Numerically-aware orderings for sparse symmetric indefinite linear systems

Numerically-aware orderings for sparse symmetric indefinite linear systems

Jonathan D. Hogg[1], Jennifer A. Scott[1], H. Sue Thorne[1]

Abstract

Sparse symmetric indefinite problems arise in a large number of important application areas; they are often solved through the use of an LDL^T factorization via a sparse direct solver. Whilst for many problems, prescaling the system matrix A is sufficient to maintain stability of the factorization, for a small but important fraction of problems numerical pivoting is required. Pivoting often incurs a significant overhead and consequently a number of techniques have been proposed to try and limit the need for pivoting. In particular, numerically-aware ordering algorithms may be used, that is, orderings that depend not only on the sparsity pattern of A but also on the values of its (scaled) entries.

Current approaches identify large entries of A and symmetrically permute them onto the subdiagonal where they can be used as part of a 2×2 pivot. This is numerically effective, but the fill in the factor L and hence the runtime of the factorization and subsequent triangular solves may be significantly increased over a standard ordering if no pivoting is required.

We present a new algorithm that combines a matching-based approach with a numerically-aware nested dissection ordering. Numerical comparisons with current approaches for some tough symmetric indefinite problems are given.

Keywords: nested dissection, numerically aware ordering, sparse symmetric matrices, sparse matrix ordering, sparse direct methods.

AMS(MOS) subject classifications: 65F05, 65F30, 65F50

---

[1] Scientific Computing Department, STFC Rutherford Appleton Laboratory, Harwell Campus, Didcot OX11 0QX, UK.

Correspondence to: jennifer.scott@stfc.ac.uk
This work was support by EPSRC grant EP/M025179/1.

August 3, 2017

1 Introduction

In this paper, we are concerned with using a direct solver for the solution of large sparse linear systems

$$Ax = b,$$

where A is symmetric indefinite and non-singular. Sparse direct methods solve such systems by performing a factorization of A of the form

$$A = LDL^T,$$

where L is a unit lower triangular matrix and D is a block diagonal matrix with non-singular 1×1 and 2×2 blocks. In practice, a more general factorization of the form

$$SAS = PLD(PL)^T$$

is computed, where S is a diagonal scaling matrix and P is a permutation matrix that holds the elimination order. For efficiency in terms of both time and memory, it is essential to choose P to exploit the sparsity structure of A. The structure of L is the union of the structure of the permuted matrix P^TAP and new entries known as *fill*. The amount of fill is highly dependent on the choice of P.

It is well established that choosing P to minimize fill is an NP-complete problem [27]. Most sparse symmetric factorization algorithms heuristically determine P on the basis of the sparsity pattern of A before performing any numerical calculations, and two main classes of algorithm are commonly used. The first are those derived from the original Markowitz minimum degree heuristic, with the approximate minimum degree (AMD) algorithm of Amestoy et al. [1] currently being the most popular. The second class are based on nested dissection techniques, with the METIS package [23] being the best known implementation. This class of methods perform particularly well on very large problems (although they cost more than AMD orderings to compute). For positive-definite systems, the elimination order chosen during the analyse phase of the sparse direct solver using the sparsity pattern can be used unaltered by the numerical factorization. However, for indefinite systems, it can be necessary to modify the elimination order to maintain numerical stability. This is done by delaying the elimination of variables that could cause instability until later in the factorization when the associated pivot (that is, the 1×1 or 2×2 block used to eliminate one, respectively, two variables) can be safely used, and this is still the only provably stable approach. The scaling matrix S is normally chosen to try and reduce the number of these delayed pivots. A comparison of the effectiveness of common scalings is given in [17].

For the most numerically challenging indefinite systems, current scaling techniques alone are insufficient to keep the number of delayed pivots acceptably low [20]. In such cases, methods have been proposed to combine the fill-reducing ordering P and the scaling S so that a numerically-aware ordering is computed. Based on the original idea of Duff and Gilbert [6], for rows and columns with a small entry on the diagonal, Duff and Pralet [9] and Schenk and Gärtner [25] describe techniques for permuting large entries of A onto the subdiagonal using a matching-based ordering. These entries can be used to form stable 2×2 pivots and to thus reduce the number of delayed pivots. In the sparse solver package PARDISO [25], this approach is taken further, with no pivots being delayed beyond the current node in the elimination tree; instead, pivot candidates that are too small are modified by some prescribed perturbation. This is known as static pivoting; it generally requires the use of iterative refinement (or other iterative method) to recover accuracy (potentially adding to the solution time). This works well for many matrices, but is demonstrably unstable on a few very tough examples.

The survey paper of Hogg and Scott [20] discusses the above schemes and several others and concludes that at present matching-based ordering techniques are the most effective method for keeping the number of delayed pivots acceptably low.

Current state-of-the-art matching-based orderings use a matching as a preprocessing step before applying the ordering scheme of choice to a compressed matrix. Whilst limiting the pivot modifications needed during the factorization, the resulting ordering can lead to significantly more fill than if the ordering scheme was applied directly to A. This results in higher memory costs and greater factorization flops counts. The aim of this paper is to investigate ways of computing an elimination order that needs few modifications when applied to tough indefinite problems but that also does not lead to large amounts of extra fill. Duff and Pralet [9] consider a numerically-aware minimum degree algorithm but, as nested dissection is often the method of choice, our aim is to develop a numerically-aware nested dissection algorithm. This is the key contribution of this paper. We compare the effectiveness of the proposed algorithm in reducing the number of delayed pivots it produces with a traditional non-numerically aware ordering and with the unpublished scheme of Gupta that is used in the sparse direct solver WSMP [13].

The paper is laid out as follows. In Section 2, we give a high-level overview of nested dissection algorithms, whilst Section 3 provides the relevant background on pivoting techniques and Section 4 gives an overview of the existing matching-based ordering algorithm. With the background established, in Section 5 we go on to describe our new numerically-aware nested dissection ordering algorithm. In Section 6, we describe the algorithm used by Gupta within WSMP. Finally, we present numerical results in Section 7 and summarise our findings in Section 8.

2 Nested Dissection Overview

The nested dissection algorithm begins by associating with the symmetrically-structured matrix $A = \{a_{i,j}\}$ an undirected graph $\mathcal{G}(A)$ whose vertices represent rows (equivalently, columns) and whose edges represent non-zero entries of A. If there is an entry $a_{i,j}$ then there is an edge (i,j) from i to j. This is illustrated by the example in Figure 1. We will assume throughout our discussions that A is irreducible so that $\mathcal{G}(A)$ is connected (otherwise, the algorithms may be applied to each component in turn).

Nested dissection proceeds by identifying a small set of vertices $\mathcal{S}$ (known as a *separator*) that if removed separates the graph into two disjoint subgraphs described by the vertex subsets $\mathcal{B}$ and $\mathcal{W}$. The rows and columns belonging to $\mathcal{B}$ are ordered first, then those belonging to $\mathcal{W}$ and finally those in $\mathcal{S}$. The reordered matrix has the following form

$$\begin{pmatrix} A_{\mathcal{BB}} & 0 & A_{\mathcal{BS}} \\ 0 & A_{\mathcal{WW}} & A_{\mathcal{WS}} \\ A_{\mathcal{BS}}^T & A_{\mathcal{WS}}^T & A_{\mathcal{SS}} \end{pmatrix}.$$

This is illustrated for our example in Figure 2. Provided the variables are eliminated in the permuted order, no fill will occur within the zero blocks. If $|\mathcal{S}|$ is small and the sizes of $\mathcal{B}$ and $\mathcal{W}$ are well balanced, these zero blocks account for approximately half the possible entries in the matrix. The process can then be applied recursively to the submatrices $A_{\mathcal{BB}}$ and $A_{\mathcal{WW}}$ until the vertex subsets are of size less than some prescribed threshold. At this stage, a local ordering techniques (such as AMD) is generally more effective than nested dissection, and so a switch is made.

The performance and efficacy of nested dissection is highly dependent on the approach used to determine the separator $\mathcal{S}$ at each stage of the recursion. The original approach [10] and the algorithms described in the book by George and Liu [11] employed level set based methods. However, most current approaches use multilevel techniques that create a hierarchy of graphs, each representing the original graph, but with a smaller dimension. The smallest (that is, the coarsest) graph in the sequence is partitioned. This partition is propagated back through the sequence of graphs, while being periodically refined. In the mid-1990's, a number of efficient multilevel implementations were designed and developed, most notably Chaco from Hendrickson and Leland [15, 16], METIS from Karypis and Kumar [22, 23], and SCOTCH from Pellegrini [24]. In this paper, we use a nested dissection code that we developed to provide us with a research vehicle to facilitate testing new ideas. A description of this code, which incorporates both a level-set approach and a multilevel approach, together with numerical comparisons in terms of quality and ordering times with METIS, is given in the report by Ashcraft et al. [2]. However, the ideas presented in this paper will work with any approach for finding the separator.

3 Pivoting conditions

For indefinite systems, the elimination order chosen during the analyse phase of a direct solver provides a tentative pivot sequence. During the numerical factorization, potential pivots need to be checked for stability. The method used for this varies from solver to solver, but essentially they seek to avoid dividing a large off-diagonal entry by a small diagonal one, leading to growth in the entries of the factor L. If a pivot candidate fails the test for stability, it is delayed until later in the elimination order (we refer to such pivots as *delayed pivots*). By delaying the elimination of variable k, either an update from another

$$
\begin{array}{c}
1\\2\\3\\4\\5\\6\\7\\8\\9\\10\\11\\12\\13\\14\\15
\end{array}
\left(
\begin{array}{ccccccccccccccc}
x & x & x & & & & & & & & & & & & \\
x & x & x & x & x & & & & & & & & & & \\
x & x & x & & x & x & & & & & & & & & \\
 & x & & x & x & & x & x & & & & & & & \\
 & x & x & x & x & x & & x & x & & & & & & \\
 & & x & & x & x & & & x & & & & & & \\
 & & & x & & & x & x & & x & x & & & & \\
 & & & x & x & & x & x & x & & x & x & & & \\
 & & & & x & x & & x & x & & & x & & & \\
 & & & & & & x & & & x & x & & x & & \\
 & & & & & & x & x & & x & x & x & x & x & \\
 & & & & & & & x & x & & x & x & & x & \\
 & & & & & & & & & x & x & & x & x & x \\
 & & & & & & & & & & x & x & x & x & x \\
 & & & & & & & & & & & & x & x & x
\end{array}
\right)
$$

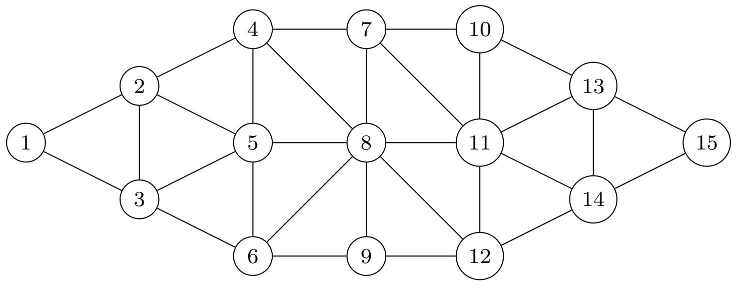
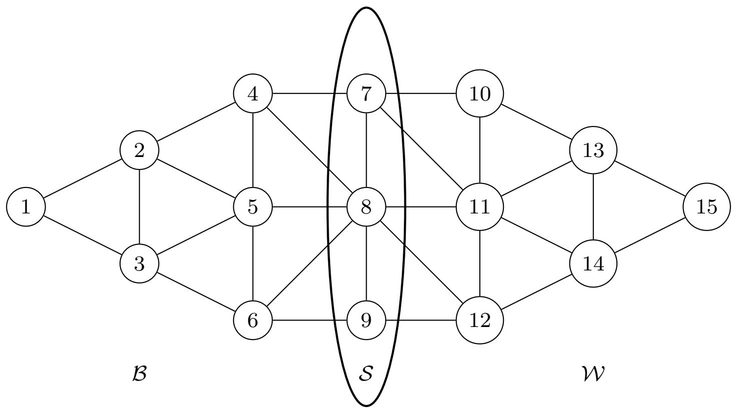


Figure 1: Correspondence between a symmetrically structured matrix and its graph $\mathcal{G}(A)$



$$
\begin{array}{c}
1\\2\\3\\4\\5\\6\\10\\11\\12\\13\\14\\15\\7\\8\\9
\end{array}
\left(
\begin{array}{cccccc|cccccc|ccc}
x & x & x & & & & & & & & & & & & \\
x & x & x & x & x & & & & & & & & & & \\
x & x & x & & x & x & & & & & & & & & \\
 & x & & x & x & & & & & & & & x & x & \\
 & x & x & x & x & x & & & & & & & & x & \\
 & & x & & x & x & & & & & & & & x & x \\
\hline
 & & & & & & x & x & & x & & & x & & \\
 & & & & & & x & x & x & x & x & & x & x & \\
 & & & & & & & x & x & & x & & & x & x \\
 & & & & & & x & x & & x & x & x & & & \\
 & & & & & & & x & x & x & x & x & & & \\
 & & & & & & & & & x & x & x & & & \\
\hline
 & & & x & & & x & x & & & & & x & x & \\
 & & & x & x & x & & x & x & & & & x & x & x \\
 & & & & & x & & & x & & & & & x & x
\end{array}
\right)
$$

Figure 2: Partitioned graph and corresponding reordered matrix

elimination will increase the magnitude of $a_{k,k}$, or column k will become adjacent to column $k+1$ with the property that $a_{k,k+1}$ is large and hence can be incorporated into a stable 2×2 pivot.

In our work, we will use the sufficient (but not necessary) conditions given by Duff et al. [7] for threshold partial pivoting to be stable. Let $A^{(k)}$ denote the Schur complement after columns $1, \dots, k-1$ of A have been eliminated, and let u_{num} be the pivot threshold. Then the stability conditions, which are employed in a number of direct solvers (including `MA57` and `HSL_MA97` [5, 19]), are:

- a 1×1 pivot on column k is stable if

$$\max_{i>k} |a^{(k)}_{i,k}| < u^{-1}_{num} |a^{(k)}_{k,k}| \tag{1}$$

- a 2×2 pivot on columns k and $k+1$ is stable if

$$\left| \begin{pmatrix} a^{(k)}_{k,k} & a^{(k)}_{k,k+1} \\ a^{(k)}_{k+1,k} & a^{(k)}_{k+1,k+1} \end{pmatrix}^{-1} \right| \begin{pmatrix} \max_{i>k+1} |a^{(k)}_{i,k}| \\ \max_{i>k+1} |a^{(k)}_{i,k+1}| \end{pmatrix} \le u^{-1}_{num} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \tag{2}$$

  where the modulus of the matrix is interpreted element-wise. Additionally, it is required that the pivot can be stably inverted.

Observe that these conditions imply that each entry in L is bounded in modulus by u^{-1}_{num} and that growth between $A^{(k)}$ and $A^{(k+1)}$ (and hence in D) is at most u^{-1}_{num}. The default setting for u_{num} is normally 0.01. In practice, columns with similar sparsity patterns are grouped into *supernodes* and pivots may be freely chosen in any order from within the supernode without altering the amount of fill. We are only concerned when we need to delay pivots to later supernodes.

4 Existing matching-based ordering algorithms

In this section, we recall the matching-based ordering algorithm and describe an existing proposal to combine a matching-based ordering with a minimum-degree based ordering.

4.1 Basic matching-based ordering

Matching-based ordering algorithms are built on the assumption that large entries in A remain large (in a relative sense) as the factorization progresses. Hence, large entries on the diagonal may be used as stable 1×1 pivots while symmetrically permuting large entries onto the subdiagonal before the factorization commences allows the formation of stable 2×2 pivots without the need to modify the selected elimination order. This assumption appears to be borne out in practice [20].

Matching algorithms work on the bipartite graph $\mathcal{G}_A = (V_r \cup V_c, E)$, where V_r and V_c are vertex sets, corresponding to the rows and columns of A, respectively, and $E = \{(i,j) | a_{i,j} \neq 0\}$ is the set of edges connecting the vertices in V_r and V_c. A subset $\mathcal{M} \subseteq E$ is called a *matching* if no two edges in $\mathcal{M}$ are incident to the same vertex. In matrix terms, a matching corresponds to a set of nonzero entries with no two belonging to the same row or column. $\mathcal{M}$ is a *symmetric matching* if $(i,j) \in \mathcal{M} \Rightarrow (j,i) \in \mathcal{M}$; otherwise, the matching is *unsymmetric*, which we denote as $\mathcal{M}_{unsym}$.

An unsymmetric matching $\mathcal{M}_{unsym}$ may be found using the Hungarian algorithm (as implemented, for example, in the widely used HSL package `MC64` [8]). In addition to the matching $\mathcal{M}_{unsym}$, the Hungarian algorithm also finds scaling matrices S_r and S_c such that the entries of the scaled matrix $S_r A S_c$ satisfy the following conditions:

$$\begin{aligned} |(S_r A S_c)_{i,j}| &= 1 \qquad (i,j) \in \mathcal{M}_{unsym}, \\ |(S_r A S_c)_{i,j}| &\le 1 \qquad (i,j) \notin \mathcal{M}_{unsym}. \end{aligned}$$

It can be shown that by taking $S = \sqrt{S_r S_c}$ (where the square root is interpreted element-wise), the above conditions also hold for the symmetrically scaled matrix SAS. Thus the matched pairs (i,j) give the location of the largest entries in the scaled matrix. These are exactly the entries that we want on the subdiagonal.

Algorithm 1 Symmetrize a matching

```
Mark all vertices as unvisited
Initialize $\mathcal{M} = \varnothing$
for each $(i, j) \in \mathcal{M}_{unsym}$ do
  if $i$ is unvisited and $j$ is unvisited then
    Add $(i, j)$ and $(j, i)$ to $\mathcal{M}$
    Mark $i$ and $j$ as visited
  end if
end for
```

However, as $\mathcal{M}_{unsym}$ is not symmetric, it may supply more than one possible j for each column i. To select a single j, we obtain a symmetric matching $\mathcal{M}$ using Algorithm 1. We observe that $\mathcal{M}$ may not be unique. Furthermore, some vertices in $\mathcal{M}_{unsym}$ may not be in $\mathcal{M}$. If $(i, j) \in \mathcal{M}$ then we refer to i and j as *partners*.

Given a symmetric matching, the entries (i, j) and (j, i) of A may be symmetrically permuted into 2×2 blocks on the diagonal. The graph of the permuted matrix may be compressed by replacing each pair of rows and columns corresponding to a 2×2 block by a single row and column. The sparsity pattern of the replacement row and column is the union of the patterns of the rows and columns it replaces. A traditional fill-reducing ordering (for example, nested dissection) algorithm is computed for the compressed matrix $A_{\mathcal{M}}$. Finally, the compression process is reversed to obtain an ordering for A. We summarise the matching-based algorithm as Algorithm 2 (see also [14, 26]).

Algorithm 2 Basic matching-based ordering

```
Perform a Hungarian matching on $A$ to obtain a matching $\mathcal{M}_{unsym}$ (and a symmetrized scaling $S$)
Symmetrize $\mathcal{M}_{unsym}$ to obtain $\mathcal{M}$
Using $\mathcal{M}$ compress $A$ to obtain $A_{\mathcal{M}}$
Compute a fill reducing ordering for $A_{\mathcal{M}}$
Uncompress the ordering of $A_{\mathcal{M}}$ to obtain an ordering for $A$
```

The main downside to this approach is that the rows and columns represented by each $(i, j) \in \mathcal{M}$ may have very different non-zero patterns. This can result in much worse fill than an ordering technique that does not take into account numerical values (see, for example [9]).

In the rest of our discussion, we always refer to the scaled matrix SAS. Thus, for convenience of notation, we shall here on refer to it as A.

4.2 Constrained orderings

Duff and Pralet [9] observe that partners (i, j) do not always need to be ordered together. Consider a matched pivot

$$\begin{pmatrix} a_{i,i} & a_{i,j} \\ a_{i,j} & a_{j,j} \end{pmatrix},$$

ordered such that $|a_{i,i}| \geq |a_{j,j}|$, and let θ be an absolute threshold. If $|a_{j,j}| > \theta$ then the pivot may be split and i and j eliminated as two 1×1 pivots in any order because both are large with respect to θ. Unfortunately, experiments show that for our tough indefinite problems (see Section 7) this happens sufficiently rarely that it offers little practical improvement.

A matched pivot may also be split if $|a_{i,i}| > \theta > |a_{j,j}|$ provided i is eliminated before j. Based on these observations, Duff and Pralet describe modifying their AMD implementation to respect these conditions to produce a *constrained ordering*. In addition, they propose a *relaxed constrained ordering* in which j may be eliminated before i so long as it has been updated by some large entry $a_{j,k}$ satisfying $|a_{j,k}| > drop$ for some given tolerance *drop*. In their tests, Duff and Pralet use $\theta = 10^{-2}$ and $drop = 0.9$. They report that, as expected, the basic matching-based approach produces the smallest number of delayed pivots but that the constrained and relaxed constrained orderings can significantly reduce the factorization time required by the serial symmetric indefinite direct solver `MA57` [5] as well as the total operations performed and the fill in L.

5 A new numerically-aware nested dissection algorithm

Our numerically-aware algorithm proceeds along similar lines to the relaxed constrained ordering of Duff and Pralet. However, we use relative rather than absolute criteria for defining large entries and we apply the approach to the more complicated case of nested dissection.

Given a threshold u_{ord}, we define entries of the (scaled) matrix A to be large as follows:

- a diagonal entry $a_{i,i}$ is large if
$$\max_{k\neq i} |a_{i,k}| < u_{ord}^{-1} |a_{i,i}| \tag{3}$$
- an off-diagonal entry $a_{i,j}$ is large if
$$\left| \begin{pmatrix} a_{i,i} & a_{i,j} \\ a_{j,i} & a_{j,j} \end{pmatrix}^{-1} \right| \begin{pmatrix} \max_{k\neq i,j} |a_{i,k}| \\ \max_{k\neq i,j} |a_{j,k}| \end{pmatrix} \leq u_{ord}^{-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \tag{4}$$
  where the modulus of the matrix is interpreted element-wise.

Entries that are not large are defined to be *small.* Note that these criteria correspond to the pivot tests (1) and (2) with $A^{(k)}$ set to A but the value u_{ord} used to determine large entries need not be the same as the threshold u_{num} that is used in the numerical factorization. We define the graph $\mathcal{G}_L(A)$ to be the pruned graph that contains only those edges that correspond to large entries of A. Note that $\mathcal{G}_L(A)$ may not be connected but this does not effect what follows.

We define a matching $\mathcal{M}$ to be *compatible* with a partition $(\mathcal{B}, \mathcal{W}, \mathcal{S})$ if:

1. For all $(i, j) \in \mathcal{M}$, both i and j are in the same subset; and
2. For each $i \notin \mathcal{S}$, the diagonal entry $a_{i,i}$ is large, or i has a partner j such that $a_{j,i}$ is large.

In terms of the ordering, this means that the submatrices associated with $\mathcal{B}$ and $\mathcal{W}$ are likely to have full numerical rank, and the matching suggests large entries that can be permuted to the subdiagonal. Note that vertices in the separator need not be matched: if $j \in \mathcal{S}$ is not matched, then all variables j corresponding to large entries $a_{i,j}$ must be in $A_{\mathcal{BS}}$ or $A_{\mathcal{WS}}$ and hence will be eliminated before i. Hence a pivot candidate $a_{j,j}^{(k)}$ in $A_{\mathcal{SS}}$ is likely to be large and accepted as a pivot (even if the diagonal entry $a_{j,j}$ was not large in A).

The algorithms described below take a (symmetric) initial matching $\mathcal{M}$, and a *candidate partition* $(\mathcal{B}, \mathcal{W}, \mathcal{S})$ and modifies them to obtain a compatible matching and partition. The initial matching can either be derived from symmetrization of a Hungarian matching as described in Section 4, or can be taken from a higher level of dissection in the recursive case. The candidate partition is obtained using standard techniques that do not consider numerical information (see for example [2]).

In order to describe our algorithms, we need to define the following. An *alternating path* for a matching $\mathcal{M}$ is a path whose edges are alternately in the matching and not in the matching. The matching can be modified using such a path to obtain a new matching by removing those edges from the path that are already in $\mathcal{M}$, and adding those edges that are not. The vertices that are part of the modified matching are those that were in the original matching, except that the vertices at each end of the path may have been added or removed. We say an edge (i, j) is *cut* by the separator $\mathcal{S}$ if $i \in \mathcal{S}$ and $j \notin \mathcal{S}$, or if $j \in \mathcal{S}$ and $i \notin \mathcal{S}$. We define an *$\mathcal{S}$-restricted alternating path* to be an alternating path on which no edges except the first and last are cut by the separator $\mathcal{S}$.

Algorithm 3 iterates over matched edges that are cut by the separator. Either an alternating path is found that makes the matching compatible, or one of the vertices is moved into $\mathcal{S}$. Note that, for efficiency, the length of the $\mathcal{S}$-restricted alternating path could be limited.

Consider the matrix shown in Figure 3 (with the same partition as in our previous example of Section 2) and an initial matching $\mathcal{M} = \{(6, 9), (7, 11), (8, 12), (13, 14)\}$. The associated graph is shown as Figure 4(a). Note that the partition and matching are not compatible because of the cut edges $(6, 9), (7, 11)$ and $(8, 12)$. First we consider the cut edge $(8, 12)$; a suitable alternating path is shown in Figure 4(b). Note how condition (a) of Algorithm 3 on the alternating path ensures that when we using it to modify the matching, we eliminate two cut edges, resulting in the new modified matching $\mathcal{M} = \{(11, 13), (12, 14)\}$. An alternative alternating path would be $8 \to 12 \to 14 \to 13 \to 15$, which uses

Algorithm 3 Make a partition $(\mathcal{B}, \mathcal{W}, \mathcal{S})$ and matching $\mathcal{M}$ compatible

```
for each cut edge $(i, j) \in \mathcal{M}$ do
    Attempt to find an $\mathcal{S}$-restricted alternating path in $\mathcal{G}_L(A)$ from $i$ to some vertex $q$ that satisfies the
    following conditions:
     1.   The first edge is $(i, j)$; and
     2.   The last edge $(p, q)$ satisfies either:
          (a)   $q \in \mathcal{S}$ and $(p, q) \in \mathcal{M}$; or
          (b)   $q \notin \mathcal{S}$ and $(p, q) \notin \mathcal{M}$.
    If such a path exists, use it to modify the matching $\mathcal{M}$.
    If no such path exists, move $j$ into $\mathcal{S}$.
end for
```

$$\begin{array}{r} 1\\2\\3\\4\\5\\6\\7\\8\\9\\10\\11\\12\\13\\14\\15 \end{array}
\left(\begin{array}{ccccccccccccccc}
1.0 & 0.2 & 0.5 & & & & & & & & & & & & \\
0.2 & 1.0 & 0.2 & 0.3 & 0.5 & & & & & & & & & & \\
0.5 & 0.2 & 1.0 & & 0.2 & \mathbf{0.01} & & & & & & & & & \\
 & 0.3 & & 1.0 & 0.1 & & 0.2 & 0.7 & & & & & & & \\
 & 0.5 & 0.2 & 0.1 & 1.0 & \mathbf{0.01} & & 0.4 & & & & & & & \\
 & & \mathbf{0.01} & & \mathbf{0.01} & \mathbf{0.0} & & 0.1 & 1.0 & & & & & & \\
 & & & 0.2 & & & 0.8 & 0.5 & & 1.0 & 0.8 & & & & \\
 & & & 0.7 & 0.4 & 0.1 & 0.5 & 1.0 & 0.3 & & 0.4 & 0.8 & & & \\
 & & & & & 1.0 & & 0.3 & 1.0 & & & 0.5 & & & \\
 & & & & & & 1.0 & & & 1.0 & 0.3 & & \mathbf{0.01} & & \\
 & & & & & & 0.8 & 0.4 & & 0.3 & \mathbf{0.0} & \mathbf{0.01} & 1.0 & 0.8 & \\
 & & & & & & & 0.8 & 0.5 & & \mathbf{0.01} & \mathbf{0.0} & & 1.0 & \\
 & & & & & & & & & \mathbf{0.01} & 1.0 & & \mathbf{0.0} & 0.8 & 0.8 \\
 & & & & & & & & & & 0.8 & 1.0 & 0.8 & \mathbf{0.0} & 0.2 \\
 & & & & & & & & & & & & 0.8 & 0.2 & 1.0
\end{array}\right)$$

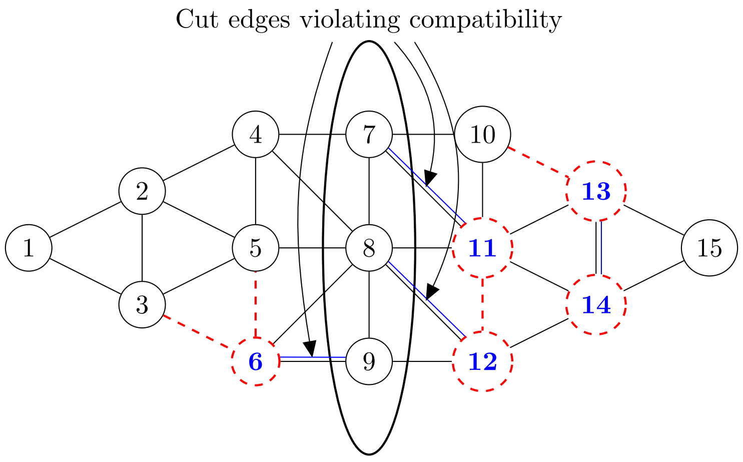
Figure 3: Example matrix with values, small edges highlighted in blue/bold.

the condition (b) for the final edge; in this case only one cut edge would be removed and an additional iteration would be required to remove $(7, 11)$. After applying the shown alternating path we are left with the cut edge $(6, 9)$. As there are no $\mathcal{S}$-restricted alternating paths, we must move 6 into the separator. This results in our final compatible partition and matching shown in Figure 4(c).

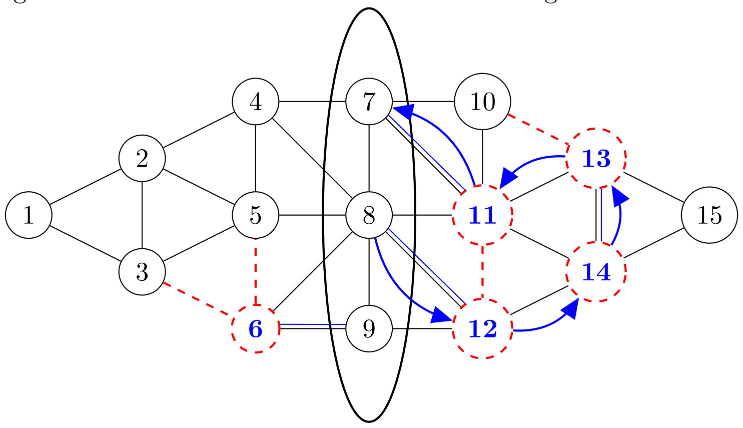
Having achieved compatibility, the separator may contain vertices that can be safely moved from $\mathcal{S}$ to subset $\mathcal{B}$ or $\mathcal{W}$. In our example, vertex 9 can be moved from $\mathcal{S}$ to $\mathcal{W}$ without violating the compatibility conditions. Moving such vertices is known as *trimming*; our trimming algorithm is listed as Algorithm 4. Let $\mathcal{S}_{ignore}$ be a subset of $\mathcal{S}$ containing vertices that cannot be moved into $\mathcal{B}$ or $\mathcal{W}$: this is initialised to the empty set. At each iteration, Algorithm 4 divides $\mathcal{S} \setminus \mathcal{S}_{ignore}$ into disjoint adjacency sets: $\mathcal{S}_{\mathcal{B}}$ for vertices that are adjacent to vertices in $\mathcal{B}$, $\mathcal{S}_{\mathcal{W}}$ for vertices that are adjacent to vertices in $\mathcal{W}$, $\mathcal{S}_{both}$ for vertices that are adjacent to vertices in both $\mathcal{B}$ and $\mathcal{W}$, and $\mathcal{S}_0$ for vertices that are fully internal to the separator. Observe that (ignoring compatibility) we can move a vertex in $\mathcal{S}_{\mathcal{B}}$ into $\mathcal{B}$, a vertex in $\mathcal{S}_{\mathcal{W}}$ into $\mathcal{W}$, or a vertex in $\mathcal{S}_0$ into either without losing the property that $\mathcal{S}$ is a separator. However, doing so may cause other vertices to become adjacent to $\mathcal{B}$ or $\mathcal{W}$ when they were not before, so the adjacency sets must be updated after each move. It is sufficient to only consider those vertices adjacent to the one moved.

Let $\hat{\mathcal{S}} = \mathcal{S}_0 \cup \mathcal{S}_{\mathcal{B}} \cup \mathcal{S}_{\mathcal{W}}$. To maintain compatibility, we can only move a vertex $i \in \hat{\mathcal{S}}$ into $\mathcal{B}$ or $\mathcal{W}$ if either $a_{i,i}$ is large, or we can find a partner to match it with. For simplicity, we first consider a partner $j \in \hat{\mathcal{S}}$ that can move to the same partition. If no such j exists, we then look for an $\mathcal{S}$-constrained alternating path that pairs i either with an unmatched vertex in its destination set, or with another vertex in $\hat{\mathcal{S}}$. If there is no such path, then we cannot move i into that subset and instead it moves to $\mathcal{S}_{ignore}$ and another i is chosen.

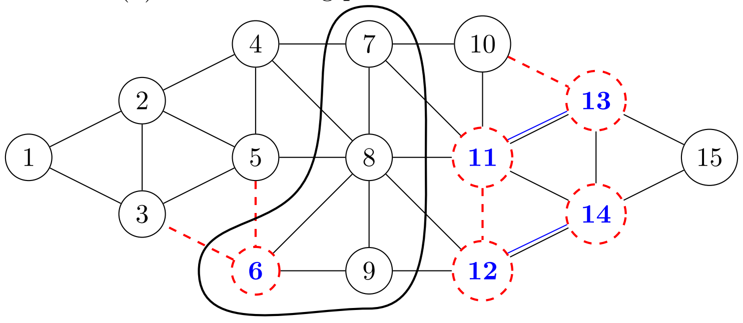
Often there is a choice of vertex to move. We prefer to avoid moving vertices in $\mathcal{S}_0$ if possible because after the move this normally results in additional vertices being reclassified into $\mathcal{S}_{both}$; otherwise, we only consider whether we want to move a vertex into $\mathcal{B}$ or $\mathcal{W}$. We select a candidate $i_{\mathcal{B}}$ that can move to $\mathcal{B}$ and create a set of move candidates $\mathcal{C}_{\mathcal{B}}$ containing only $i_{\mathcal{B}}$. We then select a candidate $i_{\mathcal{W}}$ that can



(a) Initial graph. Vertices in $\mathcal{B} \cup \mathcal{W}$ with small diagonal are shown as dashed circles with text in bold/blue. Edges in $\mathcal{M}$ are shown as double lines. Small edges are shown as dashed lines.



(b) An alternating path from vertex 8 to 7.



(c) After applying alternating path and moving vertex 6 into separator.

Figure 4: Application of Algorithm 3 to make a partition $(\mathcal{B}, \mathcal{W}, \mathcal{S})$ and matching $\mathcal{M}$ compatible for the matrix in Figure 3.

move to $\mathcal{W}$ and create a set of move candidates $\mathcal{C}_{\mathcal{W}}$ containing only $i_{\mathcal{W}}$. We determine whether we also need to move a partner to maintain compatibility and add these partners to the candidate move sets. We then calculate which move would provide a more balanced partition (recall that the more balanced these subsets are, the more we restrict fill). In Algorithm 4, the cost function $c(\cdot,\cdot)$. denotes our measure of balance (lower is better). Further details of the cost function are given in [2].

Algorithm 4 Trim separator whilst maintaining compatibility

```
Initialise $\mathcal{S}_{ignore} = \varnothing$.
Partition vertices in $\mathcal{S} \setminus \mathcal{S}_{ignore}$ into the sets $\mathcal{S}_0, \mathcal{S}_\mathcal{B}, \mathcal{S}_\mathcal{W}$ and $\mathcal{S}_{both}$ as follows:

        $i \in \mathcal{S}_0$      if  $j \in \mathcal{S}$                      $\forall$ edges $(i,j)$
        $i \in \mathcal{S}_\mathcal{B}$      if  $j \in \mathcal{S} \cup \mathcal{B}$     $\forall$ edges $(i,j)$   and $\exists$ an edge $(i,j)$ such that $j \in \mathcal{B}$
        $i \in \mathcal{S}_\mathcal{W}$      if  $j \in \mathcal{S} \cup \mathcal{W}$    $\forall$ edges $(i,j)$   and $\exists$ an edge $(i,j)$ such that $j \in \mathcal{W}$
        $i \in \mathcal{S}_{both}$  otherwise.

Set $\hat{\mathcal{S}} = \mathcal{S}_0 \cup \mathcal{S}_\mathcal{B} \cup \mathcal{S}_\mathcal{W}$.
while $\hat{\mathcal{S}}$ is non-empty do
  Initialise candidate vertex sets $\mathcal{C}_\mathcal{B} = \varnothing$ and $\mathcal{C}_\mathcal{W} = \varnothing$.
  Initialise candidate alternating paths $\mathcal{P}_\mathcal{B} = \varnothing$ and $\mathcal{P}_\mathcal{W} = \varnothing$.
  if $\exists\ i_\mathcal{B} \in \mathcal{S}_\mathcal{B}$, set $\mathcal{C}_\mathcal{B} = \{i_\mathcal{B}\}$   else if $\exists\ i_\mathcal{B} \in \mathcal{S}_0$, set $\mathcal{C}_\mathcal{B} = \{i_\mathcal{B}\}$.
  if $\exists\ i_\mathcal{W} \in \mathcal{S}_\mathcal{W}$, set $\mathcal{C}_\mathcal{W} = \{i_\mathcal{W}\}$   else if $\exists\ i_\mathcal{W} \in \mathcal{S}_0$, set $\mathcal{C}_\mathcal{W} = \{i_\mathcal{W}\}$.
  if $\exists\ i_\mathcal{B} \in \mathcal{C}_\mathcal{B}$ and $a_{i_\mathcal{B},i_\mathcal{B}}$ is small then
    if $\exists\ j_\mathcal{B} \in \mathcal{S}_0 \cup \mathcal{S}_\mathcal{B}$ such that $a_{j_\mathcal{B},i_\mathcal{B}}$ is large, set $\mathcal{C}_\mathcal{B} = \mathcal{C}_\mathcal{B} \cup \{j_\mathcal{B}\}$.
    else if $\exists$ a $\mathcal{S}$-restricted alternating path $\mathcal{P}$ in $\mathcal{G}_L(A)$ from $i_\mathcal{B}$ to some unmatched vertex
    $q \in \mathcal{B} \cup \mathcal{S}_\mathcal{B} \cup \mathcal{S}_0$, set $\mathcal{P}_\mathcal{B} = \mathcal{P}$, and if $q \in \hat{\mathcal{S}}$, set $\mathcal{C}_\mathcal{B} = \mathcal{C}_\mathcal{B} \cup \{q\}$.
    else if $i_\mathcal{B} \in \mathcal{S}_0$, move $i_\mathcal{B}$ to $\mathcal{S}_\mathcal{W}$ and go to next iteration.
    else move $i_\mathcal{B}$ from $\mathcal{S}_\mathcal{B}$ to $\mathcal{S}_{ignore}$ and go to next iteration.
  end if
  if $\exists\ i_\mathcal{W} \in \mathcal{C}_\mathcal{W}$ and $a_{i_\mathcal{W},i_\mathcal{W}}$ is small then
    if $\exists\ j_\mathcal{W} \in \mathcal{S}_0 \cup \mathcal{S}_\mathcal{W}$ such that $a_{j_\mathcal{W},i_\mathcal{W}}$ is large, set $\mathcal{C}_\mathcal{W} = \mathcal{C}_\mathcal{W} \cup \{j_\mathcal{W}\}$.
    else if $\exists$ a $\mathcal{S}$-restricted alternating path $\mathcal{P}$ in $\mathcal{G}_L(A)$ from $i_\mathcal{W}$ to some unmatched vertex
    $q \in \mathcal{W} \cup \mathcal{S}_\mathcal{W} \cup \mathcal{S}_0$, set $\mathcal{P}_\mathcal{W} = \mathcal{P}$, and if $q \in \hat{\mathcal{S}}$, set $\mathcal{C}_\mathcal{W} = \mathcal{C}_\mathcal{W} \cup \{q\}$.
    else if $i_\mathcal{W} \in \mathcal{S}_0$, move $i_\mathcal{W}$ to $\mathcal{S}_\mathcal{B}$ and go to next iteration.
    else move $i_\mathcal{W}$ from $\mathcal{S}_\mathcal{W}$ to $\mathcal{S}_{ignore}$ and go to next iteration.
  end if
  if $c(\mathcal{C}_\mathcal{B}, \mathcal{B}) < c(\mathcal{C}_\mathcal{W}, \mathcal{W})$ then
    Move $\mathcal{C}_\mathcal{B}$ into $\mathcal{B}$.
    Use alternating path $\mathcal{P}_\mathcal{B}$ to modify $\mathcal{M}$.
  else
    Move $\mathcal{C}_\mathcal{W}$ into $\mathcal{W}$.
    Use alternating path $\mathcal{P}_\mathcal{W}$ to modify $\mathcal{M}$.
  end if
  Update the sets $\mathcal{S}_0, \mathcal{S}_\mathcal{B}, \mathcal{S}_\mathcal{W}$ and $\mathcal{S}_{both}$. Update $\hat{\mathcal{S}}$.
end while
```

It is sometimes beneficial to explicitly order dense rows at the end of the elimination order before using nested dissection. This process may result in pairs given by our initial matching to become separated. However, we can treat the set of dense rows as a separator $\mathcal{S}$ that generates a partition where one of $\mathcal{B}$ or $\mathcal{W}$ is potentially empty. We can then apply Algorithm 3 to restore compatibility by either altering $\mathcal{M}$ or ordering some additional rows with their partner dense rows.

We also observe that we must be careful to ensure that the matched vertices resulting from the above are adjacent in the final ordering. In our implementation, we use AMD to order the partition at the lowest level of the nested dissection recursion and use the compression methodology of Section 4 at that stage.

We do not want the analyse phase to split pairs that have very different sparsity patterns. To prevent this, we ensure that for each pair (i, j):

- If either i or j has a large diagonal entry, it is ordered first to obtain a pivot of the form
$$\begin{pmatrix} x & x \\ x & \epsilon \end{pmatrix},$$
where ϵ denotes a small (or zero) entry. If this pivot is split, it can be safely handled as two 1×1 pivots.
- Otherwise, we have a pivot of the form
$$\begin{pmatrix} \epsilon & x \\ x & \epsilon \end{pmatrix}.$$
In this case, we order the densest column first, in the hope that the filled-in pattern of the second column will be sufficiently similar so that i and j are put into same supernode.

6 Gupta algorithm

The sparse solver WSMP [13] does not employ unsymmetric matchings. Instead, after appropriate scaling of A (which is not necessarily matching-based), pairs (i, j) are identified as follows [12]. First define

$$\mathcal{E}_{\text{diff}}(i, j) \quad = \quad \{k : a_{k,i} \neq 0, a_{k,j} = 0\} \cup \{k : a_{k,i} = 0, a_{k,j} \neq 0\}.$$

That is, $\mathcal{E}_{\text{diff}}(i, j)$ is the set of all non-zero entries that occur in either column i or column j, but not in both. Initialise $\mathcal{M}$ to $\emptyset$ and let i be the first column with a small diagonal entry. A partner j is chosen as follows:

$$j = \arg\max_{j} \frac{|a_{j,i}|}{|\mathcal{E}_{\text{diff}}(i, j))|}.$$

This hopefully corresponds to a large off diagonal entry $a_{j,i}$ and a column j that has a similar sparsity pattern to column i. Edge (i, j) is added to $\mathcal{M}$. When the next column with a small diagonal entry is considered, columns i and j are omitted from the search for a partner, and so on.

The criterion used in WSMP is a simple absolute threshold to detect whether the diagonal is effectively zero; specifically, the test is

$$|a_{i,i}| < small,$$

with *small* set to 10^{-18}. Having constructed $\mathcal{M}$, the matrix is compressed and then ordered using a nested dissection algorithm that employs weights to reflect the compression. We do not do this in our implementation, as we found it offers no significant advantage over the unweighted version [20].

7 Numerical results

In this section, we present numerical results for the 23 numerically challenging problems that were identified by Hogg and Scott [20] as requiring a matching-based ordering to limit the delayed pivots; the problems are listed in Table 1. Here we report the number of entries in L and flop count to compute L returned by the analyse phase of `HSL_MA97` (v2.3.0) using the nested dissection code described in [2] to compute the elimination order. We would like our new numerically-aware nested dissection algorithm to produce orderings such that the subsequent factorization achieves statistics that are close to these with few delayed pivots. With the exception of the choice of ordering, `HSL_MA97` is run with its default settings. In particular, unless stated otherwise, the partial pivoting threshold is $u_{num} = 0.01$. In all tests, the symmetric scaling from the Hungarian algorithm discussed in Section 4.1 is used. As it is beyond the scope of this paper to develop an efficient implementation of our new ordering algorithm (it is highly non trivial to do this), timing results are omitted. In all our tests, we generate the right-hand side vector b by setting x to be the vector of all 1s. We check that the scaled backward error is $\approx 10^{-14}$ and employ iterative refinement if necessary.

For each ordering, we are primarily concerned with two statistics reported by `HSL_MA97`:

Table 1: Numerically challenging symmetric indefinite problems. n and $nz(A)$ denote the dimension of and the number of non-zero entries in A. $nz(L)$ and $fflop$ are the number of entries in L and the flop count to compute L returned by the analyse phase of HSL_MA97 with nested dissection ordering.

	Problem	n	$nz(A)$	$nz(L)$	$fflop$	Description/Application
1.	TSOPF/TSOPF_FS_b39_c7	28216	730080	1.66×10^6	1.06×10^8	Optimal power flow
2.	QY/case39	40216	1042160	2.39×10^6	1.53×10^8	Optimal power flow
3.	TSOPF/TSOPF_FS_b39_c19	76216	1977600	4.51×10^6	2.87×10^8	Optimal power flow
4.	TSOPF/TSOPF_FS_b39_c30	120216	3121160	7.11×10^6	4.53×10^8	Optimal power flow
5.	GHS_indef/stokes128	49666	558594	3.43×10^6	5.78×10^8	Finite Element: Stokes problem
6.	TSOPF/TSOPF_FS_b162_c3	30798	1801300	4.09×10^6	6.16×10^8	Optimal power flow
7.	GHS_indef/darcy003	389874	2101242	8.23×10^6	6.41×10^8	Finite Element: Darcy's equation
8.	GHS_indef/cont-201	80595	438795	4.54×10^6	7.37×10^8	Optimization: Convex QP
9.	TSOPF/TSOPF_FS_b162_c4	40798	2398220	5.45×10^6	8.32×10^8	Optimal power flow
10.	GHS_indef/ncvxqp1	12111	73963	1.94×10^6	1.00×10^9	Optimization: Non-convex QP
11.	GHS_indef/cont-300	180895	988195	1.15×10^7	2.62×10^9	Optimization: Convex QP
12.	GHS_indef/d_pretok	182730	1641672	3.01×10^7	2.64×10^9	Finite Element: Underground mine
13.	GHS_indef/cvxqp3	17500	122462	3.86×10^6	2.83×10^9	Optimization: Convex QP
14.	TSOPF/TSOPF_FS_b300	29214	4400122	9.86×10^6	3.85×10^9	Optimal power flow
15.	TSOPF/TSOPF_FS_b300_c1	29214	4400122	9.86×10^6	3.85×10^9	Optimal power flow
16.	GHS_indef/bratu3d	27792	173796	7.38×10^6	5.42×10^9	Optimization
17.	TSOPF/TSOPF_FS_b300_c2	56814	8767466	1.96×10^7	7.58×10^9	Optimal power flow
18.	TSOPF/TSOPF_FS_b300_c3	84414	13135930	2.96×10^6	1.16×10^{10}	Optimal power flow
19.	GHS_indef/ncvxqp5	62500	424966	1.35×10^7	1.19×10^{10}	Optimization: Non-convex QP
20.	TSOPF/TSOPF_FS_b162_c1	10798	608540	8.27×10^6	1.62×10^{10}	Optimal power flow
21.	GHS_indef/turon_m	189924	1690876	2.68×10^7	2.02×10^{10}	Finite Element: underground mine
22.	GHS_indef/ncvxqp3	75000	499964	2.13×10^7	2.64×10^{10}	Optimization: Non-convex QP
23.	GHS_indef/ncvxqp7	87500	574962	3.19×10^7	5.77×10^{10}	Optimization: Non-convex QP

ndelay is the number of delayed pivots (the total number of times a pivot is passed up a node in the assembly tree: a single pivot passed several steps up the tree counts multiple times). This demonstrates how far we have deviated during the numerical factorization from the elimination order chosen by the analyse phase. In addition to limiting the flop count and memory footprint, keeping this statistic low is important for effective load balancing in parallel codes.

fflop is the number of floating-point operations (flops) performed during the numerical factorization (including those arising from delayed pivots). Keeping this statistic low is important as the factorization is (normally) compute-bound.

The number of entries in the factors may also be of interest, particularly where the solution phase of the solver is applied repeatedly. We note that this statistic has similar behaviour to *fflop* and thus we do not report on it explicitly.

For the sake of convenience, we will refer to the ordering methods as follows:

nd: Ordering computed using our nested dissection code.

match-nd: Basic matching-based ordering computed using Algorithm 2 of Section 4.

na-nd: Ordering computed using our numerically-aware nested dissection method described in Section 5.

gupta-zero: Ordering computed using our implementation of the Gupta heuristic, using an absolute comparison with $small = 10^{-18}$ to determine if a diagonal entry is zero.

gupta-small: Ordering computed using our implementation of the Gupta heuristic, using the test (3) with $u_{ord} = 0.4$ to determine small diagonal entries.

As noted at the end of Section 4.1, in our experiments A is always prescaled (this includes when the Gupta heuristic is employed).

7.1 Choice of $\mathbf{u_{ord}}$ for na-nd

Table 2 presents results for our numerically-aware nested dissection algorithm (na-nd) run using a range of values of the parameter u_{ord} that controls our definition of large entries (3). In general, as u_{ord} increases, the number of delayed pivots (*ndelay*) reduces. There are two underlying effects that explain the behaviour with respect to the number of flops. As u_{ord} increases, the constraints on the ordering become more severe, resulting in the analyse phase predicting higher fill and flop count. However, a large number of delayed pivots in the subsequent factorization phase can generate significant additional fill and flops, and thus reducing *ndelay* can also reduce fill and flops.

Table 2: *ndelay* and *fflop* for a range of values u_{ord}. The best results are in bold and those that are more than 20% worse than the best are in italics.

Problem	*ndelay*			*fflop*		
$u_{ord} =$	0.01	0.1	0.4	0.01	0.1	0.4
TSOPF/TSOPF_FS_b39_c7	994	**990**	**990**	$\mathbf{1.32\times10^{8}}$	1.33×10^{8}	1.33×10^{8}
QY/case39	1771	**1767**	**1767**	$\mathbf{1.99\times10^{8}}$	2.00×10^{8}	2.01×10^{8}
TSOPF/TSOPF_FS_b39_c19	2663	**2659**	**2659**	$\mathbf{4.15\times10^{8}}$	4.17×10^{8}	4.20×10^{8}
TSOPF/TSOPF_FS_b39_c30	4202	**4198**	**4198**	$\mathbf{8.06\times10^{8}}$	8.11×10^{8}	8.18×10^{8}
GHS_indef/stokes128	**3424**	**3424**	3426	$\mathbf{6.01\times10^{8}}$	$\mathbf{6.01\times10^{8}}$	$\mathbf{6.01\times10^{8}}$
TSOPF/TSOPF_FS_b162_c3	757	**727**	776	7.95×10^{8}	$\mathbf{7.94\times10^{8}}$	8.07×10^{8}
GHS_indef/darcy003	15675	15675	**15320**	$\mathbf{6.88\times10^{8}}$	$\mathbf{6.88\times10^{8}}$	$\mathbf{6.88\times10^{8}}$
GHS_indef/cont-201	*3096*	*9680*	**446**	$\mathit{9.23\times10^{8}}$	$\mathit{1.29\times10^{9}}$	$\mathbf{7.37\times10^{8}}$
TSOPF/TSOPF_FS_b162_c4	1055	**1025**	1029	$\mathbf{1.14\times10^{9}}$	$\mathbf{1.14\times10^{9}}$	$\mathbf{1.14\times10^{9}}$
GHS_indef/ncvxqp1	*234*	*238*	**40**	$\mathbf{1.48\times10^{9}}$	1.50×10^{9}	$\mathit{1.88\times10^{9}}$
GHS_indef/cont-300	*6584*	*23899*	**977**	$\mathit{3.08\times10^{9}}$	$\mathit{4.12\times10^{9}}$	$\mathbf{2.55\times10^{9}}$
GHS_indef/d_pretok	*7548*	*7064*	**2363**	$\mathbf{2.69\times10^{10}}$	2.71×10^{10}	$\mathit{3.23\times10^{10}}$
GHS_indef/cvxqp3	*894*	*772*	**525**	5.75×10^{9}	6.01×10^{9}	$\mathbf{5.04\times10^{9}}$
TSOPF/TSOPF_FS_b300	*427*	228	**227**	$\mathbf{5.01\times10^{9}}$	5.04×10^{9}	5.04×10^{9}
TSOPF/TSOPF_FS_b300_c1	*823*	*853*	**452**	$\mathbf{4.91\times10^{9}}$	$\mathit{6.02\times10^{9}}$	5.61×10^{9}
GHS_indef/bratu3d	*9692*	*9692*	**341**	6.29×10^{9}	6.29×10^{9}	$\mathbf{5.44\times10^{9}}$
TSOPF/TSOPF_FS_b300_c2	*1058*	**613**	*813*	1.05×10^{10}	$\mathbf{9.71\times10^{9}}$	9.98×10^{9}
TSOPF/TSOPF_FS_b300_c3	1597	**1561**	1648	1.59×10^{10}	$\mathbf{1.49\times10^{10}}$	1.50×10^{10}
GHS_indef/ncvxqp5	*962*	*468*	**136**	$\mathbf{1.21\times10^{10}}$	1.24×10^{10}	$\mathit{1.84\times10^{10}}$
TSOPF/TSOPF_FS_b162_c1	*514*	194	**190**	$\mathbf{3.60\times10^{10}}$	3.63×10^{10}	3.64×10^{10}
GHS_indef/turon_m	*8189*	*8086*	**3555**	$\mathbf{2.04\times10^{10}}$	$\mathbf{2.04\times10^{10}}$	2.09×10^{10}
GHS_indef/ncvxqp3	*1620*	**1317**	1397	3.45×10^{10}	$\mathbf{3.30\times10^{10}}$	3.87×10^{10}
GHS_indef/ncvxqp7	*7087*	*13590*	**5615**	$\mathbf{1.01\times10^{11}}$	1.07×10^{11}	1.18×10^{11}

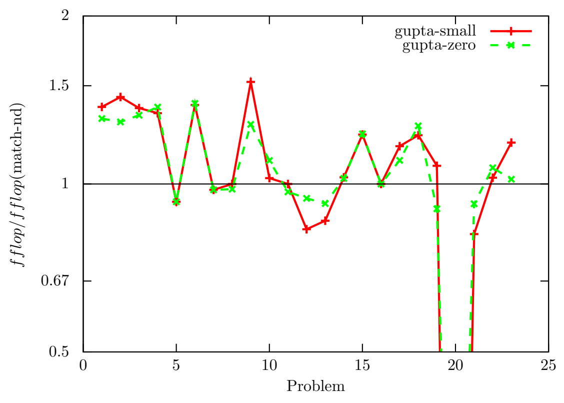
From further experiments using a wider range of values for u_{ord}, we observed that for each problem the number of delayed pivots plateaus once u_{ord} is some critical value (which is problem dependent). For $u_{ord} \geq 0.5$, we observed a significant deterioration in the quality of the ordering because of less flexibility in choosing matching. This was most apparent in a large increase in the flop count (in many cases more than twice those required for $u_{ord} = 0.4$), but for some problems the number of delayed pivots also increased. Based on our experiments, we use $u_{ord} = 0.4$ in the remainder of our tests as it provides a good balance between minimising *ndelay* and *fflop*.

Note that we also experimented with employing an absolute rather than a relative method for determining large entries using the values and meaning of θ and *drop* as proposed by Duff and Pralet [9]. We found that, even though we prescaled the matrix, the relative scheme almost always gave better results, especially with regard to the flop count.

7.2 Variants of the basic matched ordering algorithm

Figure 5 presents a comparison of the factorization phase flop counts for the gupta-small and gupta-zero orderings, normalised against the basic matching-based ordering (match-nd). Here, points above the horizontal line 1 represent worse performance than match-nd, whilst points below the line indicate better performance. The numbers of delayed pivots are reported in Table 3. We see that there is generally little to choose between the relative gupta-small and absolute gupta-zero orderings, with both often requiring 20-40% more flops than match-nd as well as leading to a higher number of delayed pivots than match-nd. In particular, for problem 23 (GHS_indef/ncvxqp7), the gupta-small ordering

Figure 5: A comparison of the factorization flop counts for the gupta-small and gupta-zero orderings, normalised against the match-nd ordering ($u_{ord} = 0.4$).



gives almost 175,000 delayed pivots. Problem 20 (TSOPF/TSOPF_FS_b162_c1) represents an anomaly: both gupta-small and gupta-zero do much better than match-nd. Further investigation reveals this is an artifact of the nested dissection ordering software choosing a significantly worse separator in this case.

7.3 Comparison of ND methods

Figure 6 compares the factorization flop counts of the nd and na-nd orderings, normalised against match-nd. The numerically-aware ordering (na-nd) generally has the lowest factorization flop count and is never significantly worse than match-nd. Results for the standard nested dissection algorithm (nd) are included for comparison; they confirm that it can perform significantly less well than either of the numerically-aware algorithms, although for more than half our examples, its has a lower flop count than match-nd.

Figure 6: A comparison of the factorization flop counts for the nd and na-nd orderings normalised against the match-nd ordering.

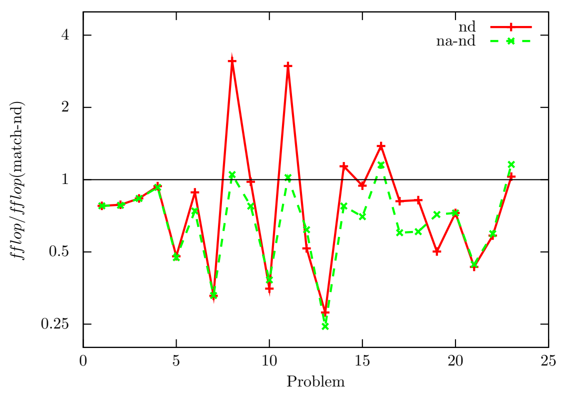


Table 3 reports the number of delayed pivots. We see that, with the exception of problem 7 (GHS_indef/darcy003), na-nd significantly decreases the number of delayed pivots compared to standard nested dissection (nd), although for some problems (including GHS_indef/stokes128 and GHS_indef/d_pretok), the number of delayed pivots for na-nd can be greater than for match-nd. A closer look at problem 7 shows that each delay is very local; on average each delayed pivot moves only 1.7 nodes up the tree.

Table 4 gives the same information, but with a much smaller partial pivoting threshold ($u_{num} = 10^{-8}$). Comparison with Table 3 shows that using a smaller threshold can reduce the number of delayed pivots but often has little effect. However, using a small value of u_{num} may reduce stability. We can assess this by looking at the number of steps of iterative refinement that are required to obtain a scaled backwards

Table 3: Number of delayed pivots for various orderings with $u_{ord} = 0.4, u_{num} = 0.01$.

Problem	nd	match-nd	na-nd	gupta-small	gupta-zero
TSOPF/TSOPF_FS_b39_c7	5367	1529	990	4668	2532
QY/case39	8180	2375	1767	12255	3457
TSOPF/TSOPF_FS_b39_c19	14457	4151	2659	17375	7334
TSOPF/TSOPF_FS_b39_c30	22768	6550	4198	18678	12098
GHS_indef/stokes128	8312	8	3426	21	21
TSOPF/TSOPF_FS_b162_c3	11477	1365	776	6565	8141
GHS_indef/darcy003	50042	79	15320	492	495
GHS_indef/cont-201	64186	0	446	0	1
TSOPF/TSOPF_FS_b162_c4	19507	1885	1029	8269	11883
GHS_indef/ncvxqp1	13640	27	40	7214	4455
GHS_indef/cont-300	130615	0	977	0	1
GHS_indef/d_pretok	24809	106	2363	160	2859
GHS_indef/cvxqp3	36775	130	525	16869	4127
TSOPF/TSOPF_FS_b300	14541	1265	227	877	1115
TSOPF/TSOPF_FS_b300_c1	14680	1680	452	60453	7569
GHS_indef/bratu3d	16485	488	341	488	488
TSOPF/TSOPF_FS_b300_c2	22505	3986	813	7373	26520
TSOPF/TSOPF_FS_b300_c3	35435	5831	1648	40667	52032
GHS_indef/ncvxqp5	10880	88	136	3715	4141
TSOPF/TSOPF_FS_b162_c1	65850	92	190	1979	2499
GHS_indef/turon_m	17633	19	3555	46	465
GHS_indef/ncvxqp3	87069	578	1397	62260	59312
GHS_indef/ncvxqp7	262548	443	5615	174362	104209

Table 4: Number of delayed pivots for various orderings with $u_{ord} = 0.4, u_{num} = 10^{-8}$. † indicates iterative refinement failed to converge.

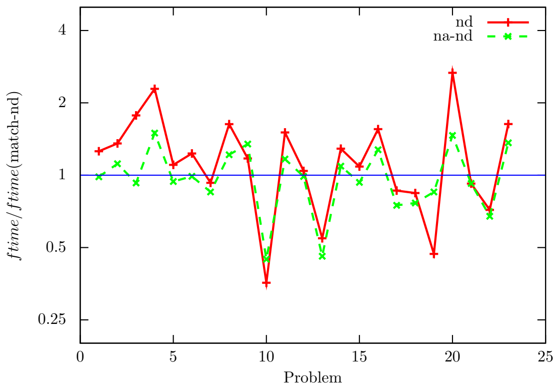
Problem	nd	match-nd	na-nd	gupta-small	gupta-zero
TSOPF/TSOPF_FS_b39_c7	4302	56	552	2508	494
QY/case39	6676	46	672	8155	283
TSOPF/TSOPF_FS_b39_c19	11617	160	1490	11579	1561
TSOPF/TSOPF_FS_b39_c30	18274	258	2355	11411	2665
GHS_indef/stokes128	8312	8	3426	21	21
TSOPF/TSOPF_FS_b162_c3	10431	24	369	3622	345
GHS_indef/darcy003	50042	79	15320	492	495
GHS_indef/cont-201	62858†	0	442	0	0
TSOPF/TSOPF_FS_b162_c4	17967	37	412	4394	899
GHS_indef/ncvxqp1	11960	8	28	4285	2609
GHS_indef/cont-300	127237†	0	974	0	0
GHS_indef/d_pretok	16634	106	1994	58	65
GHS_indef/cvxqp3	36648	97	70	16410	3894
TSOPF/TSOPF_FS_b300	13925	0	70	50	3
TSOPF/TSOPF_FS_b300_c1	13925	21	51	52955	3106
GHS_indef/bratu3d	7900	487	249	487	487
TSOPF/TSOPF_FS_b300_c2	20512	2	134	2547	380
TSOPF/TSOPF_FS_b300_c3	32560	3	202	28549	12035
GHS_indef/ncvxqp5	10131	0	29	3473	3891
TSOPF/TSOPF_FS_b162_c1	63112	1	4	995	361
GHS_indef/turon_m	17529	19	3478	40	46
GHS_indef/ncvxqp3	86383	397	727	54404	58885
GHS_indef/ncvxqp7	259652	300	2601	170873	103704

error of less than 10^{-14}. For the default setting ($u_{num} = 0.01$), no more than two iterations were required for each of the problems in our test set. For $u_{num} = 10^{-8}$ and the nd ordering, iterative refinement failed to converge for two problems (the solution diverged). For one problem the scaled backwards error for na-nd was $< 10^{-13}$ after ten iterations (so convergence not quite achieved) while for the other methods, for all problems at most five iterations were needed.

7.4 Comparison of factorization times

In Figure 7, we present times (in seconds) for the factorization phase of `HSL_MA97` (v2.3.0) using the nested dissection ordering algorithms. The results were obtained on a machine with two Xeon E5-2695 v3 processors providing a total of 28 cores; 16 OpenMP threads were used. The compiler was gfortran 4.8.2 using the options -g -O2 -fopenmp and we used BLAS from the Intel MKL v11.2.0 library. We see that, for a small number of examples (notably problems 10 and 13), nd and na-nd result in significantly faster factorization times but they are not universally faster than match-nd. Furthermore, using na-nd is generally faster than using nd. We are currently developing a new sparse indefinite solver [18] and we anticipate that, in terms of factorization times, its pivoting strategy will benefit more than `HSL_MA97` from the numerically-aware approach.

Figure 7: A comparison of the factorization times for the nd and na-nd orderings normalised against the match-nd ordering.



7.5 Comparison on general matrices

Finally, we measure the overhead of using na-nd on general indefinite matrices, for which a numerically-aware ordering is not required. We use the set of 25 general symmetric problems from [21], shown in Table 5; the set includes some problems with a saddle-point structure (for instance, GHS_indef/c-72).

Once these problems have been scaled, very few delayed pivots are generated under any ordering. The flop counts for nd, match-nd and na-nd are compared in Figure 8. In most cases the counts for the matching-based orderings are similar to those for traditional nested dissection. Where there is a difference, the overhead of using na-nd is considerably less than using match-nd. For 20 out of the 25 problems, na-nd results in less than a 10% overhead in the flop count.

We observe that for many of these test examples that do not have zeros on the diagonal, the matching may be largely on the diagonal entries. Consequently, the ordering has few constraints, so both the match-nd and na-nd approaches produce orderings that are similar to the traditional nd ordering.

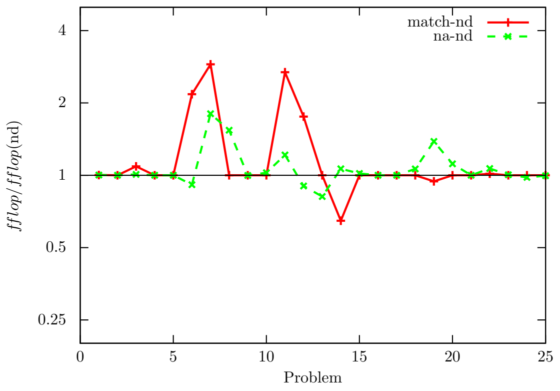
8 Conclusions

We have presented a new version of the nested dissection sparse matrix ordering algorithm that maintains a matching based on the numerical values. For tough symmetric indefinite problems, this new algorithm is able to deliver a significantly lower operation count for a numerical factorization than standard nested dissection whilst also keeping the number of delayed pivots small.

Table 5: General symmetric indefinite problems. n and $nz(A)$ denote the dimension of and the number of non-zero entries in A. $nz(L)$ and $fflop$ are the number of entries in L and the flop count to compute L returned by the analyse phase of HSL_MA97 with nested dissection ordering.

	Problem	n	$nz(A)$	$nz(L)$	$fflop$	Description/Application
1.	GHS_indef/boyd1	93279	1211231	6.53×10^5	4.67×10^6	Optimization: convex QP
2.	GHS_indef/dixmaanl	60000	299998	6.53×10^5	7.55×10^6	Optimization: Dixon-Maany problem
3.	GHS_indef/a2nnsnsl	80016	347222	8.49×10^5	9.64×10^6	Optimization: linear complementarity
4.	GHS_indef/blockqp1	60012	640033	7.80×10^5	1.02×10^7	Optimization: Block QP
5.	Oberwolfach/rail_79841	79841	553921	2.98×10^6	2.35×10^8	Model reduction problem
6.	GHS_indef/dawson5	51537	1010777	4.82×10^6	1.03×10^9	Structural: aeroplane actuator
7.	GHS_indef/c-72	84064	707546	4.62×10^6	1.68×10^9	Optimization
8.	Boeing/bcsstk39	46772	2060662	7.93×10^6	2.24×10^9	Structural: solid state rocket booster
9.	GHS_indef/helm2d03	392257	2741935	2.33×10^7	5.75×10^9	Helmholtz
10.	Oberwolfach/filter3D	106437	2707179	1.95×10^7	7.27×10^9	Model reduction
11.	Boeing/pct20stif	52329	2698463	1.16×10^7	7.45×10^9	Structural
12.	GHS_indef/copter2	55476	759952	1.19×10^7	7.58×10^9	CFD: rotor blade
13.	Boeing/crystk03	24696	1751178	1.11×10^7	8.25×10^9	Structural
14.	Andrianov/mip1	66463	10352819	1.21×10^7	8.71×10^9	Optimization
15.	Koutsovasilis/F2	71505	5294285	2.01×10^7	9.19×10^9	Structural: engine piston rod
16.	McRae/ecology1	1000000	4996000	4.61×10^7	1.33×10^{10}	Landscape ecology model
17.	Oberwolfach/gas_sensor	66917	1703365	2.96×10^7	3.01×10^{10}	Model reduction
18.	Cunningham/qa8fk	66127	1660579	3.02×10^7	3.65×10^{10}	Acoustics
19.	GHS_indef/bmw3_2	227362	11288630	6.13×10^7	5.98×10^{10}	Structural
20.	Oberwolfach/t3dh	79171	4352105	6.04×10^7	1.12×10^{11}	Model reduction: micropyros thruster
21.	Lin/Lin	256000	1766400	9.01×10^7	1.63×10^{11}	Eigenvalue problem
22.	GHS_indef/sparsine	50000	1548988	2.05×10^8	1.34×10^{12}	Structural optimization
23.	PARSEC/Ga10As10H30	113081	6115633	6.41×10^8	6.48×10^{12}	Quantum chemistry
24.	PARSEC/Ge99H100	112985	8451395	6.49×10^8	6.57×10^{12}	Quantum chemistry
25.	PARSEC/Ga19As19H42	133123	8884839	8.13×10^8	8.93×10^{12}	Quantum chemistry

Figure 8: A comparison of the factorization flop counts for the match-nd and na-nd orderings normalised against the nd ordering for the test set given in Table 5.



We have also compared the unpublished strategies suggested by Gupta and demonstrated they do not offer significant benefit in terms of numerical quality over pre-existing matching-based methods or our new scheme. We note however that in many cases they are much faster to apply.

As discussed in Section 3, we have used the threshold partial pivoting strategy of [7] . Other possible pivoting strategies include Bunch-Kaufman [4] and more advanced variants, such as those discussed by Ashcraft, Grimes and Lewis [3]. But tough indefinite problems will still require an approach such as matching to reduce the number of delayed pivots. For example, the direct solver PARDISO [25] combines dense Bunch-Kaufman pivoting with a static pivoting strategy and for saddle-point systems recommends the use of matching [14, 26].

Potential future work includes applying the modified AMD approach of Duff and Pralet at the base level of our numerically-aware nested dissection algorithm to try and deliver further benefits. Our aim will then be to optimize the implementation of the resulting algorithm and to integrate it into our existing nested dissection software as an additional option.

Code Availability

A development version of the nested dissection ordering software used in this paper may be checked out of our source code repository using the following command:

```
svn co -r619 http://ccpforge.cse.rl.ac.uk/svn/spral/branches/num_aware_nd
```

This code has not been optimised for high performance and we do not currently plan to release it as part of the mathematical software libraries that the Computational Mathematics Group develops and maintains at the STFC Rutherford Appleton Laboratory (see `http://www.hsl.rl.ac.uk/` and `http://www.numerical.rl.ac.uk/spral/`).

Acknowledgements

We would like to thank Anshul Gupta of IBM Research for sharing details of his approach with us. We are also grateful to our colleague Iain Duff for commenting on a draft of this paper. Many thanks also to three anonymous reviewers for their helpful and constructive feedback.

References

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, 30(3):381–388, 2004.

[2] C. Ashcraft, I. S. Duff, J. D. Hogg, J. A. Scott, and H. S. Thorne. Nested dissection revisited. Technical Report RAL-TR-2016-4, STFC Rutherford Appleton Laboratory, 2016. `http://purl.org/net/epubs/work/24733312`.

[3] C. Ashcraft, R. G. Grimes, and J. G. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM J. on Matrix Analysis and Applications*, 20(2):513–561, 1999.

[4] J. R. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation*, 31:163–179, 1977.

[5] I. S Duff. MA57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software*, 30(2):118–144, 2004.

[6] I. S. Duff and J. R. Gilbert. Maximum-weighted matching and block pivoting for symmetric indefinite systems. In *Abstract book of Householder Symposium XV*, pages 73–75, June 2002.

[7] I. S. Duff, N. I. M. Gould, J. K. Reid, J. A. Scott, and K. Turner. Factorization of sparse symmetric indefinite matrices. *IMA Journal of Numerical Analysis*, 11:181–2044, 1991.

[8] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. on Matrix Analysis and Applications*, 20(4):889–901, 1999.

[9] I. S. Duff and S. Pralet. Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM J. on Matrix Analysis and Applications*, 27:313–240, 2005.

[10] A. George. Nested dissection of a regular finite-element mesh. *SIAM J. on Numerical Analysis*, 10:345–363, 1973.

[11] A. George and J.W.H. Liu. *Computer solution of large sparse positive definite systems.* Prentice-Hall, Englewood Cliffs, N.J., 1981.

[12] A. Gupta. Private communication, 2015.

[13] A. Gupta and H. Avron. WSMP: Watson Sparse Matrix Package. Part I - direct solution of symmetric systems. Version 13.06. Technical Report RC 21886, IBM T. J. Watson Research Center, Yorktown Heights, NY, 2013. `http://www.research.ibm.com/projects/wsmp`.

[14] M. Hagemann and O. Schenk. Weighted matchings for preconditioning symmetric indefinite linear systems. *SIAM J. on Scientific Computing*, 28(2):403–420, 2006.

[15] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. *SIAM J. Sci. Comput.*, 16:452–469, 1994.

[16] B. Hendrickson and R. Leland. The Chaco users guide, version 2.0. Technical Report SAND942692, Sandia National Laboratories, Albuquerque, NM, 1995.

[17] J. D. Hogg. *High Performance Cholesky and Symmetric Indefinte Factorizations with Applications.* PhD thesis, University of Edinburgh, 2010.

[18] J. D. Hogg. A new sparse LDLT solver using a posteriori threshold pivoting. Technical Report RAL-TR-2016-017, STFC Rutherford Appleton Laboratory, 2016.

[19] J. D. Hogg and J. A. Scott. `HSL_MA97`: a bit-compatible multifrontal code for sparse symmetric systems. Technical Report RAL-TR-2011-024, STFC Rutherford Appleton Laboratory, 2011.

[20] J. D. Hogg and J. A. Scott. Pivoting strategies for tough sparse indefinite systems. *ACM Transactions on Mathematical Software*, 40, 2013. Article 4, 19 pages.

[21] J. D. Hogg and J. A. Scott. Compressed threshold pivoting for sparse symmetric indefinite systems. *SIAM J. on Matrix Analysis and Applications*, 35(2):783–817, 2014.

[22] G. Karypis and V. Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system. Technical Report TR 95-035, University of Minnesota, 1995.

[23] G. Karypis and V. Kumar. METIS: A software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing orderings of sparse matrices - version 4.0, 1998. `http://www-users.cs.umn.edu/~karypis/metis/`.

[24] F. Pellegrini. SCOTCH 6.0, 2012. `http://www.labri.fr/perso/pelegrin/scotch/`.

[25] O. Schenk and K. Gärtner. On fast factorization pivoting methods for symmetric indefinite systems. *Electronic Transactions on Numerical Analysis*, 23:158–179, 2006.

[26] O. Schenk, A. Wächter, and M. Hagemann. Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization. *J. of Computational Optimization and Applications*, 36(2-3):321–341, 2007.

[27] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. on Algebraic Discrete Methods*, 2(1):77–79, 1981.