).

For the second set, **ILUT**( $10^{-4}$ ) is the best preconditioner in terms of robustness and iteration count. For the matrices in this set, the **ILUT**( $10^{-4}$ ) preconditioned solver fails to converge in only 2 out of 12 matrices whereas **SCPRE**(*dec*) does not converge on 4. Although **ILUT**( $10^{-4}$ ) is better than **SCPRE**(*dec*) for 10 out of 12 matrices in the second set, its average relative memory usage is 9.39 which is almost 3 times as much as the relative memory requirement of **SCPRE**(*dec*). Note that, for the second set, even **ILUT**( $10^{-3}$ ) uses slightly more memory than **SCPRE**(*dec*). However, it fails to converge on 7 matrices. Hence, if memory is the bottleneck, **SCPRE**(*dec*) may be a suitable choice for preconditioning.

The performance of the **SCPRE**-based preconditioners depends on the application. For example, as Table 5.3 shows, **SCPRE**(*dec*) preconditioned **GMRES** fails to converge in 3 out of 7 matrices from electromagnetics applications. On the other hand, it fails to converge on only 4 of the remaining 42 matrices. Hence its performance is much better for circuit and device simulation applications. Note that, even though some of these matrices are reducible, they have a large reducible block with size much larger than *mbs*. That is, we still have a large subproblem to deal with. On the circuit simulation and semiconductor device matrices, **SCPRE** works better than **XPABLO** which is another block based preconditioner with a promising performance in practice for several matrix classes (Benzi et al. 1997, Choi and Szyld 1996, Dayar and Stewart 2000). Note that we used the BTF forms of the reducible matrices for both the **XPABLO** and **ILUT** preconditioners. Hence, reducibility alone is not a reason for the good performance of **SCPRE**-based preconditioners.

### 5.1.1 Memory usage

As Table 5.3 shows, the memory usage of **ILUT**( $10^{-4}$ ) is much more than that of **XPABLO** and **SCPRE**. Table 5.4 shows the results of additional experiments conducted to further compare the memory usage of **SCPRE** and **ILUT** preconditioners. There are 6 optimization matrices in the set. **SCPRE**-based preconditioned solvers converged for 5 of them. For **ILUT**-based solvers with drop tolerance  $10^{-3}$  and  $10^{-4}$ , the numbers of matrices for which the solver converged are 4 and 5, respectively. Hence, on this matrix set, **SCPRE** is as robust as **ILUT**. With respect to the number of iterations, **ILUT** is much better with 7–8 iterations on the average instead of 36 for **SCPRE**. The main reason for such a big difference is the matrix *c-73b* where the **SCPRE** preconditioned solver requires 127 inner iterations. On the other hand, the average relative memory usage of **ILUT** is 11–18 times more than that of **SCPRE**. This difference is due to the matrices *c-73b* and *c-big* where **ILUT**'s relative memory requirements are 28.47 and 61.89, respectively. Additionally, for the matrix *boyd2*, **ILUT** could not generate a preconditioner since the maximum memory available in the system, 24GB, is exceeded. Given that only 28MB is used to store *boyd2*, the relative memory requirement of **ILUT** is excessive. This shows that although **SCPRE**-preconditioned solvers require more iterations than **ILUT**-preconditioned ones, **SCPRE** can still be a good replacement for some matrix classes if the matrices are big and memory is the main bottleneck.

Table 5.3: Number of inner iterations for GMRES using XPABLO, ILUT and SCPRE preconditioners and block Gauss-Seidel iterations. For SCPRE and XPABLO,  $mbs$  is set to 2000 and 5000 for the first and second sets, respectively. For SCPRE, we give the results using two permutations for  $\sigma_0$ , based on descending order and RCM. For XPABLO, we give the results for both the UX and LX variants. For ILUT, the drop tolerance is set to  $10^{-3}$  and  $10^{-4}$ . A ‘-’ sign indicates that the preconditioned solver did not converge. Average relative memory requirements are computed by taking the averages over the cases when the solvers converge.

	Matrix	XPABLO		SCPRE		ILUT	
		UX	LX	dec	RCM	$10^{-3}$	$10^{-4}$
$mbs = 2000$	<i>Hamrle2</i>	31	31	16	28	6	<b>4</b>
	<i>rajat03</i>	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>
	<i>circuit_3</i>	135	137	<b>9</b>	61	-	-
	<i>coupled</i>	12	12	11	13	6	<b>4</b>
	<i>memplus</i>	9	9	<b>5</b>	18	15	9
	<i>rajat22</i>	36	37	21	61	36	<b>16</b>
	<i>onetone2</i>	-	-	-	<b>248</b>	-	-
	<i>onetone1</i>	-	-	-	<b>297</b>	-	-
	<i>rajat15</i>	-	-	120	467	-	<b>33</b>
	<i>ckt11752_tr_0</i>	197	188	<b>19</b>	323	-	-
	<i>circuit_4</i>	100	81	<b>39</b>	346	-	-
	<i>bcircuit</i>	-	-	<b>40</b>	620	568	93
	<i>rajat18</i>	-	-	<b>11</b>	-	393	54
	<i>hcircuit</i>	8	9	9	21	9	<b>5</b>
	<i>ASIC_100ks</i>	9	10	9	10	<b>4</b>	<b>4</b>
	<i>ASIC_100k</i>	9	9	10	10	<b>4</b>	<b>4</b>
	<i>ASIC_680ks</i>	<b>3</b>	4	<b>3</b>	4	4	4
	<i>rajat23</i>	40	41	<b>16</b>	88	47	18
	<i>twotone</i>	-	-	<b>25</b>	128	-	48
	<i>trans5</i>	9	9	<b>5</b>	7	7	6
	<i>dc2</i>	13	12	12	11	10	<b>6</b>
	<i>G2_circuit</i>	-	-	444	834	124	<b>30</b>
	<i>scircuit</i>	741	764	<b>317</b>	977	-	-
	<i>transient</i>	-	-	<b>33</b>	-	-	-
	<i>Raj1</i>	775	789	636	-	269	<b>39</b>
	<i>ASIC_320ks</i>	4	4	<b>1</b>	4	2	2
	<i>ASIC_320k</i>	5	5	<b>2</b>	3	3	3
	<i>utm5940</i>	-	-	-	-	-	<b>29</b>
	<i>dw4096</i>	881	798	13	141	24	<b>10</b>
	<i>Zhao1</i>	7	7	4	9	4	<b>3</b>
	<i>igbt3</i>	29	29	20	17	94	<b>12</b>
	<i>wang3</i>	107	105	54	58	18	<b>9</b>
	<i>wang4</i>	39	38	21	36	11	<b>6</b>
	<i>ecl32</i>	99	99	30	32	32	<b>13</b>
	<i>ibm_matrix_2</i>	-	249	<b>10</b>	16	-	-
	<i>matrix-new_3</i>	85	86	<b>30</b>	41	-	-
	<i>matrix_9</i>	146	90	98	<b>88</b>	-	-
Avg. relative memory		2.95	3.04	3.36	3.19	2.12	4.02
$mbs = 5000$	<i>ASIC_680k</i>	<b>2</b>	<b>2</b>	27	<b>2</b>	3	3
	<i>G3_circuit</i>	-	-	357	422	212	<b>81</b>
	<i>rajat29</i>	-	-	<b>11</b>	-	-	-
	<i>rajat30</i>	12	12	14	15	7	<b>5</b>
	<i>Hamrle3</i>	-	-	-	-	-	<b>17</b>
	<i>memchip</i>	26	27	10	20	8	<b>5</b>
	<i>offshore</i>	330	327	488	451	-	<b>15</b>
	<i>tmt_sym</i>	-	-	-	-	-	<b>69</b>
	<i>t2em</i>	-	-	876	-	132	<b>38</b>
	<i>tmt_unsym</i>	-	-	-	-	-	<b>136</b>
	<i>para-4</i>	-	-	-	-	-	<b>433</b>
	<i>ohne2</i>	-	-	<b>196</b>	-	-	-
Avg. relative memory		3.58	3.58	3.23	2.51	3.36	9.39

Table 5.4: Number of inner iterations and relative memory usage of GMRES using SCPRE or ILUT preconditioners with block Gauss-Seidel iterations for optimization matrices. For SCPRE,  $mbs$  is set to 2000, and  $\sigma_0$  is obtained by using the descending order. For ILUT, the drop tolerance is set to  $10^{-3}$  and  $10^{-4}$ . The ‘\*’ sign indicates that the memory of our machine (24 GBytes) is not sufficient to obtain the preconditioner. The ‘-’ sign indicates that the preconditioner is obtained but the solver did not converge in fewer than 1000 iterations. Average number of iterations and relative memory requirements are computed by taking the averages over the cases when the solvers converge.

Matrix	SCPRE( <i>dec</i> )		ILUT- $10^{-3}$		ILUT- $10^{-4}$	
	<i>iters</i>	<i>mem</i>	<i>iters</i>	<i>mem</i>	<i>iters</i>	<i>mem</i>
<i>ex_data1</i>	<b>14</b>	0.60	-	-	23	<b>0.47</b>
<i>boyd1</i>	18	<b>0.19</b>	7	0.77	<b>5</b>	1.03
<i>majorbasis</i>	10	2.50	4	<b>1.20</b>	<b>3</b>	1.87
<i>c-73b</i>	127	<b>0.72</b>	10	14.19	<b>5</b>	28.47
<i>c-big</i>	-	-	6	<b>30.39</b>	<b>4</b>	61.89
<i>boyd2</i>	<b>13</b>	<b>1.17</b>	*	*	*	*
Avg.	36	1.04	7	11.64	8	18.75

## 5.2 Experiments with block Jacobi iterations

Table 5.5 shows the performance of SCPRE and XPABLO preconditioners for block Jacobi iterations. ILUT is not included here since it does not explicitly give a block diagonal structure. Similar to the experiments with block Gauss-Seidel iterations, performance of SCPRE(*dec*) is better than that of SCPRE(RCM) for the matrices in our sets. For XPABLO, applying the preconditioner only to the blocks in the BTF form, variant J-red, reduces the number of iterations on 11 matrices. Furthermore, for 5 of the matrices J-red converges whereas J does not. Note that there are 32 reducible matrices in the sets and J-red differs from J only for these matrices. Although J-red required more iterations for convergence for matrices *matrix\_new\_3* and *matrix\_9*, for the matrices in our experiments, J-red generally performs better than J.

As Table 5.5 shows, SCPRE(*dec*) preconditioned GMRES converges for 36 matrices whereas XPABLO’s J-red variant converges for only 24 matrices. The XPABLO based preconditioner has the least number of iterations in only 8 cases whereas the SCPRE variants are better on 35 matrices. The difference in the performance is not due to the relative memory usage of SCPRE variants. For the first set, SCPRE(*dec*) uses only 8% more memory than XPABLO(J-red) on average and, for the second set, its memory usage is much less.

In the right-hand side of Table 5.5, the execution times of the GMRES solver are given. As the table shows, for most of the cases, the best solver in terms of iteration count has also the best execution time. Note that there are some exceptions such as *matrix\_9* for which the solver preconditioned by XPABLO(J) requires 49 iterations fewer than when preconditioned by SCPRE but its execution time is slightly more. This is because, for this matrix,  $mem_{XPABLO(J)} = 8.43$  and  $mem_{SCPRE(dec)} = 3.69$ , and the cheaper cost of computing  $\mathbf{M}^{-1}\mathbf{x}$  more than compensates for the difference in iteration counts. For 39 matrices, a SCPRE variant has the best or very close to the best time. In summary, SCPRE(*dec*) performs better than XPABLO variants in our block Jacobi experiments.

Table 5.5: Number of inner iterations and solver times (in seconds) for GMRES using XPABLO and SCPRE preconditioners and block Jacobi iterations. The maximum block size  $mb$ s is set to 2000 and 5000 for the first and second sets, respectively. For SCPRE, we give the results using two permutations for  $\sigma_0$ , based on descending order and RCM. For XPABLO, we give the results for the J variant which is used with the parameters suggested in Fritzsche et al (2007). The J-red variant, described in the text, also uses the same parameters. A ‘-’ sign indicates that the preconditioned solver did not converge. Average relative memory requirements are computed by taking the averages over the cases when the solver converges.

Matrix	# iterations				solver time (in secs.)			
	XPABLO		SCPRE		XPABLO		SCPRE	
	J	J-red	dec	RCM	J	J-red	dec	RCM
<i>Hamrle2</i>	99	99	<b>31</b>	96	0.32	0.32	<b>0.08</b>	0.30
<i>rajat03</i>	7	<b>3</b>	4	4	0.03	<b>0.01</b>	<b>0.01</b>	<b>0.01</b>
<i>circuit_3</i>	680	327	<b>19</b>	179	3.72	1.81	<b>0.07</b>	0.93
<i>coupled</i>	43	22	<b>21</b>	25	0.22	<b>0.09</b>	<b>0.08</b>	0.11
<i>memplus</i>	17	17	<b>8</b>	33	0.08	0.08	<b>0.03</b>	0.19
<i>rajat22</i>	190	77	<b>42</b>	124	3.03	1.61	<b>0.63</b>	1.88
<i>onetone2</i>	-	-	-	<b>627</b>	-	-	-	<b>9.62</b>
<i>onetone1</i>	-	-	-	<b>622</b>	-	-	-	<b>14.95</b>
<i>rajat15</i>	-	-	<b>265</b>	-	-	-	<b>4.85</b>	-
<i>ckt11752_tr_0</i>	-	776	<b>36</b>	-	-	20.28	<b>0.83</b>	-
<i>circuit_4</i>	-	864	<b>112</b>	-	-	27.48	<b>3.33</b>	-
<i>bcircuit</i>	-	-	<b>107</b>	-	-	-	<b>3.06</b>	-
<i>rajat18</i>	-	-	<b>16</b>	-	-	-	<b>0.39</b>	-
<i>hcircuit</i>	16	<b>15</b>	16	40	<b>0.43</b>	<b>0.47</b>	<b>0.43</b>	1.55
<i>ASIC_100ks</i>	17	17	<b>16</b>	18	<b>0.46</b>	0.50	<b>0.44</b>	0.50
<i>ASIC_100k</i>	17	<b>16</b>	17	18	<b>0.48</b>	0.52	<b>0.49</b>	0.51
<i>ASIC_680ks</i>	-	<b>8</b>	-	<b>8</b>	-	<b>0.72</b>	-	<b>0.74</b>
<i>rajat23</i>	203	140	<b>32</b>	208	9.29	7.65	<b>1.17</b>	9.09
<i>twotone</i>	-	-	<b>49</b>	322	-	-	<b>2.70</b>	17.11
<i>trans5</i>	23	16	<b>9</b>	13	0.75	0.47	<b>0.24</b>	0.36
<i>dc2</i>	76	21	<b>20</b>	<b>20</b>	3.32	<b>0.67</b>	<b>0.64</b>	<b>0.64</b>
<i>G2_circuit</i>	-	-	<b>833</b>	-	-	-	<b>56.55</b>	-
<i>scircuit</i>	-	-	<b>682</b>	-	-	-	<b>49.25</b>	-
<i>transient</i>	-	-	<b>186</b>	-	-	-	<b>13.60</b>	-
<i>Raj1</i>	-	-	-	-	-	-	-	-
<i>ASIC_320ks</i>	5	6	<b>1</b>	7	0.43	0.70	<b>0.19</b>	0.54
<i>ASIC_320k</i>	11	9	<b>3</b>	10	0.91	0.85	<b>0.42</b>	0.61
<i>utm5940</i>	-	-	-	-	-	-	-	-
<i>dw4096</i>	-	-	<b>24</b>	-	-	-	<b>0.11</b>	-
<i>Zhao1</i>	12	12	<b>7</b>	16	0.12	0.12	<b>0.08</b>	0.19
<i>igbt3</i>	60	60	32	<b>26</b>	0.52	0.52	<b>0.21</b>	0.17
<i>wang3</i>	263	263	140	<b>138</b>	3.26	3.26	1.96	<b>1.78</b>
<i>wang4</i>	91	91	<b>39</b>	79	1.24	1.24	<b>0.54</b>	1.02
<i>ecl32</i>	-	-	<b>79</b>	90	-	-	<b>2.50</b>	3.08
<i>ibm_matrix_2</i>	-	344	<b>22</b>	30	-	12.29	<b>0.65</b>	1.09
<i>matrix-new_3</i>	184	248	<b>71</b>	95	13.21	20.20	<b>4.64</b>	6.79
<i>matrix_9</i>	<b>208</b>	240	257	346	<b>14.85</b>	18.80	<b>14.41</b>	21.83
Avg. relative memory	2.67	3.10	3.35	3.32				
<i>ASIC_680k</i>	-	<b>3</b>	-	<b>3</b>	-	<b>0.54</b>	-	<b>0.54</b>
<i>G3_circuit</i>	-	-	<b>674</b>	-	-	-	<b>516.38</b>	-
<i>rajat29</i>	-	-	<b>18</b>	-	-	-	<b>3.03</b>	-
<i>rajat30</i>	43	<b>22</b>	24	27	11.84	<b>4.59</b>	5.12	6.12
<i>Hamrle3</i>	-	-	-	-	-	-	-	-
<i>memchip</i>	41	50	<b>17</b>	38	53.39	74.55	<b>14.60</b>	38.82
<i>offshore</i>	<b>883</b>	<b>883</b>	-	-	<b>189.98</b>	<b>189.98</b>	-	-
<i>tmt_sym</i>	-	-	-	-	-	-	-	-
<i>t2em</i>	-	-	-	-	-	-	-	-
<i>tmt_unsym</i>	-	-	-	-	-	-	-	-
<i>para-4</i>	-	-	-	-	-	-	-	-
<i>ohne2</i>	-	-	-	-	-	-	-	-
Avg. relative memory	3.58	2.84	1.81	0.92				



### 5.3 Cost of generating the preconditioner

It has been the aim of this paper to establish the viability of using hierarchical decompositions to obtain a block preconditioning matrix that greatly reduces the number of iterations of Krylov solvers without requiring too much additional memory.

However, the cost of obtaining the preconditioning matrix is also important, especially if it is being generated for the solution of a single system. The analysis presented in Section 3.1.1 shows that the complexity of the HD algorithm is  $\mathcal{O}(m \log m)$  which means that it scales well as problem sizes increase. However, we note that the complexity of XPABLO is  $\mathcal{O}(m + n)$  which is thus linear in the order and number of entries in the matrix and could be expected to have smaller generation execution times than SCPRE.

A straight comparison of generation times is not meaningful as our implementation is fully in MATLAB without any low-level optimization whereas, for XPABLO, we used the available implementation in C, for which the compiler directly optimizes the code for the machine. It is the intention in future work to develop and optimize the implementation, but it is certainly outside the scope of this present work.

However, there is no doubt that although our algorithm has good complexity bounds it is quite complicated so we did time the generation of the SCPRE preconditioner on some of our test matrices. For example, for the CSP matrices of Table 5.1, we found that SCPRE took between 2.5 and 9.5 seconds whereas XPABLO required between 0.25 and 1.40 seconds. Thus, although our algorithm takes much longer, and would still be slower with an efficient implementation (which we estimate would be about 5 times faster), the times are not unreasonable and indicate that our approach is feasible even for one-off solutions. Indeed, if we look at the total cost, we are still faster than XPABLO on several problems in the one-off case, and of course the greater robustness of our more costly preconditioner compensates for this extra cost.

Table 5.6: Preconditioner generation times for 10 CSP matrices from the first matrix set in seconds. For SCPRE and XPABLO,  $mbs$  is set to 2000. For XPABLO, the UX variant is used, and for SCPRE,  $\sigma_0$  is obtained by using the descending order. For ILUT, the drop tolerance is set to  $10^{-3}$ . Results are the averages of 5 executions.

Matrix	XPABLO	SCPRE( <i>dec</i> )	ILUT- $10^{-3}$
<i>rajat15</i>	0.23	5.92	0.71
<i>ckt11752_tr_0</i>	0.45	3.97	0.23
<i>circuit_4</i>	0.15	8.13	0.11
<i>bcircuit</i>	0.21	3.95	0.12
<i>rajat18</i>	0.23	7.31	0.08
<i>hcircuit</i>	0.28	5.03	0.09
<i>ASIC_100ks</i>	0.35	9.65	0.24
<i>ASIC_100k</i>	0.43	9.50	0.19
<i>ASIC_680ks</i>	1.40	9.96	0.28
<i>rajat23</i>	0.29	7.64	0.08
<i>twotone</i>	1.02	9.09	12.41

## 6 Conclusions and future work

Given a linear system  $\mathbf{Ax} = \mathbf{b}$ , we have proposed a method to construct generic block diagonal and block triangular preconditioners. The proposed approach is based on Tarjan's algorithm HD for hierarchical decomposition of a digraph into its strong subgraphs. Although our preconditioner SCPRE, is outperformed by ILUT for electromagnetics matrices, we obtain promising results for

many device and circuit simulation matrices and we suggest using it with these types of problems. In future research, the structure of graphs for different classes of matrices can be analysed to try to understand the reason for the difference in performance.

There are two main parameters for the algorithm: the permutation  $\sigma_0$  of the edges and the maximum block size  $mbs$ . For  $\sigma_0$ , we used two approaches: the first sorts the edges in order of decreasing weights. With this approach, we wanted to include nonzeros with large magnitudes in our preconditioner. The second approach uses the well known reverse Cuthill-McKee ordering. We tested this approach since a sparsity structure with a small bandwidth may be useful for putting more nonzeros into the preconditioner. The permutation decisions are validated by the experiments which also show that the first approach is usually better than the second. In future work, other ways to generate  $\sigma_0$  can be investigated.

The second parameter,  $mbs$ , affects the memory requirement of the matrix significantly, and hence the number of iterations required for convergence. The experiments show that for the preconditioners ILUT, SCPRE and XPABLO, the memory requirement and the number of iterations are inversely correlated. For the proposed preconditioner SCPRE,  $mbs$  needs to be set by the user without knowing how much memory will be required by the solver. In future work, we will look for a self-tuning mechanism which enables SCPRE to determine  $mbs$  automatically given the memory available to store the preconditioner. A straightforward tuning mechanism, which combines the blocks only when sufficient memory for the factors is available, can be easily implemented and integrated into SCPRE. However, this simple idea still needs to be enhanced to optimize the execution time of SCPRE and further reduce the number of iterations required for convergence.

## Acknowledgments

The authors thank Dr. Philip Knight of the University of Strathclyde for bringing the elegant hierarchical decomposition algorithm of Tarjan to our attention. They also thank Dr. Mikko Byckling of CERFACS for valuable discussions and his GMRES implementation. Finally, we would like to thank the referees for their comments that have significantly improved our manuscript.

## References

- Amestoy, P. R., Davis, T. A. and Duff, I. S. (1994), An approximate minimum degree ordering algorithm, Technical Report TR-94-039, Computer and Information Science Department, University of Florida.
- Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. and van der Vorst, H., eds (1994), *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, SIAM, Philadelphia, PA.
- Benzi, M., Choi, H. and Szyld, D. (1997), Threshold ordering for preconditioning nonsymmetric problems, in G. Golub, S. Lui, F. Luk and R. Plemmons, eds, ‘Proceedings of the Workshop on Scientific Computing ’97 – Hong Kong’, Lecture Notes in Computer Science, Springer-Verlag, pp. 159–165.
- Benzi, M., Haws, J. C. and Tuma, M. (2000), ‘Preconditioning highly indefinite and nonsymmetric matrices’, *SIAM J. Scientific Computing* **22**(4), 1333–1353.



- Choi, H. and Szyld, D. B. (1996), Application of threshold partitioning of sparse matrices to Markov chains, in ‘Proceedings of the IEEE International Computer Performance and Dependability Symposium, IPDS’96, Urbana-Champaign, Illinois, September 4–6, 1996’, IEEE Computer Society Press, Los Alamitos, California, pp. 158–165. Was technical report 96-21 from Department of Mathematics, Temple University, Philadelphia.
- Cuthill, E. and McKee, J. (1969), Reducing the bandwidth of sparse symmetric matrices, in ‘Proceedings 24th National Conference of the Association for Computing Machinery, Brandon Press, New Jersey’, Brandon Press, New Jersey, pp. 157–172.
- Davis, T. A. (2006), *Direct Methods for Sparse Linear Systems*, SIAM, Philadelphia.
- Davis, T. A. and Hu, Y. (2011), ‘The University of Florida sparse matrix collection’, *ACM Trans. Math. Softw.* **38**(1), 1–25. <http://www.cise.ufl.edu/research/sparse/matrices/>.
- Davis, T. A. and Natarajan, E. P. (2010), ‘Algorithm 907: KLU, A direct sparse solver for circuit simulation problems’, *ACM Trans. Math. Softw.* **37**(3), 17 pages.
- Dayar, T. and Stewart, W. J. (2000), ‘Comparison of partitioning techniques for two-level iterative solvers on large, sparse markov chains’, *SIAM J. Scientific Computing* **21**, 1691–1705.
- Duff, I. S. and Koster, J. (2001), ‘On algorithms for permuting large entries to the diagonal of a sparse matrix’, *SIAM J. Matrix Analysis and Applications* **22**(4), 973–996.
- Duff, I. S. and Reid, J. K. (1978a), ‘Algorithm 529: Permutations to block triangular form [F1]’, *ACM Trans. Math. Softw.* **4**(2), 189–192.
- Duff, I. S. and Reid, J. K. (1978b), ‘An implementation of Tarjan’s algorithm for the block triangularization of a matrix’, *ACM Trans. Math. Softw.* **4**(2), 137–147.
- Fritzsche, D. (2010), Overlapping and Nonoverlapping Orderings for Preconditioning, PhD thesis, Temple University, Philadelphia, PA.
- Fritzsche, D., Frommer, A. and Szyld, D. B. (2007), ‘Extensions of certain graph-based algorithms for preconditioning’, *SIAM J. Scientific Computing* **29**, 2144–2161.
- Garey, M. R. and Johnson, D. S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman.
- Gavril, F. (1977), Some NP-complete problems on graphs, in ‘Proc. 11th Conference on Information Sciences and Systems’, pp. 91–95.
- HSL (2011), ‘HSL 2011: A collection of Fortran codes for large scale scientific computation’. <http://www.hsl.rl.ac.uk>.
- Olschowka, M. and Neumaier, A. (1996), ‘A new pivoting strategy for Gaussian elimination’, *Linear Algebra and its Applications* **240**, 131–151.
- Saad, Y. (1994), ‘ILUT: a dual threshold incomplete LU factorization’, *Numerical Linear Algebra with Applications* **1**(4), 387–402.

- Saad, Y. (2003), *Iterative Methods for Sparse Linear Systems Second Edition*, Society for Industrial and Applied Mathematics.
- Saad, Y. and Schultz, M. H. (1986), ‘GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems.’, *SIAM J. Scientific and Statistical Computing* **7**, 856–869.
- Tarjan, R. E. (1972), ‘Depth-first search and linear graph algorithms’, *SIAM Journal of Computing* **1**(2), 146–160.
- Tarjan, R. E. (1982), ‘A hierarchical clustering algorithm using strong components’, *Inf. Proc. Lett.* **14**, 26–29.
- Tarjan, R. E. (1983), ‘An improved algorithm for hierarchical clustering using strong components’, *Inf. Proc. Lett.* **17**, 37–41.
- Thornquist, H., Keiter, E. R., Hoekstra, R. J., Day, D. M. and Boman, E. G. (2009), A parallel preconditioning strategy for efficient transistor-level circuit simulation, *in* ‘International Conference on Computer-Aided Design (ICCAD’09)’, ACM, pp. 410–417.