Preprint
RAL-P-2011-001

Preconditioners based on strong components

IS Duff, K Kaya

Submitted for publication in SIAM Journal on Matrix Analysis and Applications

January 2011

RAL Library
Science and Technology Facilities Council
Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
OX11 0QX

Tel: +44(0)1235 445384
Fax: +44(0)1235 446403
email: 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 Components[1]

Iain S. Duff[2,3] and Kamer Kaya[3]

ABSTRACT

This paper proposes an approach for obtaining 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 components (Tarjan 1982, Tarjan 1983). To the best of our knowledge, this is the first work that uses Tarjan's algorithm for preconditioning purposes. We describe the method, analyse its performance, and compare it with preconditioners from the literature such as `ILUT` (Saad 1994, Saad 2003) and `XPABLO` (Fritzsche 2010, Fritzsche, Frommer and Szyld 2007) and show that it is the best preconditioner for many matrices.

Keywords: sparse matrices, strong component, preconditioners

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

---

[1]This report is available through the URL `www.numerical.rl.ac.uk/reports/reports.html`. The work of the first author was partially supported by the EPSRC Grant EP/E053351/1. It has also appeared as CERFACS report TR/PA/10/97.

[2]Atlas Centre, STFC Rutherford Appleton Laboratory, Oxon OX11 0QX, UK (`iain.duff@stfc.ac.uk`).

[3]`CERFACS, 42 Av. G. Coriolis, 31057, Toulouse, France (duff@cerfacs.fr, Kamer.Kaya@cerfacs.fr).`

Computational Science and Engineering Department
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

December 21, 2010

Contents

1 Introduction **1**

2 Background **1**

2.1 Finding Strongly Connected Components 2

2.2 Tarjan's Algorithm for Hierarchical Clustering 3

3 A Strong Component based Preconditioner **8**

3.1 Obtaining a Nonzero Diagonal and Scaling 9

3.2 Obtaining the Block Triangular Form 10

3.2.1 Obtaining the Permutation 10

3.2.2 Obtaining the Blocks 10

3.2.3 Combining the Blocks 14

3.2.4 Ordering the Blocks 15

4 Using `SCPRE` with an Iterative Solver **15**

4.1 Robustness 16

5 Experiments **16**

6 Conclusions and Future Work **23**

1 Introduction

Given a linear system $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a real, large, sparse square matrix, 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 components. It permutes the rows and columns of the original matrix $\mathbf{A}$ and obtains a block triangular preconditioning matrix containing a subset of the nonzeros of $\mathbf{A}$ where the maximum size of a diagonal block is smaller than a desired value.

The proposed algorithm that creates $\mathbf{M}$ is a modified version of Tarjan's algorithm `HD` that decomposes a digraph into its strong components 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, it requires distinct edge weights. In this paper, we propose a modification on `HD` which allows us to handle digraphs whose edge weights are not necessarily distinct. We use this modified algorithm to obtain a block triangular matrix $\mathbf{M}$ where the strong components 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 algorithm for preconditioning purposes.

We have conducted several experiments to see the efficiency of the proposed 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 also 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 et al. 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).

For the experiments, we used several circuit simulation, device simulation and computational fluid dynamics (CFD) matrices from the University of Florida Sparse Matrix Collection (`http://www.cise.ufl.edu/research/sparse/matrices/`). Experimental results show that the performance of the proposed algorithm is comparable to that of `ILUT` and `XPABLO` preconditioners for this set of matrices. Furthermore, it performs better than the other preconditioners for device and circuit simulation matrices.

Section 2 gives the notation used in the paper and background on Tarjan's algorithm. 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 $G = (V, E)$ be a directed graph (or a *digraph*) where V is a set of n vertices and E is a set of m edges. An edge going from one vertex, $u \in V$, to another, $v \in V$, is denoted as uv. Let $w(uv)$ be the weight of the edge uv. Figure 2.1 shows a simple digraph with 6 vertices and 13 edges.

If $u = v$ the edge uv is called a *loop*. An alternating sequence of vertices and edges is called a *path*. A path is called *closed* if its first and last vertex are the same. A vertex $u \in V$ is *connected* to $v \in V$ if there is 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.

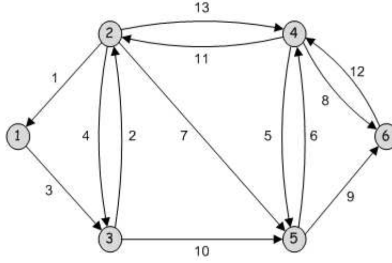


Figure 2.1: A digraph with 6 vertices and 13 edges. The numbers on the edges define an ordering. That is, the first and the last edges are $(2,1)$ and $(2,4)$, respectively.

A digraph $G' = (V', E')$ is a subgraph of G if $V' \subset V$ and $E' \subseteq E \cap (V' \times V')$. Furthermore, if G' is maximally strongly connected, i.e., if there is no strongly connected 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.

Let $G = (V, E)$ be a digraph and $\mathcal{P}(V) = \{V_1, V_2, \cdots, V_k\}$ 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 $\texttt{condense}(G, \mathcal{V}) = G^{\mathcal{V}} = (V^{\mathcal{V}_2}, E^{\mathcal{V}_1})$ where each vertex set $V_i \in \mathcal{V}_2$ 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 as same as those in digraphs. An example for the `condense` operation is given in Figure 2.2.

2.1 Finding Strongly Connected Components

To find the strongly connected components of a digraph, Tarjan proposed an $\mathcal{O}(n+m)$ algorithm `SCC` (Tarjan 1972). The algorithm uses a stack obtained from a depth-first search that records the current path and all the closed paths so far identified. The vertices are numbered and placed in the stack in the order in which they are visited. In addition to the visit number, *visitno*, during the course of the algorithm each vertex u in the stack has a *lowlink* property which is defined as

$$\textit{lowlink}(u) = \min\{\textit{visitno}(v) : u \text{ is connected to } v\}.$$

Note that only the first vertex u of a strongly connected component in the stack has the property that $\textit{lowlink}(u) = \textit{visitno}(u)$. With this algorithm, each strong component eventually appears as a group of vertices at the top of the stack and then its vertices are removed from the stack. We refer the user to Duff, Erisman and Reid (1986) and Tarjan (1972) for a detailed description of the algorithm.

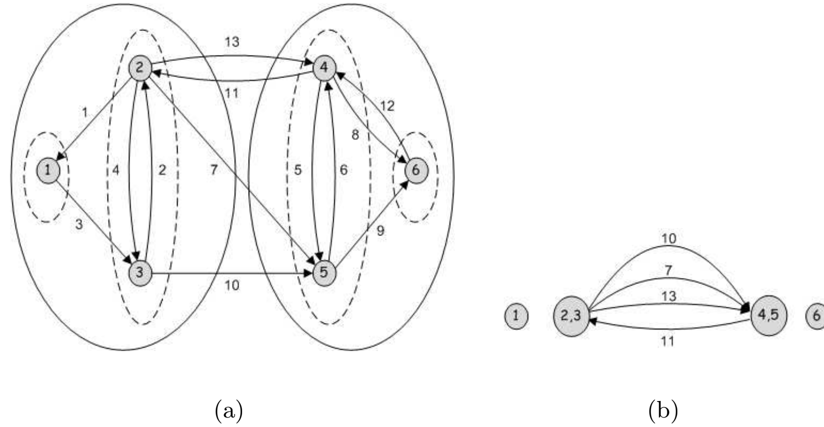


Figure 2.2: An example for the `condense` operation on the digraph in Figure 2.1. The partitions $\mathcal{V}_1 = \{\{1,2,3\},\{4,5,6\}\}$ and $\mathcal{V}_2 = \{\{1\},\{2,3\},\{4,5\},\{6\}\}$ are shown in 2(a). The condensed graph is shown in 2(b).

2.2 Tarjan's Algorithm for Hierarchical Clustering

Let $G = (V, E)$ be a strongly connected digraph with n vertices and m weighted edges. A hierarchical decomposition of G into its strong components can be defined in the following way. Let σ_0 be a permutation of the edges. For $1 \leq i \leq m$, let $\sigma_0(i)$ be the ith edge in σ_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. Consider that edges are added one by one to G_0 in the order determined by σ_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. The hierarchical decomposition of G into its strong components with respect to the edge permutation σ_0 shows which strong components are formed in this process hierarchically. Note that a strong component in a hierarchical decomposition is indeed a strong component of some digraph G_i. However, it is just a strong subgraph of G and not a strong component since G itself is strongly connected and has only one strong component.

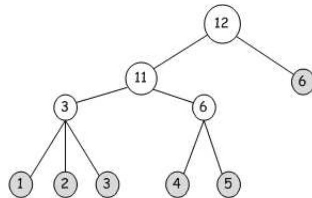


Figure 2.3: The hierarchical decomposition tree for the digraph G and the permutation in Figure 2.1.

A hierarchical decomposition can be represented with a hierarchical decomposition tree T, whose leaf nodes correspond to the vertices in V, non-leaf nodes correspond to the 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 σ_0 is the ordering determined by the edge numbers,

the hierarchical decomposition tree for the digraph in Figure 2.1 is given in Figure 2.3. As the figure shows, after the addition of the 3rd and 6th edges in σ_0, the sets of vertices $\{1,2,3\}$ and $\{4,5\}$ form a strong component, respectively. These strong components are then combined and form a larger one after the addition of the 11th edge. In Fig 2.3, the root of the tree is labelled with 12. Hence, the first 12 edges in σ_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 σ_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 σ_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 $1 \le i \le m$ and the cost of `SCC` is $\mathcal{O}(n + m)$. 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 slightly so that it also works for the case when some edges have equal weights. Note that the edge weights do not play a role in the connectivity of the digraph. In Tarjan's algorithm, they are used in a preprocessing step which defines a permutation σ_0 of the edges. In addition, they are also used for comparison purposes during the course of the algorithm. We eliminate the necessity of the latter by using the indices of the edges with respect to σ_0 for comparison. With this slight modification, the algorithm remains correct when some edges have the same weight.

Tarjan's algorithm `HD` uses a recursive approach and for every recursive call, it gets a digraph $G = (V, E)$, a permutation σ 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 σ_0 which is a permutation of all the edges in the original digraph. A high level description of `HD` is given in Algorithm 1.

Note that the digraph $G = (V, E)$ and the permutation σ in the recursive description of `HD` in Algorithm 1 are not the original digraph and σ_0 except for the initial call. Similarly, for a call of $\texttt{HD}(G = (V, E), \sigma, i)$, 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 problem). Note that in the first step, `HD` knows that G_i is acyclic, i.e., there are $|V|$ strong components of G_i, one for each vertex. If the problem 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 σ. Hence, the algorithm `HD` returns a tree T having a root labelled with $\sigma_0^{-1}(\sigma(|E|))$ and $|V|$ leaves. If the problem size is not one, `HD` 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 should have been combined before the addition of the $(j+1)$th edge. Hence, the algorithm calls $\texttt{HD}(G_j, \sigma, i)$. 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 2.

In Algorithm 2, for the ℓth strong component $SC_\ell = (V_\ell, E_\ell)$, lines 11–14 find the integer i_ℓ such that the subgraph of SC_ℓ containing only its first i_ℓ edges is acyclic. Since G_i, the graph containing the first i edges of G in σ, is known to be acyclic, for SC_ℓ, i_ℓ is set to the index of the last edge uv in σ_ℓ such that $\sigma^{-1}(uv) \le i$. If no such edge exists, i.e., all the edges in E_ℓ

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

1: **if** the number of remaining edges to be investigated is one **then**
2: The last edge $uv \in E$ makes the graph strongly connected. Return a tree T containing the vertices in V as the leaves where the root is labelled with $\sigma_0^{-1}(uv)$.
3: **else**
4: Do a recursive binary search on the edges of the graph to find the first edge such that its addition makes the graph strongly connected by setting $j = \lceil (i + |E|)/2 \rceil$.
5: **if** G_j is strongly connected **then**
6: Continue the binary search with the edges between i and j, inclusive, by calling $\texttt{HD}(G_j, \sigma, i)$.
7: **else**
8: For each non-trivial strong component SC_ℓ of G_j, i.e., those containing more than one vertex, do a recursive call $\texttt{HD}(SC_\ell, \sigma_\ell, i_\ell)$ where σ_ℓ, inherited from σ, is the permutation of the edges of SC_ℓ and i_ℓ is index of the last edge in σ_ℓ such that the first i_ℓ edges of SC_ℓ are known to form an acyclic graph. Note that the trees returned by these recursive calls and the trivial strong components (singleton vertices) make a forest.
9: Continue with the binary search to find the edge which combines all the strong components SC_ℓ of G_j into one strong component. Create a condensed graph in which each vertex represents a strong component of G_j. For each edge $uv \in E$ of G, such that u and v are in different strong components of G_j, this condensed graph contains a distinct edge between the vertices corresponding to the strong components of u and v and it is labelled with the label of uv. Do a recursive call on this graph with a permutation of its edges and largest possible i' such that the first i' edges in the permutation are known to form an acyclic graph.
10: Let T be the tree obtained from the last recursive call in line 9. Replace each leaf of T with the corresponding subtree obtained in line 8 and return T. For trivial components, the subtree is assumed to be a singleton vertex.

Algorithm 2 $T = \mathtt{HD}(G = (V, E), \sigma, i)$. 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 σ_0^{-1}(σ(|E|))
3:   return T
4: j = ⌈(i + |E|)/2⌉
5: if G_j = (V, {σ(k) : 1 ≤ k ≤ j}) is strongly connected then
6:   return T = HD(G_j, σ, i)
7: else
8:   for each strong component SC_ℓ = (V_ℓ, E_ℓ) of G_j do
9:     if |V_ℓ| > 1 then
10:      σ_ℓ = the permutation of E_ℓ ordered with respect to σ
11:      if i = 0 or (σ^{-1}(uv) > i, ∀uv ∈ E_ℓ) then
12:        i_ℓ = 0
13:      else
14:        i_ℓ = max{k : σ^{-1}(σ_ℓ(k)) ≤ i}
15:      T_ℓ = HD(SC_ℓ, σ_ℓ, i_ℓ)
16:    else
17:      T_ℓ = (V_l, ∅)
18:  𝒱_1 = 𝒱_2 = {V_ℓ : SC_ℓ is a strong component of G_j}
19:  𝒱 = {𝒱_1, 𝒱_2}
20:  G^𝒱 = condense(G, 𝒱) = (V^{𝒱_2}, E^{𝒱_1})
21:  σ^𝒱 = the permutation of E^{𝒱_1} ordered with respect to σ
22:  if (σ^{-1}(uv) > j, ∀uv ∈ E^{𝒱_1}) then
23:    i^𝒱 = 0
24:  else
25:    i^𝒱 = max{k : σ^{-1}(σ^𝒱(k)) ≤ j}
26:  T^𝒱 = HD(G^𝒱, σ^𝒱, i^𝒱)
27:  replace the leaves of T^𝒱 with the corresponding trees T_ℓ
28:  return T^𝒱
```

come after the ith edge in σ, i_ℓ is set to 0. 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 jth one at least two strong components should have been combined. To find this edge, another recursive call, $\mathtt{HD}(G^{\mathcal{V}}, \sigma^{\mathcal{V}}, i^{\mathcal{V}})$ is made for the condensed graph $G^{\mathcal{V}}$. The $i^{\mathcal{V}}$ value is set in a similar fashion to i_ℓ as described above. But this time instead of i we use j since we know that the graph $G_j^{\mathcal{V}} = (V^{\mathcal{V}_2}, \{\sigma^{\mathcal{V}}(k) : 1 \leq k \leq j\})$ is acyclic.

At line 6 of Algorithm 2, the size of the problem becomes at most $j-i$ and for lines 15 and 26, 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 $|E| = m$, $\mathtt{t}(m, r)$ be the total complexity of a problem with m edges and r problem size, and k be the number of recursive calls. Then

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

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 $t(m, r) = \mathcal{O}(m \log r)$. 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 2.1. Assume that σ_0 is the ordering described in the figure. In the initial call, step 4 of Algorithm 1 computes $j = 7$ and checks if G_7 is strongly connected. As Figure 2.4 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, HD does not make a recursive call for it. An additional recursive call is made for the condensed graph. Figure 2.5 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 $\sigma^{\mathcal{V}}(1) = \sigma(7)$.

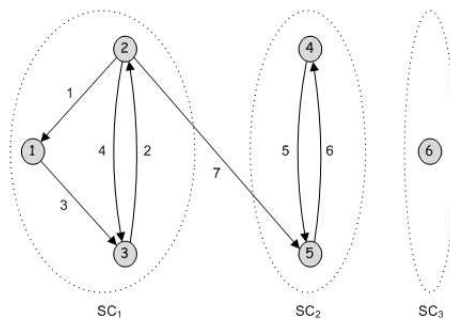
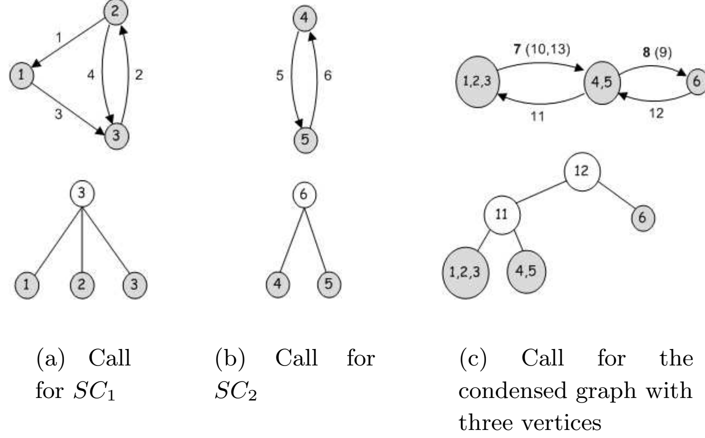


Figure 2.4: Strong components of G_7 for the digraph G given in Figure 2.1

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 one in the permutation σ. In Figure 5(c), these edges, $\sigma(7)$ and $\sigma(8)$, are shown in bold. Tarjan (1983) states that although having less edges in the condensed graphs with this modification is desirable, in practice the added simplicity does not compensate for the cost for the reduction of multigraphs to simple digraphs. This is also validated by our preliminary experiments.



(a) Call for SC_1

(b) Call for SC_2

(c) Call for the condensed graph with three vertices

Figure 2.5: Three recursive calls for the digraph G and σ_0 in Figure 2.1. Internal nodes in trees are labelled with the σ_0^{-1} value of the corresponding edge. Note that the overall hierarchical decomposition tree is already given in Figure 2.3.

3 A Strong Component based Preconditioner

Let $\mathbf{A}$ be a large, nonsingular, $n \times n$ sparse matrix with m nonzeros. The digraph $G = (V, E)$, associated with $\mathbf{A}$, has n vertices in V where v_i corresponds to the ith row/column of $\mathbf{A}$ for $1 \leq i \leq n$ and $v_i v_j \in E$ iff $\mathbf{A}_{ij}$ is nonzero, for $1 \leq i \neq j \leq n$. The weight of an edge is set to the absolute value of the corresponding nonzero. Hence, all of the edges have positive weights.

We assume that $\mathbf{A}$ is irreducible, i.e., it cannot be permuted into block triangular form by simultaneous row/column permutations. Note that a matrix is irreducible if and only if its associated digraph is strongly connected. The directed graph G in Figure 2.1 is the associated digraph for the matrix in Figure 3.1. Since we know that G is strongly connected this matrix is irreducible. Note that the diagonal entries can be omitted in the associated digraph since they do not play a role in the reducibility of the matrix. They will correspond to the loops in the associated digraph, and hence they do not affect the connectivity of the digraph. Note that we only consider transformations of the form $\mathbf{PAP}^T$ where $\mathbf{P}$ is a permutation matrix which means that we preserve the large entries that we previously permuted to the diagonal using an unsymmetric permutation.

We will generate a preconditioner $\mathbf{M}$ with a block upper-triangular structure where the size of each block is smaller than a requested maximum block size mbs. Given $\mathbf{A}$ and mbs, our proposed `SCPRE` algorithm first scales the matrix and then obtains a nonzero diagonal by permuting the columns of $\mathbf{A}$ with a permutation matrix $\mathbf{P}$. After that it generates another permutation matrix $\mathbf{Q}$ such that the block upper triangular part of the matrix $\mathbf{A}' = \mathbf{QAPQ}^T$ contains a large

	1	2	3	4	5	6
1	X		3			
2	1	X	4	13	7	
3		2	X		10	
4		11		X	5	8
5				6	X	9
6				12		X

Figure 3.1: The associated matrix for digraph in Figure 2.1

proportion of the nonzeros and the sum of the absolute values, i.e., magnitudes, of these nonzeros is also large. In this section, we describe the steps of the algorithm `SCPRE` in detail.

3.1 Obtaining a Nonzero Diagonal and Scaling

Let π be a permutation of the columns of $\mathbf{A}$ such that all of the entries, $\mathbf{A}_{i\pi(i)}$, are nonzero. The permutation π has an associated permutation matrix $\mathbf{P}$ such that $\mathbf{AP}$ has the nonzero entries $\mathbf{A}_{i\pi(i)}$ on the diagonal. Finding such a permutation (usually called finding a *transversal*), has been extensively studied and several algorithms have been proposed in the literature (Duff 1981*a*, Duff 1981*b*, Duff and Wiberg 1988, Hopcroft and Karp 1973, Pothen and Fan 1990). Note that a transversal yielding a nonzero diagonal always exists for nonsingular matrices. However, it is usually not unique. Among these transversals, those that maximize the product $\prod_{i=1}^{n} |\mathbf{A}_{i\pi(i)}|$ are called the *maximum product transversals*. As previous work on direct and iterative solvers has shown, to solve linear systems using such a transversal is more promising than using a random one (Benzi, Haws and Tůma 2000, Duff and Koster 2001). The problem of finding a maximum product transversal can be reduced to the well known linear sum assignment problem (LSAP) (Burkard, Dell'Amico and Martello 2009) by taking the logarithm of the magnitudes of the nonzeros. For LSAP, there exists primal-dual algorithms in the literature (Fredman and Tarjan 1987, Jonker and Volgenant 1986). Actually, these are the first polynomial methods proposed for LSAP (Kuhn 1955). By using a primal-dual approach, in addition to the permutation obtained from the primal solution, Olschowska and Neumaier (Olschowska and Neumaier 1996) use the dual solution's variables to 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 and *I*-matrices are also of interest (Benzi et al. 2000, Duff and Koster 2001).

Using the ideas above, Duff and Koster (2001) implemented an algorithm `MC64` such that, given a matrix $\mathbf{A}$, it returns a permutation π and row and column scaling vectors $\mathbf{r}$ and $\mathbf{c}$. Let $\mathbf{P}$ be the corresponding permutation matrix for π and $\mathbf{R}$ and $\mathbf{C}$ be the diagonal scaling matrices such that $\mathbf{R}_{ii} = \texttt{exp}(\mathbf{r}_i)$ and $\mathbf{C}_{ii} = \texttt{exp}(\mathbf{c}_i)$ for $1 \leq i \leq n$ where `exp` is the exponentiation function with base e. Then $\mathbf{A_{MC64}} = (\mathbf{RAC})\mathbf{P}$ is an *I*-matrix. We use the algorithm `MC64` as a preprocessing step and obtain the permuted and scaled matrix $\mathbf{A_{MC64}}$. Note that `MC64` has other options such as finding a random transversal or a maximum bottleneck transversal which maximizes $\texttt{min}_i(|\mathbf{A}_{i\pi(i)}|)$. In our experiments, we observed that permuting the matrix with

respect to the maximum product transversal is the most promising one for the preconditioned iterative solver in terms of the iteration count.

3.2 Obtaining the Block Triangular Form

SCPRE obtains the block upper triangular matrix $\mathbf{M}$ in four steps: first, it creates a permutation σ_0 for the edge set of the corresponding digraph. Second, it uses a modified version of Tarjan's HD algorithm to obtain the initial block structure. Third, it combines the blocks to put more nonzeros into the block diagonal of $\mathbf{M}$. While combining these blocks, SCPRE takes block sizes into account and always obtain blocks of size smaller than or equal to mbs. Finally, it determines an ordering and finds a row/column permutation to put more nonzeros into the upper block diagonal $\mathbf{M}$. While describing these steps, the terms *row/column* and *vertex* can be used interchangeably, as well as the terms *nonzero* and *edge*.

3.2.1 Obtaining the Permutation

As mentioned in Section 2.2, Tarjan proposed HD for hierarchical clustering purposes and sorted the edges with respect to increasing edge weights. That is, for the permutation σ_0 used for hierarchical clustering, if $i < j$ then $w(\sigma_0(i)) \leq w(\sigma_0(j))$. In this work, we propose using two different approaches to obtain the permutation: the first one solely depends on the weights of the edges and sorts them in the order of decreasing edge weights, i.e., we define the permutation σ such as $w(\sigma(i)) \geq w(\sigma(j))$ if $i < j$. The second one uses both the weight information and the sparsity pattern of the matrix. It first uses the reverse Cuthill-McKee (RCM) ordering (Cuthill and McKee 1969, George 1971) to find a symmetric row/column permutation and relabels the vertices of the digraph accordingly. After that, the edges with weights larger than a threshold $\lambda \in [0, 1)$ 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 $j < \ell$. After these edges, those remaining with weights smaller than λ are put to the end of σ_0 in the order of decreasing weights.

Note that when we sort the edges with respect to the decreasing edge weights, we increase the chance that entries with larger magnitudes are within the block diagonal after permuting the rows and columns with SCPRE's output. Also eliminating edges with relatively smaller weights has the same effect when combined with the RCM ordering which tries to reduce the bandwidth of the matrix and put the entries closer to the diagonal.

3.2.2 Obtaining the Blocks

The output of Tarjan's HD algorithm, the decomposition tree T, can be used for preconditioning without modifying the algorithm: Given the maximum block size mbs, by partitioning the leaves to the maximal subtrees containing at most mbs leaves, an initial block diagonal for the preconditioner $\mathbf{M}$ is obtained. For the decomposition tree T in Figure 2.3, the cases for $mbs = 2$ and $mbs = 3$ are given in Figure 3.2. 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, greater than mbs. Hence, there will be 5 blocks after this phase. However, such a combination is possible 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 the line 9 of

HD to check if the current strong component has more than mbs vertices (instead of one 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.

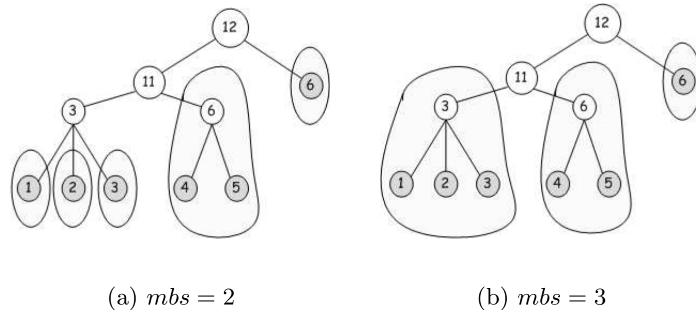


Figure 3.2: Using the output of HD algorithm. Two cases, $mbs = 2$ and $mbs = 3$, are investigated for the decomposition tree in Figure 2.3.

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 we use a finer partition $\mathcal{V}_2$ which contains the vertex sets of strong components obtained by hierarchically decomposing the strong components of size larger than mbs. For example, in Figure 2.4, 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 it 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 3.3.

As Figure 3.3 shows, with the 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 component in the hierarchical decomposition, has less than mbs leaves. The modified algorithm HDPRE, described in Algorithm 3, also handles digraphs which are not strongly connected. Note that, for preconditioning, the only information we need is the block information for the rows/columns. 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, σ and i, HDPRE requires an additional input array *vsize* which stores the number of vertices condensed into each vertex of V. Note that for a simple vertex, this value is one. Hence, for the initial call

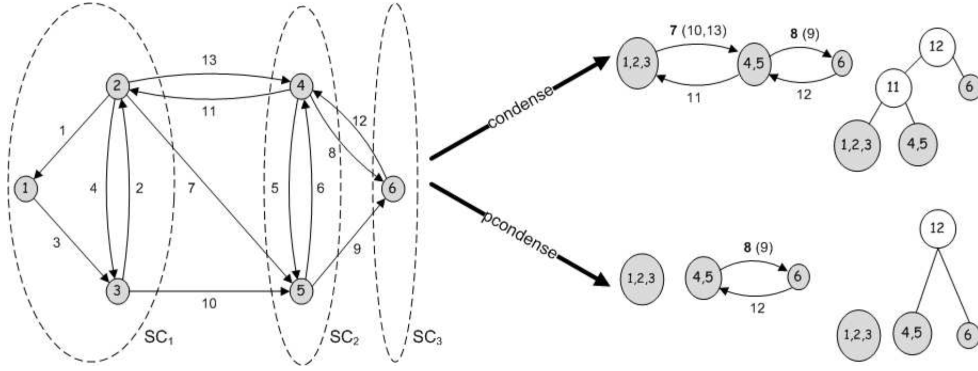


Figure 3.3: Difference between `condense` and `pcondense` operations for the strong components of G_7 given in Figure 2.4. Let $mbs = 3$ so all of the components have a desired number of vertices and $\mathcal{V}_1 = \mathcal{V}_2 = \{\{1,2,3\},\{4,5\},\{6\}\}$. Note that the condensed graphs obtained 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 size of SC_1 and SC_2 is 5, 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.

with $G = (V, E)$, $vsize$ is an array containing $|V|$ ones. On the other hand, for the condensed vertices, this value will be equal to the sum of the $vsize$ values condensed into that vertex. For the condensed digraph in Figure 5(c), $vsize = \{3, 2, 1\}$ 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 $\sum_{v \in V} vsize(v)$ and this number is larger than mbs for all recursive calls because of the size check in line 14 of Algorithm 3.

At first, `HDPRE` checks if the problem size of the recursive call $|E| - i$ is equal to one. If this is the case, it finds the strong components of G. If a strong component $SC_\ell = (V_\ell, E_\ell)$ has $\sum_{v \in V_\ell} vsize(v) > mbs$ vertices then `HDPRE` considers each vertex in V_ℓ 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 not divided. `HDPRE` constructs the $scomp$ array and returns. If the problem size, $|E| - i$ is greater than 1, similar to `HD`, it 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 `HDPRE`$(G_j, \sigma, i, vsize)$. If not, for every strong component $SC_\ell = (V_\ell, E_\ell)$ of G_j such that $\sum_{v \in V_\ell} vsize(v) > mbs$, it makes a recursive call `HDPRE`$(SC_\ell, \sigma_\ell, i_\ell, vsize_\ell)$ and updates the strong component information for the vertices in V_ℓ. This update operation can be considered as further dividing the strong component SC_ℓ hierarchically until all of the strong components obtained during this process contain at most mbs vertices.

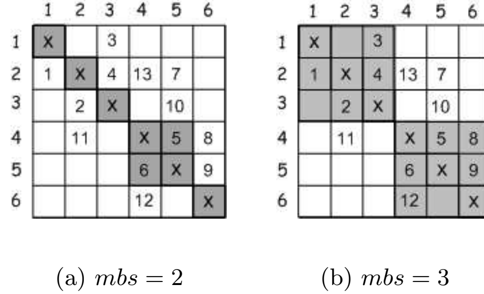
Similarly to `HD`, at line 28, `HDPRE` makes one more recursive call for the condensed graph $G^{\mathcal{V}}$ where the definition of the vertex partition $\mathcal{V}$ (in line 22) is modified as described above. In `HD`, each vertex in the condensed graph corresponds to a strong component of G_j which 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 $G^{\mathcal{V}}$, before the call, `HDPRE` checks if $G^{\mathcal{V}}$ is known to be acyclic. Note that if $i^{\mathcal{V}} = |E^{\mathcal{V}_1}|$, no strong component with two or more

Algorithm 3 $scomp = \texttt{HDPRE}(G = (V, E), \sigma, i, vsize)$ (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_ℓ = (V_ℓ, E_ℓ) of G do
4:       if Σ_{v∈V_ℓ} vsize(v) > mbs then
5:          consider each v ∈ V_ℓ as a strong component
6:       else
7:          ∀v ∈ V_ℓ, scomp(v) = ℓ
8:    return scomp
9: j = ⌈(i + |E|)/2⌉
10: if G_j = (V, {σ(k) : 1 ≤ k ≤ j}) is strongly connected then
11:    return scomp = HDPRE(G_j, σ, i, vsize)
12: else
13:    for each strong component SC_ℓ = (V_ℓ, E_ℓ) of G_j do
14:       if Σ_{v∈V_ℓ} vsize(v) > mbs then
15:          σ_ℓ = the permutation of E_ℓ ordered with respect to σ
16:          compute i_ℓ as in Algorithm 2
17:          vsize_ℓ(v) = vsize(v), ∀v ∈ V_ℓ
18:          scomp_ℓ = HDPRE(SC_ℓ, σ_ℓ, i_ℓ, vsize_ℓ)
19:          update scomp according to scomp_ℓ
20:    𝒱_1 = {V_ℓ : SC_ℓ is a strong component of G_j}
21:    𝒱_2 = {V_ℓ' : SC_ℓ' = (V_ℓ', E_ℓ') is a strong component in scomp}
22:    𝒱 = {𝒱_1, 𝒱_2}
23:    G^𝒱 = pcondense(G, 𝒱, mbs) = (V^{𝒱_2}, E^{𝒱_1})
24:    σ^𝒱 = the permutation of E^{𝒱_1} ordered with respect to σ
25:    compute i^𝒱 as in Algorithm 2
26:    if i^𝒱 ≠ |E^{𝒱_1}| then
27:       vsize^𝒱(v_ℓ') = Σ_{v∈V_ℓ'} vsize(v), ∀V_ℓ' ∈ 𝒱_2
28:       scomp^𝒱 = HDPRE(G^𝒱, σ^𝒱, i^𝒱, vsize^𝒱)
29:       update scomp with respect to scomp^𝒱
30:    return scomp
```

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 Fig 3.1, HDPRE generates the blocks for the cases $mbs = 2$ and $mbs = 3$ as given in Figure 4(a) and Figure 4(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 4(b), the condensed graph has 3 vertices where 2 of them will combine with the 12th edge in σ_0.



(a) $mbs = 2$ (b) $mbs = 3$

Figure 3.4: Initial block structure of the preconditioner after HDPRE algorithm. Two cases, $mbs = 2$ and $mbs = 3$, are investigated for the matrix in Figure 3.1.

3.2.3 Combining the Blocks

After HDPRE, an initial block diagonal partition is obtained. In the next phase, SCPRE performs a loop on the nonzeros which are not contained in a block of the diagonal to see if it is possible to put more nonzeros into the block diagonal by combining initial blocks. To do this, SCPRE first constructs a condensed simple digraph $\vec{H}$ where the vertices of $\vec{H}$ correspond to the diagonal blocks and multiple edges are combined as a single edge with a weight that is the sum of the weights of the combined edges. After that, an undirected graph H is constructed by recombining the edges uv and vu into a single edge with weight equal to $w(uv) + w(vu)$.

After H is obtained, its edges are visited in an order corresponding to a permutation σ_H. This permutation is consistent with the original permutation σ_0. That is, if the edges of the original digraph are sorted in descending order with respect to the edge weights, σ_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 $\vec{H}$, relabel the vertices of H accordingly, and order the edges as described in Section 3.2.1. Let $vsize(u)$ be the number of rows/columns in a block corresponding to the vertex u. The pseudo code of this phase is given in Algorithm 4.

Algorithm 4 comb($\sigma, vsize$), mbs is global.

```
1: for each uv ∈ σ_H do
2:   if vsize(u) + vsize(v) ≤ mbs then
3:     merge blocks u and v
4:     update vsize accordingly
```

Assume that SCPRE constructs σ_0 by sorting the edges with respect to decreasing weights. For the matrix given in Figure 4(a), if $w(2) + w(4) > w(1)$ then vertices 2 and 3 are combined. Since $mbs = 2$ and there is no edge between vertices 1 and 6, these two vertices remain as singletons.

3.2.4 Ordering the Blocks

After `comb`, no block can be combined with the others to obtain a larger block with size no larger than *mbs* to put more nonzeros into the block diagonal. However, the order of these blocks is still important since it changes the nonzeros in the upper part of $\mathbf{M}$.

Let $G = (V, E)$ be the digraph associated with the matrix and k be the number of its diagonal blocks after `comb`. Let $\mathcal{V}_1 = \{V_1, V_2, \cdots, V_k\}$ be a partition of V such that the vertices in V_i correspond to the rows/columns of ith block. Let $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_1\}$ and $G^{\mathcal{V}} = \texttt{condense}(G, \mathcal{V})$ 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. Unfortunately, we observed that this usually does not happen for real test matrices.

The problem of finding a good block permutation, which maximizes the number of nonzeros in the upper part of 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).

We use a simple greedy heuristic called `gperm` to solve the weighted version of the problem. Let $G^{\mathcal{V}}$ be the condensed graph described above. For each vertex $u \in V^{\mathcal{V}}$, let $tweight(u) = \sum_{uv \in E^{\mathcal{V}}} w(uv)$. `gperm`'s main body is a **for** loop where at the ith iteration, it chooses the vertex u with maximum *tweight* and assigns it as the ith vertex in the permutation. It then removes u from $V^{\mathcal{V}}$, its edges from $E^{\mathcal{V}}$, and continues with the next iteration.

4 Using SCPRE with an Iterative Solver

The iterative solver we use in our experiments is MATLAB's left-preconditioned `GMRES` (Saad and Schultz 1986) with restarts. The template for this can be found in Barrett, Berry, Chan, Demmel, Donato, Dongarra, Eijkhout, Pozo, Romine and der Vorst (1994). Let $\mathbf{A} = \mathbf{D} + \mathbf{U} + \mathbf{L}$ be the scaled and permuted matrix so that $\mathbf{M} = \mathbf{D} + \mathbf{U}$ is the preconditioner for $\mathbf{A}$. Now the computation $\mathbf{M}^{-1}\mathbf{A}\mathbf{x}$ becomes

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

Note that `gperm` tries to maximize the total magnitude in $\mathbf{D}$ and $\mathbf{U}$. As a consequence, $\mathbf{L}$ usually contains many fewer nonzeros than $\mathbf{A}$. Hence computing the vector $\mathbf{z} = \mathbf{L}\mathbf{x}$ usually takes very little time and the main operation is to compute $(\mathbf{D} + \mathbf{U})^{-1}\mathbf{z}$.

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 ith 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 2004, Davis Sept. 2006) before the MATLAB sparse factorization. We then solve the upper block triangular system $\mathbf{M}\mathbf{y} = \mathbf{z}$ 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

Although it is very rare, we observe that the preconditioner $\mathbf{M}$ obtained by `SCPRE` is singular for some matrices in our test set. That is, some of the blocks on the diagonal of $\mathbf{M}$ are singular. Note that since the scaled and permuted matrix is an I-matrix, the diagonal blocks of $\mathbf{M}$ are I-matrices as well. Hence, with high probability, the diagonal blocks will be nonsingular and well-conditioned.

Thus, when using the MATLAB factorization, we guard against this potential problem by using the simple and cheap stability check proposed and used by (Fritzsche 2010) for `XPABLO`. That is, if n_i is the dimension of $\mathbf{D_i}$, after computing $\mathbf{L_i}$ and $\mathbf{U_i}$, we check whether

$$\left|1 - \frac{||\mathbf{U_i}^{-1}\mathbf{L_i}^{-1}\mathbf{x}||}{||\mathbf{e}||}\right| < \sqrt{\epsilon_M}, \tag{4.1}$$

where $\mathbf{e} = (1, \ldots, 1)^T$ is an $n_i \times 1$ column vector, $\mathbf{x} = \mathbf{D_i}\mathbf{e}$, and ϵ_M is machine epsilon. He suggests that, if a block does not satisfy (4.1), $\mathbf{D_i}$ is replaced either by $\mathbf{U_i}$ or $\mathbf{L_i}$ according to whether he is solving a block upper or lower triangular system respectively. For `SCPRE`, we always use the factor having the largest Frobenius norm to replace $\mathbf{D_i}$, where the Frobenius norm of an $n \times n$ matrix B is given by

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

5 Experiments

We conduct several experiments to see the influence of the parameters on the performance of the algorithm. These are the maximum block size mbs and the permutation σ_0. For the experiments, we use two sets of matrices from the University of Florida Sparse Matrix Collection[1]. The first set contains circuit simulation matrices and the second set contains computational fluid dynamics (CFD) and device simulation matrices. The list of these matrices is given in Table 5.1.

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

$$\frac{||\mathbf{M}^{-1}(\mathbf{A}\overline{\mathbf{x}} - \mathbf{b})||}{||\mathbf{M}^{-1}\mathbf{b}||} < \epsilon$$

where $\overline{\mathbf{x}}$ is the solution found. 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 iteration count when this criterion is satisfied. However, when this does not happen, that is, when the residual after 1000 iterations is bigger than ϵ, or when the MATLAB GMRES solver detects stagnation, we put a dash symbol (-) to denote this. 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 7.8 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.

[1] `http://www.cise.ufl.edu/research/sparse/matrices/`

Table 5.1: Properties of the matrices used for the experiments. n is the dimension of the matrix, m is the number of nonzeros, $psym$ is the pattern symmetry and $nsym$ is the numerical symmetry, where the matrix is symmetric when the value is 1 in each case.

Description	Group	Matrix	n	m	$psym$	$nsym$
Circuit simulation problems	AMD	*G2_circuit*	150102	726674	1	1
	Bomhof	*circuit_2*	4510	21199	0.807	0.415
	Bomhof	*circuit_3*	12127	48137	0.770	0.300
	Bomhof	*circuit_4*	80209	307604	0.829	0.364
	Grund	*meg1*	2904	58142	0.002	0.002
	Grund	*meg4*	5860	25258	1	1
	Hamm	*bcircuit*	68902	375558	1	0.908
	Hamm	*hcircuit*	105676	513072	1	0.195
	Hamm	*memplus*	17758	99147	1	0.496
	IBM_Austin	*coupled*	11341	97193	1	0.782
	IBM_EDA	*ckt11752_dc_1*	49702	333029	0.984	0.736
	Rajat	*rajat03*	7602	32653	1	0.402
	Rajat	*rajat27*	20640	97353	0.965	0.304
	Sandia	*ASIC_100k*	99340	940621	1	0.003
	Sandia	*mult_dcop_01*	25187	193276	0.614	0.003
Computational fluid dynamics problems	DRIVCAV	*cavity16*	4562	137887	0.953	0.654
	DRIVCAV	*cavity26*	4562	138040	0.953	0
	Garon	*Garon1*	3175	84723	1	0.672
	Garon	*Garon2*	13535	373235	1	0.673
	Shyy	*shyy161*	76480	329762	0.726	0.182
	Simon	*raefsky2*	3242	293551	1	0.098
Semiconductor device problems	Sanghavi	*ecl32*	51993	380415	0.922	0.603
	Schenk_IBMSDS	*2D_27628_bjtcai*	27628	206670	1	0.219
	Schenk_IBMSDS	*2D_54019_highk*	54019	486129	0.998	0.190
	Schenk_IBMSDS	*3D_28984_Tetra*	28984	285092	0.987	0.361
	Schenk_IBMSDS	*3D_51448_3D*	51448	537038	0.992	0.187
	Schenk_IBMSDS	*ibm_matrix_2*	51448	537038	0.992	0.186
	Schenk_IBMSDS	*matrix_9*	103430	1205518	0.997	0.174
	Schenk_IBMSDS	*matrix-new_3*	125329	893984	0.987	0.283
	Schenk_ISEI	*igbt3*	10938	130500	1	0.172
	Schenk_ISEI	*nmos3*	18588	237130	1	0.167
	Wang	*wang3*	26064	177168	1	0.978
	Wang	*wang4*	26068	177196	1	0.046

In the MATLAB implementation of `ILUT`, for the jth column of the incomplete $\mathbf{L}$ and $\mathbf{U}$, entries smaller in magnitude than $dtol \times ||\mathbf{A}_{*j}||$ are deleted from the factor where $||\mathbf{A}_{*j}||$ is the norm of the jth column of $\mathbf{A}$. However, the diagonal entries of $\mathbf{U}$ are always kept to avoid a singular factor. To use `ILUT` based preconditioners, we use `RCM` before computing the incomplete factorization of the matrix. For `XPABLO` preconditioners, we use the `LX` and `UX` variants with the parameters given in Fritzsche (2010) and Fritzsche et al. (2007). As suggested in these papers, we set the minimum and maximum block sizes to 200 and 1000, respectively.

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

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

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

$$mem_{\texttt{SCPRE}} = mem_{\texttt{XPABLO}} = \frac{\sum_{i=1}^{k} (nz(\mathbf{L_i}) + nz(\mathbf{U_i}))}{nz(\mathbf{A})},$$

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 ith block in $\mathbf{D}$. Note that the relative memory requirements of the preconditioners can give an idea for the cost of computing $\mathbf{M}^{-1}\mathbf{Ax}$. Assuming x is a dense vector, a preconditioned `GMRES` iteration will require approximately $nz(\mathbf{A})(1+mem_X)$ operations for the preconditioner `X`.

There are three parameters for the proposed algorithm: the first is the maximum block size, mbs, the second is the permutation for the nonzeros, denoted by σ_0, and the third is λ used to generate σ_0 when `RCM` is used. Our experiments (not reported on here) show that setting $\lambda = 0.05$ works well for most of the matrices. Hence, in our experiments, we use this value. Table 5.2 shows that increasing mbs will decrease the number of iterations for convergence but will increase the relative memory requirement.

Table 5.2: Effect of the maximum block size mbs for `SCPRE` used to solve the matrix *rajat27*. The permutation σ_0 is obtained by ordering the edges with respect to decreasing edge weights. $\mathbf{M} = \mathbf{D} + \mathbf{U} = \mathbf{A} - \mathbf{L}$ is the preconditioner, nb is the number of blocks in $\mathbf{D}$, $mem_{\texttt{SCPRE}}$ is the relative memory requirement and $iters$ is the number of iterations required by the preconditioned `GMRES(50)` for convergence.

mbs	$\|\|\mathbf{M}\|\|_F$	$\|\|\mathbf{L}\|\|_F$	nb	$mem_{\texttt{SCPRE}}$	$iters$
1000	163.88	33.11	26	1.25	7
2000	164.22	25.37	15	1.66	5
3000	164.65	21.69	11	1.53	5
4000	164.78	19.32	8	2.03	4
5000	165.38	12.38	7	2.21	3

We conduct some experiments to show the effect of our choice of σ_0 on the performance of our algorithm. Note that in Tarjan's algorithm 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 5.3 shows, our decision to sort the edges

in decreasing order with respect to the edge weights works much better. Although we do not record our runs using Tarjan's σ_0, its performance was also considerably worse than our strategy.

Table 5.3: Effect of the permutation σ_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 250. For each case, the Frobenius norms of $\mathbf{M}$ and $\mathbf{L}$, the relative memory requirement and the number of iterations for preconditioned GMRES are given.

	Decreasing				Random			
Matrix	$\|\|\mathbf{M}\|\|_F$	$\|\|\mathbf{L}\|\|_F$	$mem_{\texttt{SCPRE}}$	*iters*	$\|\|\mathbf{M}\|\|_F$	$\|\|\mathbf{L}\|\|_F$	$mem_{\texttt{SCPRE}}$	*iters*
circuit_2	93.72	5.67	0.90	**4**	93.30	10.49	0.97	20
circuit_3	140.64	10.86	1.20	**21**	139.53	20.69	1.25	-
meg1	74.95	3.08	0.85	**4**	74.51	8.68	0.86	10
meg4	79.75	0.00	1.30	**1**	79.75	0.00	1.32	**1**
memplus	170.07	0.02	0.96	**8**	169.49	14.01	0.94	69
coupled	128.55	2.51	1.04	**12**	125.70	27.02	0.98	56
rajat03	101.26	0.00	0.93	**2**	100.04	15.68	1.65	141
rajat27	166.39	35.16	1.17	**7**	165.65	38.49	1.11	57
mult_dcop_01	257.87	0.00	0.78	**1**	257.87	0.00	0.78	**1**

Looking again at Table 5.2 we see that the number of iterations is lower when the memory used to store the preconditioner is higher and the value of $||\mathbf{L}||_F$ is lower. However, as Table 5.3 shows, $||\mathbf{L}||_F$ is more important. That is, the iterative solver may suffer due to large entries in $\mathbf{L}$ even when its memory usage is larger. Although our experiments show that $||\mathbf{L}||_F$ and the number of iterations required for convergence are not strictly correlated, smaller $||\mathbf{L}||_F$ norms usually imply faster convergence.

Table 5.4 shows the performance of SCPRE, XPABLO and ILUT for circuit simulation matrices. As the table shows, for some matrices such as *meg4* and *mult_dcop_01*, the block triangular form from SCPRE can contain all of the nonzeros and hence the solution can be found in one iteration. Figure 5.1 shows the structure of *meg4* and *mult_dcop_01* before and after permutation by SCPRE. In nearly all cases SCPRE(*dec*) outperforms SCPRE(RCM) so we recommend this variant on this class of matrices. In general all the preconditioners work well for circuit simulation problems, and the ILUT preconditioners and SCPRE(*dec*) converge for all matrices in this set. Both variants of XPABLO fail to converge for *bcircuit* and *ckt11752_dc_1* and XPABLO requires fewer iterations in only two cases and then not by a significant margin. Thus SCPRE(*dec*) is the best block based preconditioner on this set of matrices.

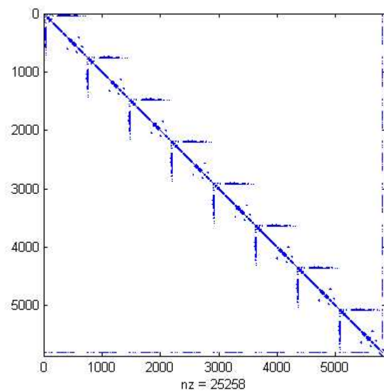
The comparison of SCPRE(*dec*) with the ILUT preconditioners is less clear cut. There are only three cases where ILUT(10^{-4}) requires more iterations than SCPRE(*dec*), although in one of these cases (*bcircuit*) it requires many more iterations and on another (*mult_dcop_01*) it requires very much more memory. Indeed, in nearly all cases, ILUT(10^{-4}) requires more memory sometimes substantially more. The memory requirements of ILUT(10^{-3}) are less than for ILUT(10^{-4}) but it still usually requires more memory than SCPRE(*dec*) and has fewer iterations on only 7 out of 15 matrices. There are only two cases where the ILUT preconditioners require significantly fewer iterations, namely *G2_circuit* and *ckt11752_dc_1* although in both cases the ILUT preconditioners require more memory. From our previous experiments we might hope to reduce the iteration count for SCPRE(*dec*) by increasing *mbs* and thus increasing our memory requirement. For *G2_circuit*, if we increase *mbs* to 5000, then the number of iterations drops to 95 and our relative memory requirement increases to 6.10 which is comparable although slightly worse than the figures for ILUT(10^{-3}). For *ckt11752_dc_1*, increasing *mbs* to 5000 reduces our iteration count to only 5 although our relative memory requirement is then 3.93. However, by increasing *mbs* to only

Table 5.4: Numbers of iterations and relative memory requirements for XPABLO, ILUT and SCPRE when they are used for circuit simulation matrices. For SCPRE, *mbs* is set to 1000 and we show the results using two permutations for σ_0, based on RCM and descending order. For XPABLO, the parameters suggested in Fritzsche (2010) are used and the minimum and maximum block sizes are set to 200 and 1000, respectively. For ILUT, the drop tolerance is set to 10^{-3} and 10^{-4}.

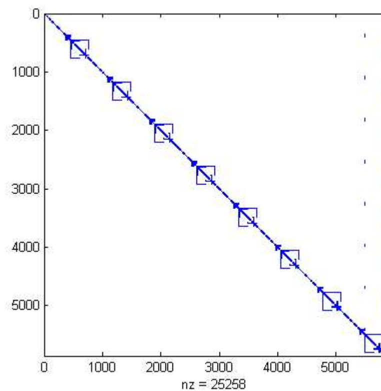
Matrix	XPABLO UX	XPABLO LX	ILUT 10^{-3}	ILUT 10^{-4}	SCPRE dec	SCPRE RCM
G2_circuit	743	727	89	**38**	642	571
	1.77	1.77	5.47	13.30	2.23	2.16
circuit_2	13	13	6	4	**3**	4
	1.17	1.17	1.23	1.77	2.17	2.06
circuit_3	572	572	3	**2**	14	78
	1.25	1.25	2.16	3.7	1.40	1.24
circuit_4	18	18	**2**	3	20	21
	0.97	0.97	1.03	2.70	0.99	0.98
meg1	9	9	**3**	**3**	**3**	4
	0.91	0.91	0.46	0.71	0.70	0.58
meg4	**1**	**1**	2	2	**1**	**1**
	1.35	1.35	0.49	0.49	1.59	1.61
bcircuit	-	-	238	136	**16**	386
	1.32	1.32	1.10	1.22	1.38	1.15
hcircuit	7	7	7	**4**	11	16
	1.19	1.19	1.54	1.94	1.25	1.18
memplus	7	7	12	8	**6**	13
	0.80	0.80	0.77	0.94	1.01	0.59
coupled	13	13	7	**5**	10	13
	1.05	1.05	1.70	3.28	1.33	1.25
ckt11752_dc_1	-	-	**11**	18	213	-
	1.02	1.02	2.55	9.02	1.32	1.09
rajat03	4	4	3	**2**	**2**	**2**
	0.77	0.77	0.92	0.93	0.98	0.96
rajat27	15	15	6	**4**	7	14
	1.16	1.16	1.58	2.27	1.25	1.21
ASIC_100k	**5**	**5**	**5**	**5**	**5**	**5**
	0.69	0.69	3.11	8.27	0.81	0.74
mult_dcop_01	12	12	6	4	**1**	**1**
	1.04	1.04	22.48	47.52	0.86	0.97

3000 we require only 11 iterations with a relative memory cost of only 1.45. Also, as Table 5.2 shows, we can use a larger *mbs* for *rajat27* to obtain faster convergence and less relative memory requirement than ILUT(10^{-4}).

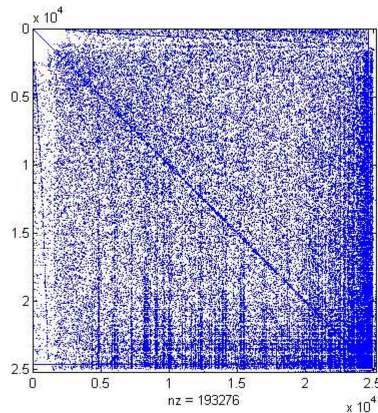
Thus, although we require a better control on *mbs* than at present and this choice is problem dependent, it is similar to the problem of choosing a drop tolerance for the ILUT preconditioners, and thus we feel we can we recommend using SCPRE(*dec*) for circuit simulation matrices especially when the amount of memory to store the preconditioner is the main concern.



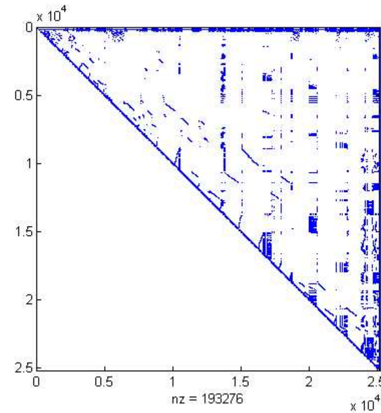
(a) *meg4*



(b) Preconditioner for *meg4* generated by SCPRE



(c) *mult_dcop_01*



(d) Preconditioner for *mult_dcop_01* generated by SCPRE

Figure 5.1: Matrices *meg4* and *mult_dcop_01* and the preconditioners obtained by the algorithm SCPRE with $mbs = 250$. Note that the numbers of nonzeros in the original matrices and preconditioners are equal which means that SCPRE manages to put all of the nonzeros in the block upper triangular form.

We show, in Table 5.5, results for a second set of test problems from applications in CFD and device simulation problems. We see from these results that the performance of the preconditioners is highly influenced by the application.

For the CFD matrices in the top part of the table, we see that the SCPRE(RCM) works much

Table 5.5: Numbers of iterations and relative memory requirements for `XPABLO`, `ILUT` and `SCPRE` when they are used for CFD and device simulation matrices. For `SCPRE`, *mbs* is set to 1000 and we show the results using two permutations for σ_0, based on `RCM` and descending order. For `XPABLO`, the parameters suggested in Fritzsche (2010) are used and the minimum and maximum block sizes are set to 200 and 1000, respectively. For `ILUT`, the drop tolerance is set to 10^{-3} and 10^{-4}.

Matrix	`XPABLO` `UX`	`XPABLO` `LX`	`ILUT` 10^{-3}	`ILUT` 10^{-4}	`SCPRE` dec	`SCPRE` `RCM`
cavity16	35	34	7	**4**	345	54
	2.03	2.03	4.20	5.33	1.21	1.43
cavity26	26	37	5	**3**	-	43
	5.30	5.30	5.87	7.70	1.26	3.22
Garon1	38	39	8	**4**	267	30
	6.01	6.01	3.41	5.55	1.79	5.96
Garon2	123	135	14	**7**	-	193
	6.75	6.75	4.56	8.48	1.17	5.83
raefsky2	36	36	12	**6**	48	33
	2.91	2.91	3.13	5.86	2.06	2.03
shyy161	**9**	-	11	-	-	-
	1.27	1.27	5.54	6.63	2.29	2.32
ecl32	67	87	35	**15**	34	32
	5.71	5.71	5.44	12.23	5.74	5.74
2D_27628_bjtcai	48	45	-	**1**	142	146
	1.85	1.85	2.58	5.81	2.22	2.48
2D_54019_highk	140	139	**16**	-	192	340
	2.45	2.45	18.68	41.27	1.54	1.54
3D_28984_Tetra	234	232	-	-	98	**89**
	2.82	2.82	1.98	5.20	2.65	2.61
3D_51448_3D	27	26	-	-	15	**12**
	5.80	5.80	20.97	36.83	5.24	5.93
ibm_matrix_2	24	23	-	-	**13**	14
	5.39	5.39	23.24	38.27	5.12	5.76
matrix_9	**168**	182	-	-	205	177
	4.96	4.96	37.32	42.57	2.27	3.88
matrix-new_3	**1**	**1**	-	-	**1**	**1**
	5.07	5.07	21.57	29.86	3.75	4.64
igbt3	99	107	42	**19**	21	20
	4.62	4.62	1.82	2.90	4.09	3.32
nmos3	37	38	**1**	**1**	24	28
	4.13	4.13	2.55	4.09	3.31	2.15
wang3	100	100	20	**10**	79	76
	2.42	2.42	8.02	25.78	3.85	2.98
wang4	38	27	13	**7**	19	37
	4.15	4.15	5.04	12.36	4.55	3.75

better than SCPRE(*dec*) although it requires more storage. Although we are not sure why the RCM ordering works well for these matrices, we believe that SCPRE(*dec*) performs badly because the edges with relatively larger weights in the digraph associated with the scaled and permuted matrix form long cycles. Also, the vertices on these cycles do not have many edges between them. Hence, the blocks that SCPRE creates contain less nonzeros than usual even when their sizes are big. By using RCM, which reduces the bandwidth of the matrix, SCPRE is able to find smaller strong components during the course of the algorithm. On these matrices, the SCPRE(RCM) and XPABLO based preconditioners are comparable with the latter usually have a lower iteration count but a higher memory requirement. The ILUT based preconditioners are clearly the best preconditioners on this set although their memory requirements are often somewhat higher.

For the device simulation matrices in the lower part of the table, the results are completely different. There is not much to choose between the versions of our SCPRE preconditioners although we prefer SCPRE(*dec*) and use that in the comparisons with the other approaches. Perhaps the most noticeable thing is that the ILUT based preconditioners are much less robust than they were in our previous experiments. Although the iteration counts are low when ILUT preconditioned GMRES converges, it fails to converge on 6 out of 12 matrices. The block based preconditioners are far more robust on this set with convergence for all the test matrices. SCPRE(*dec*) has a lower iteration count than the XPABLO based preconditioners for 8 out of 12 matrices, sometimes significantly lower and the memory requirements on both approaches are similar with SCPRE(*dec*) requiring less memory on 9 matrices. We therefore feel that we can recommend SCPRE as the best preconditioner for our device simulation matrices.

6 Conclusions and Future Work

Given a linear system $\mathbf{Ax} = \mathbf{b}$, we have proposed a method to construct a generic, block triangular preconditioner. The proposed approach is based on Tarjan's algorithm HD for hierarchical decomposition of a digraph into its strong components. Although our preconditioner SCPRE does not perform well for CFD matrices, we obtain promising results for 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 way that a permutation σ_0 of the edges is obtained and the maximum block size *mbs*. For σ_0, we tried two approaches: the first sorts the edges in the order of decreasing weights. With this approach, we wanted to include the 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 to put 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 σ_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 SCPRE, *mbs* is first set by the user and then the relative memory requirement is computed. In future work, we will look for a self-tuning mechanism which

enables SCPRE to determine *mbs* automatically given the maximum available memory to store the preconditioner.

Acknowledgments

The authors thank Dr. Philip Knight of the University of Strathclyde for bringing the elegant hierarchical decomposition algorithm of Tarjan to our attention.

References

Amestoy, P., Davis, T. A. and Duff, I. S. (2004), 'Algorithm 837: AMD, An approximate minimum degree ordering algorithm', *ACM Transactions on Mathematical Software* **30**, 381–388.

Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. and der Vorst, H. V. (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. B. (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, pp. 159–165.

Benzi, M., Haws, J. C. and Tůma, M. (2000), 'Preconditioning highly indefinite and nonsymmetric matrices', *SIAM Journal on Scientific Computing* **22**(4), 1333–1353.

Burkard, R., Dell'Amico, M. and Martello, S. (2009), *Assignment Problems*, SIAM, Philadelphia, PA, USA.

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* 'Proc. 24th Nat. Conf. ACM', pp. 157–172.

Davis, T. A. (Sept. 2006), *Direct Methods for Sparse Linear Systems*, SIAM, Philadelphia, PA, USA.

Dayar, T. and Stewart, W. J. (2000), 'Comparison of partitioning techniques for two-level iterative solvers on large, sparse markov chains', *SIAM Journal on Scientific Computing* **21**, 1691–1705.

Duff, I. S. (1981*a*), 'Algorithm 575: Permutations for a zero-free diagonal', *ACM Transactions of Mathematical Software* **7**, 387–390.

Duff, I. S. (1981*b*), 'On algorithms for obtaining a maximum transversal', *ACM Transactions of Mathematical Software* **7**, 315–330.

Duff, I. S. and Koster, J. (2001), 'On algorithms for permuting large entries to the diagonal of a sparse matrix', *SIAM Journal on Matrix Analysis and Applications* **22**, 973–996.

Duff, I. S. and Wiberg, T. (1988), 'Remarks on implementations of $\mathcal{O}(n^{1/2}\tau)$ assignment algorithms', *ACM Transactions of Mathematical Software* **14**, 267–287.

Duff, I. S., Erisman, A. M. and Reid, J. K. (1986), *Direct Methods for Sparse Matrices*, Oxford University Press.

Fredman, M. L. and Tarjan, R. E. (1987), 'Fibonacci heaps and their uses in improved network optimization algorithms', *Journal of the ACM* **34**, 596–615.

Fritzsche, D. (2010), Overlapping and Nonoverlapping Orderings for Preconditioning, PhD thesis, Temple University, May 2010.

Fritzsche, D., Frommer, A. and Szyld, D. B. (2007), 'Extensions of certain graph-based algorithms for preconditioning', *SIAM Journal on 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.

George, A. (1971), Computer implementation of the finite-element method, PhD thesis, Department of Computer Science, University of Ilinois at Urbana-Champaign.

Hopcroft, J. E. and Karp, R. M. (1973), 'An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs', *SIAM Journal of Computing* **2**(4), 225–231.

Jonker, R. and Volgenant, A. (1986), 'Improving the hungarian assignment algorithm', *Operations Research Letters* **5**, 171–175.

Kuhn, H. W. (1955), 'The hungarian method for the assignment and transportation problems', *Naval Research Logistics Quarterly* **2**, 83–97.

Olschowska, M. and Neumaier, A. (1996), 'A new pivoting strategy for Gaussian elimination', *Linear Algebra Appl.* **240**, 131–151.

Pothen, A. and Fan, C.-J. (1990), 'Computing the block triangular form of a sparse matrix', *ACM Transactions of Mathematical Software* **16**, 303–324.

Saad, Y. (1994), 'ILUT: A dual threshold incomplete ILU factorization', *Numerical Linear Algebra Appl.* **1**, 387–402.

Saad, Y. (2003), *Iterative Methods for Sparse Linear Systems. 2nd edition*, SIAM, Philadelphia, PA.

Saad, Y. and Schultz, M. H. (1986), 'GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems', *SIAM Journal on 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', *Information Processing Letters* **14**, 26–29.

Tarjan, R. E. (1983), 'An improved algorithm for hierarchical clustering using strong components', *Information Processing Letters* **17**(1), 37–41.