On matrix symmetrization and sparse direct solvers

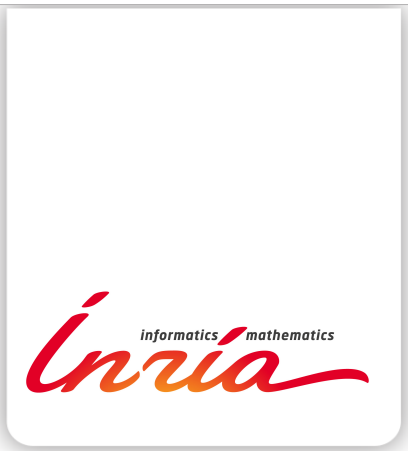
Raluca Portase, Bora Uçar

On matrix symmetrization and sparse direct solvers

Raluca Portase, Bora Uçar

RESEARCH REPORT
N° RR-8977
November 2016
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--RR-8977--FR+ENG



On matrix symmetrization and sparse direct solvers

Raluca Portase*, Bora Uçar†

Project-Team ROMA

Research Report n° RR-8977 — November 2016 — 31 pages

Abstract: We investigate algorithms for finding column permutations of sparse matrices in order to have large diagonal entries and to have many entries symmetrically positioned around the diagonal. The aim is to improve the memory and running time requirements of a certain class of sparse direct solvers. We propose efficient algorithms for this purpose by combining two existing approaches and demonstrate the effect of our findings in practice using a direct solver. In particular, we show improvements in a number of components of the running time of a sparse direct solver with respect to the state of the art on a diverse set of matrices.

Key-words: Sparse matrix, bipartite matching, LU decomposition

* Technical University of Cluj Napoca, Romania and ENS Lyon, France

† CNRS and LIP (UMR5668 CNRS-ENS Lyon-INRIA-UCBL), 46, allée d'Italie, ENS Lyon, Lyon F-69364, France.

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Sur la symmétrisation de matrices et des solveurs directs

Résumé : Nous étudions des algorithmes pour trouver des permutations de colonnes de matrices creuses afin d'avoir de grandes entrées sur la diagonale et d'avoir de nombreuses entrées symétriquement positionnées autour de la diagonale. Notre but est d'améliorer la mémoire et le temps d'exécution d'une certaine classe de solveurs directs creux. Nous proposons des algorithmes efficaces à cet effet en combinant deux approches existantes et exposons l'effet de nos résultats dans la pratique en utilisant un solveur direct. En particulier, nous montrons des améliorations dans de plusieurs components du temps d'exécution d'un solveur direct creux par rapport à l'état de l'art sur un ensemble divers de matrices.

Mots-clés : Matrice creuse, couplage biparti, décomposition LU

1 Introduction

We investigate bipartite matching algorithms for computing column permutations of a sparse matrix to achieve the following two objectives: (i) the main diagonal of the permuted matrix has entries that are large in absolute value; (ii) the sparsity pattern of the permuted matrix is as symmetric as possible. Our aim is to improve the memory and run time requirements of certain sparse direct solvers for unsymmetric matrices. The sparse direct solvers that we address perform computations using the nonzero pattern of the symmetrized matrix, noted $|\mathbf{A}|+|\mathbf{A}|^T$ for a square matrix $\mathbf{A}$, and are exemplified by MUMPS [2, 4].

Olschowka and Neumaier [18] formulate the first objective as a maximum weighed bipartite matching problem, which is polynomial time solvable. In particular, they argue that finding a column permutation which maximizes the product of the absolute values of the diagonal entries should be useful. Duff and Koster [11, 12] offer efficient implementations of a set of exact algorithms for the bipartite matching problem in the HSL [14] subroutine MC64 to permute matrices. One of the algorithms implements Olschowka and Neumaier's matching (called the maximum product perfect matching). This algorithm obtains two diagonal matrices $\mathbf{D}_r$ and $\mathbf{D}_c$, and a permutation matrix $\mathbf{Q}_{\text{MC64}}$ such that in $\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c$ all diagonal entries are equal to 1 in absolute value, and all other entries are less than or equal to 1 in absolute value. This preprocessing is shown to be very useful in avoiding pivoting [3, 12, 17, 18].

Uçar [20] investigates the second objective for $(0,1)$-matrices. Referring to two earlier studies [6, 8], he notes that the problem is NP-complete and proposes iterative improvement based heuristics. It has been observed that MUMPS works more efficiently with respect to another solver for pattern symmetric matrices [3]. Therefore, column permutations increasing the symmetry should be useful for MUMPS and similar direct solvers in improving their efficiency for a given matrix.

MC64 based preprocessing does not address the pattern symmetry; it can hurt the existing symmetry and deteriorate the performance (that is why it is applied with caution in MUMPS [3]). The heuristics for the second problem do not address the numerical issues and can lead to much higher pivoting (with respect to MC64 based preprocessing). Our aim in this paper is to find a matching that is useful both for numerical issues and pattern symmetry. As this is a multi-objective optimization problem with one of the objectives being NP-complete, the whole problem is NP-complete. We propose a heuristic to this problem by combining the algorithms from MC64 and Uçar's earlier work [20]. We first permute and scale the matrix using MC64. We then adapt the earlier symmetrization heuristic to consider only a subset of the nonzeros of the resulting matrix as candidates to be on the diagonal. The subset is chosen so that it would be helpful for numerical pivoting. By choosing all nonzero entries of $\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c$ to be in that subset, one recovers a pattern symmetrizing matching [20]; by choosing all nonzero entries of $|\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c|$ that are one to be in that subset one recovers an MC64 matching (with improved pattern symmetry). This is tied to a parameter to strike a balance between numerical issues and pattern symmetry.

The standard preprocessing phase for direct solvers for unsymmetric matrices computes an MC64 matching, and then orders the matrices for reducing the fill-in. Our multi-objective matching heuristic can effectively take place in between these two steps; by using MC64's matching as input it just needs to improve the symmetry while not losing the perspective on numerical aspects. We aim to improve all the remaining steps in solving the initial linear system. In particular, on a diverse set of 32 matrices we report 7% improvement in the ordering time, 11% improvement in the real space required to store the factors, 18% improvement in the operation count, and 12% improvement in

the total factorization and solution time of MUMPS (without taking the preprocessing time into account), with respect to the current state of the art.

The mentioned improvements are possible by introducing some overhead in the standard preprocessing phase of direct solvers. If we include the overhead in the total run time, the proposed method increases the total run time (the time spent in preprocessing, analysis, factorization and solution) by 11%. The experimental results show that the culprit is a certain component of the proposed method. This component takes longer time, in general, than MC64 in the maximum product perfect matching case. In this component, we use MC64 to solve a maximum weighted perfect matching problem defined using a subset of the entries of the original matrix, with some special edge weights. To our surprise, MC64's code for this purpose (also called for solving the maximum product perfect matching problem) runs slower than usual, and sometimes has very long run time. This component is isolated from the rest of the proposed method. It can be skipped but then one expects little gain in the memory use, operation count, and run time. Alternatively, one can try to improve this component either by finding replacements for MC64, or one can develop other techniques for the same purpose. Both of these two alternatives amount to combinatorial problems which we do not address in this paper, but leave as future work.

The paper is organized as follows. We introduce the notation and give some background on bipartite graphs and MC64 in Section 2. Since we build upon, extend, and improve the earlier work [20], we summarize it in the same section. This section also contains a brief summary of the standard preprocessing phase of sparse direct solvers for unsymmetric matrices. Next, in Section 3, we propose a modification to the standard preprocessing phase to incorporate column permutation methods achieving the two objectives. This section also explains necessary changes to the earlier work which are proposed in order to achieve the two objectives by making use of MC64. We have engineered the main data structures and the algorithms of the earlier work [20] for efficiency. We also summarize these. Later, we investigate the effect of the proposed method in Section 4, where we document the improvements with respect to the earlier work and observe the practical effects of the proposed method on MUMPS.

2 Background and notation

We associate a bipartite graph $G_{\mathbf{A}} = (R \cup C, E)$ with a given square $(n \times n)$ matrix $\mathbf{A}$. Here, R and C are two disjoint vertex sets that correspond to the set of rows and the set of columns of the matrix, and E is the edge set in which $e = (r_i, c_j) \in E$ if and only if $a_{ij} \neq 0$. When $\mathbf{A}$ is clear from the context, we use G for brevity. Although the edges are undirected, we will refer to an edge as $e = (r_i, c_j)$, the first vertex being a row vertex and the second one a column vertex. We say that r_i is adjacent to c_j if there is an edge $(r_i, c_j) \in E$. We use $\text{adj}_G(v)$, or when G is clear from the context simply $\text{adj}(v)$, to denote the set of vertices u where $(v, u) \in E$.

A matching $\mathcal{M}$ is a subset of edges no two of which share a common vertex. For a matching $\mathcal{M}$, we use $\text{mate}_{\mathcal{M}}(v)$ to denote the vertex u where $(v, u) \in \mathcal{M}$. The matching $\mathcal{M}$ is normally clear from the context, and we simply use $\text{mate}(v)$. If $\text{mate}(c) = r$, then $\text{mate}(r) = c$. We also extend this to a set S of row or column vertices such that $\text{mate}(S) = \{v : (v, s) \in \mathcal{M} \text{ for some } s \in S\}$. If all vertices appear in an edge of a matching $\mathcal{M}$, then $\mathcal{M}$ is called a perfect matching. If the edges are weighted, the weight of a matching is defined as the sum of the weights of its edges. The minimum and maximum weighted perfect matching problems are well known [16]. An $\mathcal{M}$-alternating cycle is a simple cycle whose edges are alternately in $\mathcal{M}$ and not in $\mathcal{M}$. By alternating an $\mathcal{M}$-alternating

cycle, one obtains another matching (with the same number of matched edges as $\mathcal{M}$), where the matching pairs are interchanged along the edges of the cycle. We use $\mathcal{M} \oplus \mathcal{C}$ to denote alternating the cycle $\mathcal{C}$.

A perfect matching $\mathcal{M}$ defines a permutation matrix $\mathbf{M}$ where $m_{ij} = 1$ for $(r_j, c_i) \in \mathcal{M}$. We use calligraphic letters to refer to perfect matchings, and the corresponding capital, Roman letters to refer to the associated permutation matrices. If $\mathbf{A}$ is a square matrix, and $\mathcal{M}$ is a perfect matching in its bipartite graph, then $\mathbf{AM}$ has the diagonal entries identified by the edges of $\mathcal{M}$.

The pattern of a matrix is the position of its nonzero entries. The *pattern symmetry score* of a matrix $\mathbf{A}$, denoted as SymScore($\mathbf{A}$), is defined as the number of nonzeros a_{ij} for which a_{ji} is a nonzero as well. Note that the diagonal entries contribute one, and a pair of symmetrically positioned off-diagonal entries contributes two to SymScore($\mathbf{A}$). Observe that if $\mathbf{A}$ is symmetric, SymScore($\mathbf{A}$) is equal to the number of nonzero entries of $\mathbf{A}$. In our combinatorial algorithms, we use the discrete quantity SymScore($\mathbf{A}$) as the optimization metric. Since, the dimensionless variant SymScore($\mathbf{A}$)/τ measures how symmetric the matrix $\mathbf{A}$ with τ nonzeros is, we use this variant later in the experiments, and call it the *pattern symmetry ratio*.

We use Matlab notation to refer to a submatrix in a given matrix. For example, $\mathbf{A}([r_1, r_2], [c_1, c_2])$ refers to the 2×2 submatrix of $\mathbf{A}$ which is formed by the entries at the intersection of the rows r_1 and r_2 with the columns c_1 and c_2.

2.1 Two algorithms from MC64

We use two algorithms implemented in MC64 [11, 12]. Given a square matrix $\mathbf{A}$, the first one seeks a permutation σ such that $\sum |a_{\sigma(i),i}|$ is maximized. This corresponds to finding a permutation matrix $\mathbf{P}$ such that $\mathbf{AP}$ has the largest sum of absolute values of the diagonal entries. This is achieved by defining another matrix $\mathbf{C}$ such that

$$c_{ij} = \begin{cases} a_j - |a_{ij}| & \text{for } a_{ij} \neq 0 \ , \\ \infty & \text{otherwise.} \end{cases}$$

where $a_j = \max_i\{|a_{ij}|\}$ is the largest absolute value of an entry in the jth column of $\mathbf{A}$. Then the minimum weight perfect matching problem on the bipartite graph of $\mathbf{C}$ is solved. There are a number of polynomial time algorithms for this purpose [7, Ch. 4]; the one that is implemented in MC64 is based on the shortest augmenting paths and has a worst case time complexity of $\mathcal{O}(n\tau \log n)$, for an $n \times n$ matrix with τ nonzeros.

Given a square matrix $\mathbf{A}$, the second algorithm from MC64 seeks a permutation σ such that $\prod |a_{\sigma(i),i}|$ is maximized. This corresponds to finding a permutation matrix $\mathbf{P}$ such that $\mathbf{AP}$ has the largest product of absolute values of the diagonal entries. This is formulated again as the minimum weight perfect matching problem on the bipartite graph corresponding to the matrix $\mathbf{C}$ defined as

$$c_{ij} = \begin{cases} \log a_j - \log |a_{ij}| & \text{for } a_{ij} \neq 0 \ , \\ \infty & \text{otherwise,} \end{cases}$$

with the same definition of a_j. After this transformation, MC64 uses the same shortest augmenting path based algorithm to find the desired permutation. One important property of the shortest augmenting path based algorithms is that they also compute variables u_i and v_j for $i, j = 1, \ldots, n$ for the rows and the columns which satisfy the following properties

$$\begin{cases} u_i + v_j = c_{ij} & \text{for } \sigma(j) = i \ , \\ u_i + v_j \leq c_{ij} & \text{for } c_{ij} \neq 0 \text{ and } \sigma(j) \neq i \ . \end{cases}$$

This property is important, because it can be used to construct two diagonal matrices $\mathbf{D}_r = \text{diag}(p_1, p_2, \ldots, p_n)$ and $\mathbf{D}_c = \text{diag}(q_1, q_2, \ldots, q_n)$, where $p_i = \exp(u_i)$ and $q_j = \exp(v_j)/a_j$ such that the diagonal of the permuted and scaled matrix $\mathbf{D}_r\mathbf{A}\mathbf{P}\mathbf{D}_c$ contains entries of absolute value 1, while all other entries have an absolute value no larger than 1.

2.2 Algorithms for symmetrizing pattern

Here we review the heuristic from the earlier work [20] for improving the pattern symmetry score of matrices. This heuristic works on the bipartite graph associated with the pattern of $\mathbf{A}$. It starts with a perfect matching to guarantee a zero-free diagonal, and then iteratively improves the current matching to increase the pattern symmetry score while maintaining a perfect matching at all times.

Given a matrix $\mathbf{A}$ and a perfect matching $\mathcal{M}$, the pattern symmetry score of the permuted matrix, SymScore($\mathbf{AM}$), can be computed using the function shown in Algorithm 1. This algorithm run in $\mathcal{O}(n + \tau)$ time for an $n \times n$ matrix $\mathbf{A}$ with τ nonzeros.

Algorithm 1: Computing the pattern symmetry score of a matrix under a given matching [20].

Input : $\mathbf{A}$, an $n \times n$ matrix and $G = (R \cup C, E)$ the corresponding bipartite graph.
$\mathcal{M}$, a perfect matching.
Output: SymScore($\mathbf{AM}$), the pattern symmetry score of $\mathbf{AM}$

```
mark(r) ← 0 for all r ∈ R
score ← 0
foreach (r_i, c_j) ∈ M do
    foreach c ∈ adj(r_i) do
        mark(mate(c)) ← j                                   /* mark r_i too */
    foreach r ∈ adj(c_j) do
        if mark(r) = j then
            score ← score + 1  /* increase by one for (r_i, c_j), also for a   symmetric
              entry (r_k, c_j) ∉ M with r_k = mate(c), c ∈ adj(r_i).          */
```

The pattern symmetry score can be expressed in terms of alternating cycles of length four. Consider the following four edges forming a cycle: (r_i, c_j), $(r_k, c_\ell) \in \mathcal{M}$ and (r_i, c_ℓ), $(r_k, c_j) \in E$. These four edges contribute by four to the pattern symmetry score, where two nonzeros are on the diagonal of $\mathbf{AM}$, and the other two are positioned symmetrically around the diagonal. By counting the symmetrically positioned entries of $\mathbf{A}$ using the set of all alternating cycles of length four, one obtains the formula

$$\text{SymScore}(\mathbf{AM}) = n + 2 \times |C_4| \,, \tag{1}$$

where C_4 is the set of alternating cycles of length four.

By observing the role of the alternating cycles of length four in (1), Uçar [20] proposes iteratively improving the pattern symmetry score by alternating the current matching along a set of disjoint, length-four alternating cycles. For this to be done, the set of alternating cycles of length four with respect to the initial matching is computed. Then, disjoint cycles from this set are chosen, and the pattern symmetry score is tried to be improved by alternating the selected cycle. Uçar discusses two alternatives to choose the length-four alternating cycles. The first one randomly visits those

cycles. If a visited cycle is disjoint from the previously alternated ones, then the gain of alternating the current cycle is computed, and the matching is alternated if the gain is nonnegative. The second one keeps the cycles in a priority key, using the gain of the cycles as the key value. Then, the cycle with the highest gain is selected from the priority queue, and the current matching is tentatively alternated along that cycle. At the end, the longest profitable prefix of alternations are realized. This second alternative obtained better results than the first one, but involved more data structures and operations and hence was deemed slower. In this work, we carefully re-implement the second alternative by proposing methods to update gains of the alternating cycles whenever necessary, rather than re-computing them as done in the earlier work. We also incorporate a few short-cuts to further improve the run time.

Uçar [20] proposes two upper bounds on the possible pattern symmetry score. One of them requires finding many maximum weighted matchings, and is found to be expensive. The other one, called UB1 [20], corresponds to the maximum weight of a perfect matching in the bipartite graph of $\mathbf{A}$, where the weight of an edge $(r_i, c_j) \in E$ is set to

$$\min\{|\operatorname{adj}(r_i)|, |\operatorname{adj}(c_j)|\} \; . \tag{2}$$

To see why, let us count the potentially symmetry entries from the rows, one row at a time (two symmetric entries will be counted one by one). When the row r_i and the column c_j are put in the corresponding positions, the maximum number of nonzeros in row r_i that can be have a symmetric entry is $\min\{|\operatorname{adj}(r_i)|, |\operatorname{adj}(c_j)\}$. Since two off-diagonal symmetrically positioned entries are in two different rows, summing up the weights of the edges (2) in a matching gives an upper bound. Uçar proposes to initialize the iterative improvement method using perfect matchings attaining UB1 and discusses that this helps in obtaining good results. We note that using a perfect matching attaining UB1 is a heuristic to initialize the iterative improvement method; any other perfect matching can be used.

2.3 Preprocessing phase of direct solvers

In the current-state-of-the-art direct solvers, the most common preprocessing steps applied to a given unsymmetric matrix $\mathbf{A}$ for better numerical properties and sparsity are summarized in Algorithm 2. In this algorithm, we use MUMPS to instantiate all components for clarity. First, MC64 is applied to find a column permutation $\mathbf{Q}_{\text{MC64}}$ and the associated diagonal scaling matrices $\mathbf{D}_r$ and $\mathbf{D}_c$, where $|\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c|$ has ones along the diagonal and has all other entries no larger than one. Then, the matrix is ordered for reducing the potential fill-in. In MUMPS and similar solvers, this is done by ordering the symmetrized matrix $|\mathbf{A}\mathbf{Q}_{\text{MC64}}| + |\mathbf{A}\mathbf{Q}_{\text{MC64}}|^T$, and permuting the matrix $\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c$ symmetrically with the permutation matrix $\mathbf{P}$ corresponding to the ordering found. Usually, AMD [1], MeTiS [15], or Scotch [19] are used for finding orderings. After all these preprocessing, the direct solver effectively factorizes

$$\mathbf{A}' = \mathbf{P}\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c\mathbf{P}^T \; . \tag{3}$$

The partial threshold pivoting scheme accepts a diagonal entry as a pivot, if it is larger than a given threshold times the maximum entry (in absolute values) in the current column. MUMPS uses 0.01 as default value; the number typically is between 0.001 and 0.1 [3].

Algorithm 2: The standard preprocessing steps of a direct solver for a sparse, unsymmetric matrix $\mathbf{A}$; specialized for MUMPS

Input: $\mathbf{A}$, a matrix

$\langle \mathbf{D}_r, \mathbf{D}_c, \mathbf{Q}_{\text{MC64}} \rangle \leftarrow \text{mc64}(|\mathbf{A}|)$
$\mathbf{P} \leftarrow$ fill-reducing-ordering$(|\mathbf{A}\mathbf{Q}_{\text{MC64}}| + |\mathbf{A}\mathbf{Q}_{\text{MC64}}|^T)$
$\mathbf{A}' \leftarrow \mathbf{P}\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c\mathbf{P}^T$
$t \leftarrow 0.01$ `/* default value for MUMPS */`
call MUMPS on $\mathbf{A}'$ with partial threshold pivoting using the threshold t

3 Matrix symmetrization for direct solvers

Our aim is to find another column permutation $\mathbf{Q}$ instead of $\mathbf{Q}_{\text{MC64}}$ in (3) so that $\mathbf{AQ}$ is more pattern symmetric than $\mathbf{A}\mathbf{Q}_{\text{MC64}}$. Additionally, $\mathbf{Q}$ should be numerically useful.

The immediate idea of using the algorithm from Section 2.2 increases the pattern symmetry. However, this does not take the numerics into account, and the overall factorization is likely to fail in many cases (see a short discussion in Section 4.2), if one does not modify parameters. We therefore propose finding another permutation matrix $\mathbf{Q}_{\text{Pat}}$ after MC64 and before computing the fill-reducing ordering. That is, we propose adding another step in the preprocessing, so that the matrix that is factorized is

$$\mathbf{A}'' = \mathbf{R}\mathbf{D}_r\mathbf{A}\mathbf{Q}\mathbf{D}'_c\mathbf{R}^T \,, \tag{4}$$

where $\mathbf{R}$ is a permutation matrix corresponding to a fill-reducing ordering,

$$\mathbf{Q} = \mathbf{Q}_{\text{MC64}}\mathbf{Q}_{\text{Pat}} \tag{5}$$

is a permutation matrix, and

$$\mathbf{D}'_c = \mathbf{Q}_{\text{Pat}}{}'\mathbf{D}_c\mathbf{Q}_{\text{Pat}} \tag{6}$$

is a diagonal matrix (a symmetrically permuted version of $\mathbf{D}_c$). By focusing only on the pattern while computing $\mathbf{Q}_{\text{Pat}}$, the proposed preprocessing step separates numerical issues from the structural (pattern-wise) issues in achieving the two objectives described before. This modified preprocessing framework is shown in Algorithm 3, where the numbered lines contain the differences with respect to the existing framework shown in Algorithm 2. The Lines 1, 2, 4, and 5 are straightforward. At Line 1, we find a threshold such that there are $q\tau$ nonzeros of $|\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c|$ that are no smaller than this value. Such order statistics can be found in $\mathcal{O}(\tau)$ time [9, Ch. 9]; we used Matlab's `quantile` function for this purpose. At Line 2, we select those entries of the scaled matrix that are no smaller than `threshold` and put them into the matrix $\mathbf{A}_f$. The entries in $\mathbf{A}_f$ will be allowed to be in the diagonal, and since they are larger than a threshold, we expect the identified diagonal to be heavy. Line 3 is the proposed ImproveSymmetry algorithm which has two substeps. In the first substep, we find an initial permutation $\mathbf{M}_0$ on $\mathbf{A}_f$ based on UB1 (other methods can be used), and in the second substep we iteratively improve this initial permutation for better pattern symmetry score, obtaining $\mathbf{Q}_{\text{Pat}}$. At the end $\mathbf{Q}_{\text{MC64}}\mathbf{Q}_{\text{Pat}}$ is useful for the two objectives that we seek to achieve. At Line 4, we select the minimum absolute value of a diagonal entry of the scaled and permuted matrix to set a threshold value for the partial pivoting. Notice that we are setting this threshold value depending on the (permuted and scaled) matrix $\mathbf{A}$. This is needed because of the fact that ImproveSymmetry can permute nonzeros that have magnitude

less than 1 into the diagonal. Since we want the initial sequence of pivots respected as much as possible after fill-reducing ordering, we allow small pivots, and control the pivoting by defining the threshold for partial pivoting at Line 4. Note that $t/100$ is used in order to recover the behavior of Algorithm 2, if $\mathbf{Q}_{\text{Pat}}$ is identity, or $\mathbf{Q}_{\text{MC64}}\mathbf{Q}_{\text{Pat}}$ also corresponds to a maximum product perfect matching. Since smaller values constitute a numerically worse pivot sequence more steps of iterative refinement could be required. At Line 5, the direct solver MUMPS is called to factorize $\mathbf{A}''$ and solve the original linear system.

Algorithm 3: The proposed preprocessing of a sparse matrix $\mathbf{A}$ for a direct solver

Input : $\mathbf{A}$, a matrix.
q, a number between 0 and 1.

$\langle \mathbf{D}_r, \mathbf{D}_c, \mathbf{Q}_{\text{MC64}} \rangle \leftarrow \text{mc64}(|\mathbf{A}|)$
1 threshold $\leftarrow q$ quantile of $|\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c|$
2 $\mathbf{A}_f \leftarrow |\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c| \geq$ threshold `/* $q\tau$ entries which are $\geq$ threshold are in $\mathbf{A}_f$ */`
3 { Compute an initial permutation $\mathbf{M}_0$ on $\mathbf{A}_f$ based on UB1, or other method
$\mathbf{Q}_{Pat} \leftarrow$ IterativeImprove($\mathbf{A}\mathbf{Q}_{\text{MC64}}, \mathbf{A}_f, \mathbf{M}_0$)
$\mathbf{R} \leftarrow$ fill-reducing-ordering($|\mathbf{A}\mathbf{Q}| + |\mathbf{A}\mathbf{Q}|^T$) `/* Q as in (5) */`
$\mathbf{A}'' \leftarrow \mathbf{R}\mathbf{D}_r\mathbf{A}\mathbf{Q}\mathbf{D}'_c\mathbf{R}^T$ `/* $\mathbf{D}'_c$ as in (6) */`
4 $t \leftarrow \min(\text{diag}(|\mathbf{A}''|))$
5 call MUMPS on $\mathbf{A}''$ with the partial threshold pivoting $\frac{t}{100}$ `/* if $t = 1$ this becomes equivalent to the default choice for MUMPS. */`

Since $\mathbf{AQ}$ could be very different than $\mathbf{AQ}_{\text{MC64}}$ pattern-wise, there is no direct relation between their ordering. However, $\mathbf{AQ}$ is more pattern symmetric than $\mathbf{AQ}_{\text{MC64}}$, and therefore we expect better orderings by using $|\mathbf{AQ}|+|\mathbf{AQ}|^T$. With this, we expect improvements on all remaining steps of solving the initial linear system. First, since $\mathbf{AQ}$ is more pattern symmetric than $\mathbf{AQ}_{\text{MC64}}$, the graph based ordering routines will have shorter run time and will be more effective, as $|\mathbf{AQ}|+|\mathbf{AQ}|^T$ is closer to $\mathbf{AQ}$ than $|\mathbf{AQ}_{\text{MC64}}| + |\mathbf{AQ}_{\text{MC64}}|^T$ to $\mathbf{AQ}_{\text{MC64}}$. Second, the factorization will require less memory and less operations, thanks to the improved ordering, during matrix factorization and solving with the triangular factors. This should also translate to a reduction in the total factorization and solution time, unless there are increased pivoting (during factorization) and more steps of iterative refinement [5] taken to recover the solution accuracy (due to the smaller threshold for pivoting used at Line 5 of Algorithm 3).

3.1 An iterative-improvement based heuristic

In order to separate numerical concerns from the structural ones, the proposed approach constraints the pattern symmetry improving matchings so as to include only large elements. Once the large elements are filtered into $\mathbf{A}_f$ at Line 2 of Algorithm 3, we call the two-step function ImproveSymmetry at Line 3 to improve the symmetry of $\mathbf{AQ}_{\text{MC64}}$ by matchings that include edges from $\mathbf{A}_f$. The function ImproveSymmetry starts from an initial matching and iteratively improves the pattern symmetry score of $\mathbf{AQ}_{\text{MC64}}$ with the proposed Algorithm 4. For the initial matching, we use the initialization UB1 from the earlier work (summarized in Section 2.2), but this time on the graph of $\mathbf{A}_f$, where $(r_i, c_j) \in E_f$ gets the weight with respect to $\mathbf{A}$ as in (2). We note two facts: (i) the nonzero pattern of $\mathbf{A}_f$ is a subset of that of $\mathbf{A}$; (ii) MC64 uses the same code for the maximum weighted perfect matching and the maximum product perfect matching

problems. Therefore, we expected shorter run time with MC64 while computing a UB1 attaining perfect matching than while computing a maximum product perfect matching. To our surprise this was not the case. In fact, computing a UB1 attaining perfect matching with MC64 turned out to be the most time consuming step in the overall method (more discussion is the experiments of Section 4.2.3). Overcoming the run time issue of this innocent looking component requires further investigation of perfect matching algorithms and potentially calls for effective methods replacing UB1.

The main components of the iterative improvement algorithm shown in Algorithm 4 are standard. The significant differences with respect to the earlier work [20] which enable the incorporation of the numerical concerns and improved run time are at the numbered lines.

Algorithm 4: IterativeImprove($\mathbf{A}, \mathbf{A}_f, \mathbf{M}_0$)

Input : $\mathbf{A}$, a matrix.
$\mathbf{A}_f$, another matrix containing a subset of the entries of $\mathbf{A}$. Only entries included in $\mathbf{A}_f$ are allowed to be in the matching.
$\mathbf{M}_0$, a permutation matrix corresponding to a perfect matching $\mathcal{M}_0$ on $\mathbf{A}$, which is also a perfect matching on $\mathbf{A}_f$

Output: $\mathbf{M}_1$, the permutation matrix corresponding to another perfect matching $\mathcal{M}_1$ where SymScore($\mathbf{AM}_1$) $\geq$ SymScore($\mathbf{AM}_0$)

Build $G = (R \cup C, E)$ corresponding to $\mathbf{A}$
1 Build $G_f = (R \cup C, E_f)$ corresponding to $\mathbf{A}_f$ `/* Used only in building` C_4 `below */`
$\mathcal{M}_1 \leftarrow \mathcal{M}_0$
currentSymm $\leftarrow$ SymScore($\mathbf{AM}_0$)
while *true* **do**
  initSymmScore $\leftarrow$ bestSymm $\leftarrow$ currentSymm
2 $C_4 \leftarrow \{(r_1, c_1, r_2, c_2) : (r_1, c_1) \in \mathcal{M}_1$ and $(r_2, c_2) \in \mathcal{M}_1$ and $(r_1, c_2) \in E_f$ and $(r_2, c_1) \in E_f\}$
3 Build a bipartite graph $G_o = (X \cup Y, E_o)$ where $X = R \cup C$, and Y contains a vertex for each cycle in C_4. A cycle-vertex is connected to its four vertices with an edge in E_o
  cycleHeap $\leftarrow$ a priority queue created from C_4 using the gains of the cycles as the key value
Pass **while** cycleHeap $\neq \emptyset$ **do**
    extract the cycle $\mathcal{C} = (r_1, c_1, r_2, c_2)$ with the maximum gain from cycleHeap
    currentSymm $\leftarrow$ currentSymm + gain[$\mathcal{C}$]
    if currentSymm + gain[$\mathcal{C}$] > bestSymm **then**
      update bestSymm
    $\mathcal{M}_1 \leftarrow \mathcal{M}_1 \oplus \mathcal{C}$ `/* now` (r_1, c_2) `and` (r_2, c_1) `are in` $\mathcal{M}_1$ `*/`
4   **foreach** $\mathcal{C}' \in \text{adj}_{G_o}(\{r_1, c_1, r_2, c_2\})$ **do**
      HeapDelete($\mathcal{C}'$)
5   **if** *no gain since a long time* **then**
      break the current pass
6   UpdateGains($G, \mathcal{C}, r_1, cycleHeap$)
7   UpdateGains($G, \mathcal{C}, r_2, cycleHeap$)
  rollback $\mathcal{M}_1$ to the point where bestSymm was observed
8 **if** *not enough gain after a pass* **then**
    break and no need to do another refinement pass
  initSymmScore $\leftarrow$ bestSymm

The Lines 1 and 2 of Algorithm 4 are related. The first line builds a bipartite graph from

the entries allowed to be in the diagonal (since those entries define $\mathbf{A}_f$, this graph corresponds to the bipartite graph of $\mathbf{A}_f$). The second line creates the set C_4 of the alternating cycles of length four with respect to the current matching in the graph G_f. Then, another bipartite graph G_o is constructed at Line 3. The graph G_o has the set of row and column vertices of G on one side, and the set C_4 on the other side; the edges of G_o show which vertex is included in which cycle. Before starting a pass at the while loop (labelled PASS in Algorithm 4), a priority queue is constructed on the set C_4 with the gain of alternating the cycle as the key value. For this purpose, we compute the gain values of the alternating cycles in C_4. Consider a cycle $\mathcal{C} = (r_1, c_1, r_2, c_2)$ where $(r_1, c_1), (r_2, c_2) \in \mathcal{M}_0$ and $\mathcal{M}_1 = \mathcal{M}_0$ at the beginning. The gain of alternating $\mathcal{C}$ can be computed as the difference

$$\text{gain}[\mathcal{C}] = s_1 - s_0 \ , \tag{7}$$

where

$$\begin{aligned} s_0 &= 2(|\operatorname{mate}(\operatorname{adj}_G(c_1)) \cap \operatorname{adj}_G(r_1)| + |\operatorname{mate}(\operatorname{adj}_G(c_2)) \cap \operatorname{adj}_G(r_2)|) \\ s_1 &= 2(|\operatorname{mate}(\operatorname{adj}_G(c_2)) \cap \operatorname{adj}_G(r_1)| + |\operatorname{mate}(\operatorname{adj}_G(c_1)) \cap \operatorname{adj}_G(r_2)|) \ . \end{aligned} \tag{8}$$

Here s_0 measures the contribution of the matched pairs (r_1, c_1) and (r_2, c_2) to the pattern symmetry score, whereas s_1 measures that of (r_1, c_2) and (r_2, c_1)—these two edges become matching after alternating $\mathcal{C}$. Notice that while the edges in G_f are allowed to be in a perfect matching, the gain of alternating a cycle is computed using G. This is so, as the pattern symmetry score is defined with respect to the matrix $\mathbf{A}$ in Algorithm 4. Once these gains have been computed, the algorithm extracts the most profitable cycle from the heap, updates the current pattern symmetry score by the gain of the cycle (which could be negative), and tentatively alternates $\mathcal{M}_1$ along $\mathcal{C}$. Then, all cycles containing the vertices of $\mathcal{C}$ are deleted from the heap. This guarantees that each vertex changes its mate at most once in a pass. Upon alternating $\mathcal{M}_1$ along $\mathcal{C}$, the gains of a set of cycles can change. We propose an efficient way to keep the gains up-to-date, instead of re-computing them as in the previous study [20]. Since all gains are even (symmetric pairs add two to the pattern symmetry score), we simplify the factor two from (8) and count the number of pairs that are lost or formed as the gain of alternating a cycle.

Alternating the cycle (r_1, c_1, r_2, c_2) can change the gain of all cycles of the form $\mathcal{C}' = (r_1', c_1', r_2', c_2')$, where $c_1' \in \operatorname{adj}_G(\{r_1, r_2\})$ or $c_2' \in \operatorname{adj}_G(\{r_1, r_2\})$. Any change in the gain of $\mathcal{C}'$ is due to the pattern of the nonzeros of $\mathbf{A}([r_1, r_2], [c_1', c_2'])$ and $\mathbf{A}([r_1', r_2'], [c_1, c_2])$. We separate this in two symmetrical cases: (c_1' or c_2') $\in \operatorname{adj}_G(r_1)$ and (c_1' or c_2') $\in \operatorname{adj}_G(r_2)$. Observe that in terms of the gain updates, those two cases are the opposite of each other. In other words, if we have the same adjacency for r_1 and r_2 with respect to c_1' and c_2', the amount of the gain that we get from r_1 is equal to the loss that we get from r_2, and vice versa. Therefore, the gain update logic can be simplified. We will discuss the gain updates only with respect to the neighbourhood of r_1. If both c_1' and $c_2' \in \operatorname{adj}_G(r_1)$ or if neither of them is in the adjacency of r_1, the amount of gain would not change. Figure 1 presents all the remaining possible modifications for the gain of cycle $\mathcal{C}'$. In this figure, the header of the columns shows the pattern of $\mathbf{A}(r_2', [c_1, c_2])$, that is, if these two entries are zero or nonzero (shown with $\times$). For example, the column header $0\times$ means that $c_1 \notin (\operatorname{adj}_G(r_2'))$ and $c_2 \in (\operatorname{adj}_G(r_2'))$. Similarly, the header of the rows shows the pattern of $\mathbf{A}(r_2', [c_1, c_2])$ and $\mathbf{A}(r_1', [c_1, c_2])$. To exemplify the use of these tables, we use Fig. 2, in which we represent only the nonzero entries of the two cycles of interest and their interactions. Here, $\mathcal{C} = (r_1, c_1, r_2, c_2)$ and $\mathcal{C}' = (r_1', c_1', r_2', c_2')$ are length-four alternating cycles. The gain of alternating $\mathcal{C}$ is -1. This is so, because if we match r_1' to c_2' and r_2' to c_1', the entry $a_{r_1, c_2'}$ will not have a symmetric pair. Now suppose that we alternated

$\mathbf{A}(r'_1,[c_1,c_2])$ \ $\mathbf{A}(r'_2,[c_1,c_2])$	00	0×	×0	××
00	0	−1	1	0
0×	1	0	2	1
×0	−1	−2	0	−1
××	0	−1	1	0

(a) $\mathbf{A}(r_1,[c'_1,c'_2]) = 0\times$

$\mathbf{A}(r'_1,[c_1,c_2])$ \ $\mathbf{A}(r'_2,[c_1,c_2])$	00	0×	×0	××
00	0	1	−1	0
0×	−1	0	−2	−1
×0	1	2	0	1
××	0	1	−1	0

(b) $\mathbf{A}(r_1,[c'_1,c'_2]) = \times 0$

Figure 1 – Cycle (r_1, c_1, r_2, c_2) is going to be alternated. The 4×4 cells show how to update the gain of the cycle (r'_1, c'_1, r'_2, c'_2) in two different cases a) $\mathbf{A}(r_1,[c'_1,c'_2]) = \times 0$ and b) $\mathbf{A}(r_1,[c'_1,c'_2]) = 0\times$. The header of the columns and the rows show the pattern of $\mathbf{A}(r'_2,[c_1,c_2])$ and $\mathbf{A}(r'_1,[c_1,c_2])$, respectively.

	c_1	c_2	c'_1	c'_2	
r_1	×	×		×	⋯
r_2	×	×			⋯
r'_1		×	×	×	⋯
r'_2	×		×	×	⋯
⋮	⋮	⋮	⋮	⋮	⋱

Figure 2 – When $\mathcal{C} = (r_1, c_1, r_2, c_2)$ is alternated, the gain of alternating the cycle $\mathcal{C}' = (r'_1, c'_1, r'_2, c'_2)$ can change only for the entries in $\mathbf{A}([r_1, r_2], [c'_1, c'_2])$. Before alternating $\mathcal{C}$, gain$[\mathcal{C}'] = -1$. The pattern of $\mathbf{A}([r'_1, r'_2], [c_1, c_2])$ corresponds to the cell $(0\times, \times 0)$ in Fig. 1a, and hence the gain of $\mathcal{C}'$ increases by 2 to be equal to 1, when $\mathcal{C}$ is alternated.

$\mathcal{C}$, even if a symmetrical entry is lost. The gain of the cycle $\mathcal{C}'$ will become 1, as alternating $\mathcal{C}'$ now recovers the lost entry. That is why we have to add two to the original gain of $\mathcal{C}'$, as also shown in the cell $(0\times, \times 0)$ of Fig. 1a. The observations from above are translated into the pseudocode shown in Algorithm 5, where Algorithm 4 calls this subroutine twice—once with r_1 as the third argument and once with r_2 at the same place. In the second call, Fig. 1a will be looked up for the case $\mathbf{A}(r_2,[c'_1,c'_2]) = \times 0$, and Fig. 1b will be looked up for the case $\mathbf{A}(r_2,[c'_1,c'_2]) = 0\times$.

3.2 Practical improvements

Apart from the proposed gain-update scheme, we use two techniques to improve the practical run time of Algorithm 4. The first one is to short cut a pass (Line 5). We set a limit on the number of moves (tentatively realized) between the best pattern symmetry score seen so far and the current number of moves. If this limit is reached, we break the pass. Our default setting uses the minimum of 50 and $0.005|C_4|$ computed at the beginning. The second one is to avoid performing passes with little total gain (Line 8). If the completed pass did not improve the pattern symmetry score considerably, we do not start a new pass and Algorithm 4 returns. Our default setting starts another pass, if the finishing pass has improved the pattern symmetry score by at least 5%. The effects of these short cuts are investigated empirically in Section 4.2.3.

Algorithm 5: UPDATEGAINS($G, \mathcal{C}, r$, cycleHeap)

Input : G, a bipartite graph G (corresponding to the matrix $\mathbf{A}$).
$\mathcal{C} = (r_1, r_2, c_1, c_2)$, a cycle which has been alternated.
r, a row vertex of $\mathcal{C}$; r is either r_1 or r_2.
cycleHeap, the priority queue of the length-four alternating cycles.

foreach $c' \in \text{adj}_G(r)$ **do**
  $mark(c') \leftarrow r$
foreach $r' \in \text{adj}_G(c_1)$ **do**
  $mark(r') \leftarrow c_1$
foreach $r' \in \text{adj}_G(c_2)$ **do**
  $mark(r') \leftarrow c_2$
foreach $\mathcal{C}' = (r'_1, c'_1, r'_2, c'_2) \in$ cycleHeap *where* $c'_1 \in \text{adj}_G(r)$ *or* $c'_2 \in \text{adj}_G(r)$ **do**
  initValue $\leftarrow$ gain($\mathcal{C}'$) form cycleHeap
  add $\leftarrow 0$
  if $mark(c'_2) = r$ *and* $mark(c'_1) \neq r$ `/* For` $r = r_1$`, Fig. 1a; otherwise Fig. 1b */`
  then
    if $mark(r'_1) = c_2$ **then**
      add $\leftarrow$ add $+1$
    if $mark(r'_1) = c_1$ **then**
      add $\leftarrow$ add -1
    if $mark(r'_2) = c_1$ **then**
      add $\leftarrow$ add $+1$
    if $mark(r'_2) = c_2$ **then**
      add $\leftarrow$ add -1
  else if $mark(c'_1) = r$ *and* $mark(c'_2) \neq r$ `/* For` $r = r_1$`, Fig. 1b; otherwise Fig. 1a */`
  then
    if $mark(r'_1) = c_1$ **then**
      value $\leftarrow$ add $+1$
    if $mark(r'_1) = c_2$ **then**
      value $\leftarrow$ add -1
    if $mark(r'_2) = c_2$ **then**
      value $\leftarrow$ add $+1$
    if $mark(r'_2) = c_1$ **then**
      value $\leftarrow$ add -1
  if $r = r_2$ **then**
    add $\leftarrow -$add
  if *add*$\neq 0$ **then**
    HeapUpdate($\mathcal{C}'$, initValue+add)

3.3 Run time analysis

We investigate the run time of Algorithm 4. The initialization steps up to Line 3 are straightforward and take time in $\mathcal{O}(n+\tau)$. Computing the gain of a cycle (r_1, c_1, r_2, c_2) using Eqs. (7) and (8) takes $\mathcal{O}(|\operatorname{adj}_G(r_1)| + |\operatorname{adj}_G(r_2)| + |\operatorname{adj}_G(c_1)| + |\operatorname{adj}_G(c_2)|)$ time. Since a vertex u can be in at most $|\operatorname{adj}_{G_f}(r_1)| - 1$ cycles, we conclude that the overall complexity of computing the initial gains is $\mathcal{O}\left(\sum_{u \in R \cup C}(|\operatorname{adj}_{G_f}(u)| - 1) \times (\operatorname{adj}_G(u) - 1)\right)$. The worst-case cost of building the priority queue on C_4 is $\mathcal{O}(\tau_f \log \tau_f)$, where τ_f is the number of nonzeros in $\mathbf{A}_f$. Therefore, the worst case computational complexity of the initialization steps is $\mathcal{O}\left(\tau_f \log \tau_f + \sum_{u \in X}(|\operatorname{adj}_{G_f}(u)| - 1) \times (|\operatorname{adj}_G(u)| - 1)\right)$. There are at most $\mathcal{O}(n)$ extract operations and $\mathcal{O}(\tau_f)$ deletions in a pass from the heap, which costs a total of $\mathcal{O}\left((n + \tau_f) \log \tau_f\right)$. The gain update operations at Line 6 take constant time for updating the gain of a cycle, and the total number of such updates is $\mathcal{O}(\tau)$. Since it takes $\mathcal{O}(\log \tau_f)$ time to update the heap, each pass is of time complexity $\mathcal{O}((n + \tau) \log \tau_f)$, in the worst case. We note that these run time bounds are pessimistic, and one should expect near linear time for each pass. In comparison, the gain computations at each pass in the earlier work [20] take $\mathcal{O}(\sum_{u \in X}(|\operatorname{adj}_G(u)| - 1) \times (|\operatorname{adj}_G(u)| - 1))$ time, on top of the operations performed on the heap, whose size is τ instead of τ_f.

4 Experiments

We first describe the data set and the experimental environment. Then, we present two sets of experiments. In the first set (Section 4.2), we investigate the proposed iterative improvement's performance with respect to the earlier work. While doing so, we also explore the parameter space of the proposed method. In the second set of experiments (Section 4.3), we evaluate the effects of the proposed method in the context of the direct solver MUMPS. We give detailed results in Appendix, and populate summaries in this section to facilitate the discussion.

4.1 Data set and environment

We created the data set as follows. We iterated over all real, square, unsymmetric matrices with a numerical symmetry value less than 0.95 from University of Florida Sparse Matrix Collection [10], and took all matrices with the following properties: (i) the number of rows is at least 10000 and at most 1000000; (ii) the number of nonzeros is at least 3 times larger than the number of rows, but smaller than 15000000; (iii) there is full structural rank. This set of matrices includes those unsymmetric matrices where MC64 preprocessing was found to be useful. Among these matrices, we discarded those that are binary and those that are combinatorial—in these matrices, the nonzero values are from a small set of integers. We selected at most five matrices from each family to remove any bias that might be arising from using a number of related matrices. Then, we took the largest irreducible block from these matrices, as any direct solver should process a decomposable matrix block by block for efficiency. There were a total of 136 matrices at the time of experimentation. We discarded two matrices whose largest blocks were of order less than 1000. We then applied MC64 on the largest blocks and discarded those matrices with a pattern symmetry ratio ($\textsc{SymScore}(\mathbf{AQ}_{\text{MC64}})/\tau$) larger than 0.90, as there is little potential improvement. This left us with 75 matrices at the end. Tables 7 and 8 contain these 75 matrices in alphabetical order. In these tables, we show the original pattern symmetry ratio, that obtained by MC64, and that

obtained by the proposed method (whose parameters are described later). As seen in this table, the geometric mean of the pattern symmetry ratios of the matrices in the data set is is 0.43; the geometric mean of the pattern symmetry ratios obtained by MC64 is 0.40; and the geometric mean of the pattern symmetry ratios obtained by the proposed method is 0.51. We note that MC64 can increase the symmetry ratio (mostly because replacing the zeros in the main diagonal), but in general this is not its objective.

We performed our tests on a machine with Intel(R) Xeon(R) CPU having a clock speed of 2.20GHz. We implemented the code for improving the symmetry in C and compiled with flag -O3; we call these functions through mex wrappers within Matlab. Since the run time of the proposed iterative improvement algorithm includes the term $\mathcal{O}\left(\sum_{u\in R\cup C}(|\operatorname{adj}_{G_f}(u)|-1)\times(|\operatorname{adj}_G(u)|-1)\right)$ in the complexity, one needs to be careful as $|\operatorname{adj}_G(u)|$ and $|\operatorname{adj}_{G_f}(u)|$ could be large. As is common in the standard ordering tools (for example AMD), we consider the pair of the ith row and the ith column of $\mathbf{A}$ as dense if $\max\{|\operatorname{adj}_G(r_i)|,|\operatorname{adj}_G(c_i)|\}\geq 5\times\sqrt{n}$. We remove the dense pairs from the matrix, symmetrize the rest and leave the dense row-column pairs as matched by MC64.

4.2 Comparisons with the earlier work

In this section, we investigate the improvements with respect to the earlier work [20]. More precisely, we investigate (i) the effect of the threshold in defining the filtered matrix $\mathbf{A}_f$ at Line 1 of Algorithm 3; (ii) the improvement in the run time of symmetrization achieved by using the proposed gain-updates; and (iii) the impact of the other significant components on the symmetry and the run time of the symmetrization.

4.2.1 Threshold scheme in building $\mathbf{A}_f$

The thresholding scheme at Line 1 of Algorithm 3 should affect the direct solver's performance. The larger the absolute values of the entries in $\mathbf{A}_f$, the smaller the chances that there would be numerical problems in factorizing $\mathbf{A}''$ defined in (4). On the other hand, the higher the number of nonzeros in $\mathbf{A}_f$, the higher the chances that one can improve the pattern symmetry ratio. Otherwise said, according to the entries allowed in the diagonal, the threshold value for pivoting (Line 5 of Algorithm 3) will change and thus smaller pivots may be used. This in turn can lead to more steps of iterative refinement. There is a trade-off to make. On the one side, we can allow all entries of $\mathbf{A}$ into $\mathbf{A}_f$ and hope to have improved performance in the direct solver. However, this ignores the numerical issues. On the other side, we can allow only those entries of $|\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c|$ that are equal to 1. In this case, we have observed that, in average, it does not provide any gain. In this subsection, we present experimental results that help us suggest a threshold value and use it in the remaining experiments. We suggest filtering a certain portion of entries into $\mathbf{A}_f$ to enable symmetrization, rather than filtering entries that are larger than a given threshold. While the suggested approach favors pattern symmetry, the second alternative favors numerically better pivots by limiting the smallest entry in the diagonal. We opted for the first one, as iterative refinement, when needed, is likely to recover the required precision.

We compare four thresholding schemes in Table 1 by giving the statistical indicators of the pattern symmetry ratio $\textsc{SymScore}(\mathbf{AQ})/\tau$, where $\mathbf{Q}$ is as defined in (5), for all 75 matrices. The first one "1" corresponds to using only those entries of $|\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c|$ that are 1. Since $\mathbf{Q}_{\text{Pat}}$ will allow only entries of absolute value 1 in the diagonal, this corresponds to using a permutation that could be obtained by MC64. The second thresholding scheme "half" uses the median value

Table 1 – Statistical indicators of the pattern symmetry ratios with different thresholding schemes at Line 1 of Algorithm 3. The column "1" corresponds to allowing only entries that are 1.0 into $\mathbf{A}_f$; "half" uses the median value as the threshold; "$1-e^{-1}$" allows $1-e^{-1} \approx 0.63$ of the entries of $\mathbf{A}$ into $\mathbf{A}_f$; "0" allows all entries of $\mathbf{A}$ into $\mathbf{A}_f$.

statistics	1	half	$1-e^{-1}$	0
min	0.08	0.10	0.12	0.13
max	0.89	0.93	0.94	0.96
geomean	0.40	0.47	0.51	0.58

of the entries of $|\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c|$ so that $\tau_f = \frac{\tau}{2}$. The third scheme "$1-e^{-1}$" allows the largest $1-e^{-1} \approx 0.63\tau$ entries of $|\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c|$ to be in $\mathbf{A}_f$. The last scheme, "0", allows all entries of $|\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c|$ to be in $\mathbf{A}_f$. This last alternative amounts using the earlier work [20]. A few observations are in order. First, using the first scheme "1" and not calling ImproveSymmetry, that is $\mathbf{Q} = \mathbf{Q}_{\text{MC64}}$, led to nearly the same results—all three statistical indicators agreed to the second digit (see the geometric means in Tables 1 and 8). Second, the larger the number of allowed entries, the higher the pattern symmetry ratio as expected. Allowing half of the entries into $\mathbf{A}_f$ improves the pattern symmetry ratio (with respect to using $\mathbf{Q}_{\text{MC64}}$ only) by 18%, on the average; allowing $1-e^{-1}$ of the entries improves the pattern symmetry ratio by 28%, and finally allowing all entries reaches on the average 45% improvements with respect to using $\mathbf{Q}_{\text{MC64}}$ only.

In additional experiments which are not shown here, the geometric means of the ratios of the run time of IterativeImprove to that of MC64 were 0.13, 0.42, 0.64 and 1.36 with the four thresholding schemes. That is, the thresholding scheme "0" makes IterativeImprove about 36% slower than MC64, while the thresholding scheme $1-e^{-1}$ makes IterativeImprove about 36% faster than MC64. The other schemes are even faster. Given this, we find the scheme $1-e^{-1}$ satisfactory both in terms of the pattern symmetry ratio and the run time of IterativeImprove. Therefore, we keep the thresholding scheme $1-e^{-1}$ as the default value (the remaining experiments use this value). We tried a few numbers for thresholding and found that a value close to the average of 0.50 and 0.75 and close to the higher end strikes a good balance for symmetry and run time. That is why we used $1-e^{-1}$. The performance is not sensitive to small changes in the value; the geometric mean of the pattern symmetry ratio with 0.63 and 0.64 are also 0.51.

4.2.2 Effects of the gain updates

Using the proposed gain-updates subroutine improves the run time of earlier work [20]. As expected, the gain-update technique does not change the pattern symmetry ratio—the pattern symmetry ratios obtained by IterativeImprove with and without the gain-updates agreed up to four digits. A difference is possible, as when there are ties (in the key values of the cycles in the heap), the two implementations can choose different cycles and hence explore different regions of the search space. The geometric mean of the run time of IterativeImprove with the gain-updates is 0.06 seconds; without the gain-updates it is 0.26 seconds. We conclude that the gain updates makes the code much faster. This is seen clearly if we look at a few instances closely in Table 2.

The three instances on which the run time of IterativeImprove with the re-computation of the gains were the longest (the matrices `av41092`, `ohne2`, and `PR02R`) are in Table 2. These three instances were among the only four matrices on which IterativeImprove with the gain-updates

Table 2 – Effect of the gain update (with respect to re-computing them) on the run time of IterativeImprove (Algorithm 4) in seconds. The three longest run time of IterativeImprove without the gain updates were on the first three matrices (out of 75). The run time of IterativeImprove with the gain updates was longer than one second only for the last three matrices (out of 75). We also give the geometric mean of the run time over all 75 matrices in seconds.

	gains	
matrix	re-computed	updated
av41092	59.94	0.04
ohne2	273.22	5.01
PR02R	52.02	7.53
pre2	8.54	1.62
geomean (all 75 matrices)	0.26	0.06

Table 3 – Statistical indicators of the pattern symmetry ratios obtained by IterativeImprove initialized with MC64, that is no UB1 computation (the column $\mathcal{M}_0 = \mathcal{M}_{MC64}$); the statistical indicators of the run time of computing a perfect matching attaining UB1, and the statistical indicators of the pattern symmetry ratio obtained by IterativeImprove when initialized with UB1 attaining matchings (under the label $\mathcal{M}_0 = \mathcal{M}_{UB1}$).

	$\mathcal{M}_0 = \mathcal{M}_{MC64}$	$\mathcal{M}_0 = \mathcal{M}_{UB1}$	
statistics	SymScore($\mathbf{AM}_1$)/τ	UB1 time (s)	SymScore($\mathbf{AM}_1$)/τ
min	0.10	0.01	0.12
max	0.94	273.22	0.94
geomean	0.48	0.37	0.51

took longer than one second—we added `pre2`, which was the fourth one, to the table. As seen from these instances, IterativeImprove has sometimes a long run time when UpdateGain is not used; with UpdateGain, the longest run time is 7.53 seconds. All these numbers confirm that the gain-updates has large impact in the run time, on average IterativeImprove is faster than MC64 (0.06 vs 0.10, see the geometric mean in Table 9).

4.2.3 Effects of other components

As discussed in Section 3, the function ImproveSymmetry starts with an initial perfect matching. In order to implement UB1 as an initial perfect matching, we used MC64 with the maximum weight as the objective on the bipartite graph of $\mathbf{A}_f$ using the edge weights (2). Any perfect matching algorithm could be used for this purpose. We compared using an UB1 attaining matching with that of using the already computed maximum product perfect matching (the second alternative corresponds to initializing IterativeImprove on $\mathbf{AQ}_{\text{MC64}}$ with the identity matching as the initial choice). We give the comparisons between these two alternatives in Table 3, for all 75 matrices.

As seen in Table 3, the geometric mean of the pattern symmetry ratio with UB1 as initial matching is 0.51 while that of not using it is 0.48. By looking at the geometric mean line, we

Table 4 – The run time of MC64 to compute a maximum product perfect matching, the time to compute a maximum weighted perfect matching for UB1 (using MC64 and `csa_q`), and the run time of IterativeImprove, "ItImp." in seconds. These are the five matrices where the run time of MC64 was the longest while computing UB1.

		UB1		
matrix	$MC64_\pi$	MC64	csa_q	ItImp.
ohne2	0.63	273.22	14.83	5.01
Chebyshev4	0.45	58.28	8.62	0.92
PR02R	51.96	52.02	17.35	7.53
laminar_duct3D	13.86	26.30	2.22	0.81
stomach	0.28	23.02	10.16	0.34
geomean (all 75 matrices)	0.10	0.37	0.53	0.06

see that in general MC64 runs fast to compute a maximum weighted matching on the bipartite graph of $\mathbf{A}_f$ while defining UB1. However, there are cases where MC64 can take large time while computing UB1. In Table 4, we give the run time of MC64 to compute a maximum product perfect matching $MC64_\pi$, to compute a maximum weighted perfect matching for UB1, and the run time of IterativeImprove on five matrices where the run time of MC64 was the longest while computing UB1 (the results for all matrices are in Table 9). This table also includes two of the longest run times for MC64 while computing a maximum product perfect matching. As seen in this table, the run time of computing UB1 with MC64 is about nearly four times slower in general than computing a maximum product matching with MC64 (geometric mean of 0.37 versus 0.10). MC64 implements a general purpose maximum weighted perfect matching algorithm (of time complexity $\mathcal{O}(n\tau \log n)$). However, in the instances for computing UB1, we have integer edge weights coming from a small set. With this observation, we have looked at algorithms whose run time depends on the maximum weight of an edge. One of the fastest algorithms for this case is called `csa_q` [13] and has a run time complexity of $\mathcal{O}(\sqrt{n}\tau \log Wn)$, where W is the maximum edge weight. There are a number of variants of this algorithm. We tested the default one and found it slower in general than MC64; the geometric mean of `csa_q`'s run time was 0.53 (see Table 4 for a summary and Table 9 for individual results). In the five instances of Table 4, `csa_q` is much faster than MC64 while computing UB1. However, on an instance (the matrix `largebasis`), it took more than an hour. Both algorithms perform well on average, but the maximum was much worse with `csa_q`. For these reasons we keep MC64 as the default solver for UB1. Notice that if prohibitive run times are likely (we are not aware of any study to provide a rule of thumb here), one can skip computing UB1, and call IterativeImprove using the matching found by MC64 (at the first step of Algorithm 3). This corresponds to using the thresholding scheme "1" in constructing $\mathbf{A}_f$ and in general does not provide improvement in the pattern symmetry ratio with respect to MC64.

We tested the effect of using the practical improvements discussed in Section 3.2. We report these results without giving tables to save space. As the base case, we took the thresholding scheme $1 - e^{-1}$ and used UB1 for initial matching. We compared the pattern symmetry ratio and the run time of IterativeImprove in five iterations without any short cuts to that with the default values for the short cuts. Without the short cuts, the geometric mean of the run time of

IterativeImprove was 0.27 seconds; this is still acceptable but 2.75 times slower than MC64. The improvement in the final pattern symmetry ratio was only slightly better (0.514 vs 0.512). Therefore, we find the suggested short cuts worth taking always.

We next looked at the cases where there were dense rows/columns which the proposed method removed, optimized with respect to what is remaining, and put the removed rows/columns back into the matrix. There were 11 such matrices. We report the geometric means here, without giving tables. The pattern symmetry ratios agreed to the three significant digits; and the geometric mean of the ratio of the run times were 0.29 (with respect to the 11 matrices; in others the methods are the same) in favor of removing dense rows/columns. Therefore, we suggest applying this technique always.

4.3 With MUMPS

We now observe the effects of the proposed preprocessing method in the context of the direct solver MUMPS [2, 4] version 5.0.1. ImproveSymmetry uses the thresholding scheme $1 - e^{-1}$, UB1 for initializing IterativeImprove, and all the tunings discussed so far. For the fill-reducing ordering step, we used MeTiS [15] version 5.1.0. Since we order and scale the matrices beforehand, we set MUMPS to not to preprocess for fill-reducing and column permutation. Apart from these, we used MUMPS with its default settings for all parameters concerning numerics, except the pivoting threshold which we set in Algorithm 3 at Line 4. In particular, the post-processing utilities are kept active for better numerical accuracy (which includes iterative refinement procedures that can increase the run time).

We compare the effects of $\mathbf{Q}$ (computed by ImproveSymmetry), with those of $\mathbf{Q}_{\text{MC64}}$ (computed by MC64) and with those computed by the earlier study [20] (noted $\mathbf{Q}_{01}$). In all three cases, we preprocessed for fill-reducing ordering and column permutation before calling MUMPS as we did for $\mathbf{Q}$, and set all other parameters to their default value. We are going to look at four measurements: the run time of MeTiS, the real space and the operation count during factorization, and the total time spent by MUMPS (that is, the total time in analysis without preprocessing, factorization, and solution with iterative refinement when necessary). The run time of MUMPS depends also on the compiler, the hardware, and the third party libraries (in particular BLAS), while the real space and the operation count are absolute figures. Since this is so, we expect that the improvements in the real space and the operation count to be usually more than the improvements in the run time, and observable in other settings.

Using $\mathbf{Q}_{01}$ ignores numerical issues. In our experiments with the 75 matrices, MUMPS returned with an error message (related to allocated memory) for 29 matrices with the default settings. This errors could possibly fixed with updating the memory parameters and re-running until a successful factorization is obtained. This will be time consuming, and will not achieve the ideal of using the direct solvers as a black-box. We conclude that $\mathbf{Q}_{01}$ is not a viable alternative for preprocessing unsymmetric matrices for direct methods on its own.

We now document the benefits of using ImproveSymmetry's column permutation $\mathbf{Q}$ in MUMPS with respect to using $\mathbf{Q}_{\text{MC64}}$ only. We consider the improvements of less than 10% in the pattern symmetry ratio to be insignificant to have impact on the direct solver; in which case it is advisable to use $\mathbf{Q}_{\text{MC64}}$ for better numerical properties. That is why we work with a subset of the 75 matrices in which ImproveSymmetry improved the pattern symmetry ratio by at least 10%. This set contained 32 matrices and in the following we report results on this set. Appendix

Table 5 – Geometric mean of some measurements on 32 matrices, in which the proposed method had at least 10% improvement in the symmetry ratio with respect to MC64. Those of MC64 are given in absolute terms in column $\mathbf{Q}_{\text{MC64}}$; those of ImproveSymmetry are given as the geometric mean of the ratios to MC64's results in column $\mathbf{Q}$.

measurement	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}$
pattern symmetry ratio	0.25	1.72
MeTiS time	0.77	0.93
Real space	9.55e+06	0.89
Operation count	3.00e+09	0.82
MUMPS time	2.73	0.88

contains detailed results on these 32 matrices in Tables 10 and 11.

We first note that using $\mathbf{Q}$ instead of $\mathbf{Q}_{\text{MC64}}$ means that there are potentially smaller pivots on the diagonal. Therefore, one should enable iterative refinement in MUMPS for guarding against numerical problems. As seen in Table 11, using $\mathbf{Q}_{\text{MC64}}$ necessitated one step of iterative refinement for 3 matrices, whereas using $\mathbf{Q}$ required iterative refinement on 22 matrices (two steps on four of them, and one step in others). At the end, the backward error metrics (`RINFOG(7-8)`) reported by MUMPS were always satisfactory. Since we allow smaller entries, this could also affect the pivoting in MUMPS. In Table 11, we give the number of off-diagonal pivots and the number of delayed pivots when using $\mathbf{Q}$ and $\mathbf{Q}_{\text{MC64}}$. First observation is that both $\mathbf{Q}$ and $\mathbf{Q}_{\text{MC64}}$ have off-diagonal pivots and delayed pivots in many cases. In most of the cases, the number of off-diagonal pivots is larger with $\mathbf{Q}$, whereas the number of delated pivots is smaller. That is why it is important to report the real space and operation counts after the factorization, as the analysis phase underestimates these quantities. There are some cases in which the quantities are improved with $\mathbf{Q}$. This is most probably because of the fact that $\mathbf{Q}$ allows smaller pivots (which could lead to numerical inaccuracy to be fixed with iterative refinement).

The performance difference between using $\mathbf{Q}_{\text{MC64}}$ and $\mathbf{Q}$ for solving a linear system is given Table 6 for different stages of the whose process of solving a linear system. Table 10 contains the individual results for the 32 matrices for the four metrics (MeTiS time, real space, operation count, and MUMPS time) for which we expect improvements with the proposed $\mathbf{Q}$. The results are also summarized in Table 5 and supplemented with the performance profiles given in Fig. 3. We refer to Table 6 to comment on the total time spent in solving a linear system and the overhead associated with the proposed method. First, looking at the geometric mean of the time associated with $\mathbf{Q}_{\text{MC64}}$ (which is the sum of MC64's run time, MeTiS's run time, and MUMPS's run time) and $\mathbf{Q}$ (on top of those for $\mathbf{Q}_{\text{MC64}}$, the time to compute UB1 and the run time of IterativeImprove are also included), we see that the total time is longer with $\mathbf{Q}$ (4.56 seconds vs. 5.07 seconds on average). Once MC64 is applied, the remaining time is still longer with $\mathbf{Q}$ (4.25 vs 4.78), which we investigate more as the proposed symmetry improving algorithm enters into the process after MC64 (so that we can highlight what can be gained). We look at the next step in preprocessing, in which UB1 is computed, but IterativeImprove is to be applied. Here we see that $\mathbf{Q}$ has shorter remaining time ($\mathbf{Q}_{\text{MC64}}$'s remaining time is 4.25 which is the sum of the run time of MeTiS and MUMPS, $\mathbf{Q}$'s remaining time is 4.04 which is the sum of the run time of IterativeImprove, MeTiS, and MUMPS). From this moment on, $\mathbf{Q}$'s remaining time is shorter, where MUMPS's run time (analysis without preprocessing, factorization, and solution with iterative refinement) reduces by 12%. In

Table 6 – The total run time of solving a linear system at different steps with $\mathbf{Q}_{\text{MC64}}$ and the proposed $\mathbf{Q}$. For $\mathbf{Q}_{\text{MC64}}$, the table reports the total time, the time spent after MC64, and the time spent in MUMPS (analysis without preprocessing, factorization, and solution with iterative refinement). For $\mathbf{Q}$, the table reports the total time, the time spent after MC64, the time spent after computing UB1, the time spent after iterative improvement (ItImp.), and the time spent in MUMPS.

	$\mathbf{Q}_{\text{MC64}}$			$\mathbf{Q}$				
matrix	MUMPS MC64 MeTiS	MUMPS MeTiS	MUMPS	MUMPS MC64 UB1 ItImp MeTiS	MUMPS UB1 ItImp MeTiS	MUMPS ItImp MeTiS	MUMPS MeTiS	MUMPS
av41092	9.37	7.31	5.33	11.55	9.49	7.70	7.03	5.07
bbmat	15.73	15.56	13.76	22.82	22.65	15.79	14.80	13.01
circuit_4	0.65	0.63	0.28	0.72	0.70	0.66	0.63	0.27
cz10228	0.14	0.13	0.04	0.20	0.19	0.16	0.14	0.05
cz20468	0.30	0.28	0.09	0.39	0.37	0.32	0.28	0.10
cz40948	0.61	0.57	0.17	0.81	0.77	0.67	0.59	0.19
fd15	0.19	0.18	0.08	0.21	0.20	0.18	0.18	0.08
g7jac160	16.45	16.14	15.10	17.78	17.47	16.95	16.88	15.87
g7jac180	20.56	20.20	18.98	21.97	21.61	21.01	20.93	19.73
g7jac180sc	21.02	20.18	18.98	20.31	19.47	18.52	18.40	17.20
g7jac200	24.34	23.93	22.59	25.51	25.10	24.41	24.32	22.98
g7jac200sc	24.98	23.93	22.58	22.90	21.85	20.88	20.76	19.42
largebasis	11.13	10.47	2.14	13.04	12.38	11.02	10.54	2.41
lhr17c	0.14	0.13	0.05	0.20	0.19	0.15	0.14	0.06
lhr34c	0.40	0.29	0.11	0.53	0.42	0.36	0.34	0.15
lhr71c	0.42	0.30	0.12	0.52	0.40	0.35	0.33	0.14
mac_econ_fwd500	21.94	20.26	15.94	23.50	21.82	19.32	19.23	14.96
matrix_9	106.78	106.25	104.29	124.63	124.10	118.48	118.22	116.33
ohne2	156.94	156.31	148.09	431.11	430.48	157.26	152.25	146.63
onetone1	1.71	1.67	1.00	2.01	1.97	1.56	1.49	0.97
powersim	0.11	0.10	0.04	0.14	0.13	0.11	0.10	0.04
PR02R	150.91	98.95	91.23	171.34	119.38	67.36	59.83	55.06
pre2	116.11	115.40	104.94	141.17	140.46	131.92	130.30	120.16
sinc15	18.12	18.03	17.44	17.13	17.04	16.85	16.48	15.89
sinc18	48.85	48.68	47.67	54.82	54.65	54.21	53.46	52.49
std1_Jac2	2.09	2.04	1.64	1.90	1.85	1.48	1.27	0.93
std1_Jac3	2.34	2.28	1.85	2.04	1.98	1.62	1.36	1.02
twotone	6.80	6.71	5.44	2.89	2.80	2.58	2.37	1.17
viscoplastic2	1.91	1.86	0.98	1.59	1.54	1.44	1.40	0.59
Zd_Jac2	3.34	3.27	2.69	3.39	3.32	2.77	2.48	1.96
Zd_Jac3	3.40	3.33	2.71	3.03	2.96	2.42	2.05	1.55
Zd_Jac6	2.99	2.92	2.32	3.16	3.09	2.54	2.22	1.69
geomean	4.56	4.25	2.73	5.07	4.78	4.04	3.79	2.42

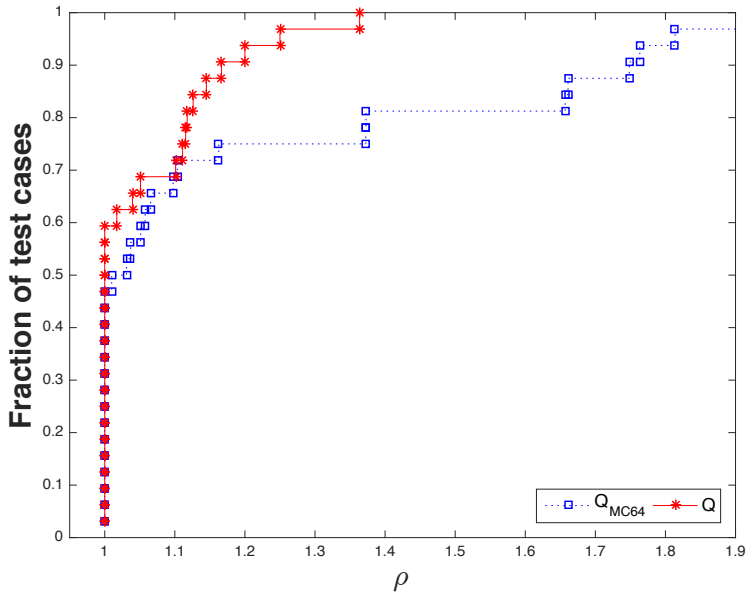


Figure 3 – Performance profiles of using $\mathbf{Q}_{\mathrm{MC64}}$ and the proposed $\mathbf{Q}$ in the run time of MUMPS with respect to the 32 matrices in which there were at least 10% improvement in the symmetry ratio.

Table 6 we see a remarkable improvement in MUMPS time with $\mathbf{Q}$. This particular case is most probably due to the fact that the final pattern symmetry ratio (see Table 8) on this matrix is 0.94 (up from 0.72), and the matrix is nearly symmetric. Furthermore, there are improvements in the real space and operation count, only one more step of iterative refinement, and smaller number of off-diagonal and delayed pivots. We note that there are other instance (see `pre2`) where we have improved number of off-diagonal and delayed pivots with about the same ratios, without gain in the real-space, with increase in the operation count, and longer MUMPS time.

The performance profile (Fig. 3) shows the percentage of test cases in which the run time of MUMPS (analysis without preprocessing, factorization, and solution with iterative refinement) with one permutation was no shorter than its run time with the other permutation by a factor of ρ, at a given value of ρ. Hence, the higher a profile, the better the permutation is. As seen in the figure, $\mathbf{Q}$'s profile is not worse than $\mathbf{Q}_{\mathrm{MC64}}$ for all ρ. In this figure, we also see that $\mathbf{Q}$ is more robust than $\mathbf{Q}_{\mathrm{MC64}}$ in the sense that the worst case of $\mathbf{Q}$ has a smaller ρ than that of $\mathbf{Q}_{\mathrm{MC64}}$. We note that looking at Table 10, we see that $\mathbf{Q}$ results in improvements in MeTiS time, real space, and operation count in most of the cases. The total run time of MUMPS shows a global improvement of 12% in the geometric mean, while showing differing performance for different matrices (in 13 cases the run time with $\mathbf{Q}_{\mathrm{MC64}}$ is shorter).

We have investigated the estimated space and operation count with $\mathbf{Q}$ and $\mathbf{Q}_{\mathrm{MC64}}$ (see the column $\frac{\text{real space}}{\text{est space}}$ in Table 11). With both $\mathbf{Q}$ and $\mathbf{Q}_{\mathrm{MC64}}$, the estimated space and the real space are close to each other. The geometric means of $\mathbf{Q}$'s ratio to $\mathbf{Q}_{\mathrm{MC64}}$'s in the estimated case was the same as in the real case. This highlights that the smaller fill-in with $\mathbf{Q}$ is not due to less pivoting.

Given the discussion above and by referring to Table 5, we arrive to the following summary for the 32 matrices (the geometric mean of the pattern symmetry ratios is increased from 0.25 to 0.43,

in this data set):

- The run time of MeTiS is reduced by 7%.
- The real space used by MUMPS (this is not symbolic, and computed after factorization) is reduced by 11%.
- The operation count in MUMPS (again computed after factorization) is reduced by 18%.
- The run time of MUMPS (that is, the total of analysis without preprocessing, factorization, and the solution time using iterative refinement) is reduced by 12%.

This analysis shows that considerable reductions in the real space and operation count is possible by increasing the symmetry. This entails reductions in the run time of the remaining steps in solving a linear system. However, the total time for solving linear system can increase, if we include the run time of all the preprocessing steps. In particular, computing UB1 is more expensive than computing a maximum product perfect matching with MC64. This step should be improved for gains in the run time. For example, if a UB1 attaining perfect matching was readily available to IterativeImprove, 5% reduction in the run time was achievable (see Table 6, where the geometric mean of the run time of MUMPS+MeTiS for $\mathbf{Q}_{\text{MC64}}$ is 4.25 and the geometric mean of the run time of MUMPS+IterativeImprove+MeTiS for $\mathbf{Q}$ is 4.04).

5 Conclusion

We considered the problem of finding column permutations of sparse matrices with two objectives. One of the objectives is to have large diagonal entries. The second objective is to have a large pattern symmetry. Both of the objectives were addressed independently. Duff and Koster address the first objective within MC64 [12] with the maximum product perfect matching. This is based on a more theoretical work by Olschowka and Neumaier [18] and has been proven to be very helpful for direct solvers. Uçar [20] addresses the second objective and highlights that the problem is NP-complete. While MC64 totally ignores the pattern symmetry, the heuristic for the pattern symmetry totally ignores the numerical issues. Since the second objective amounts to an NP-complete problem, heuristics are needed for finding a single permutation trying to achieve both of the objectives. We proposed an iterative improvement based approach which can trade the first objective to have improved symmetry. We proposed algorithmic improvements to the existing work and demonstrated the effects of the permutations within the direct solver MUMPS. In particular, with a set of matrices in which at least 10% improvement in the pattern symmetry was obtained, the memory and operation count requirement of MUMPS are improved by 11% and 18%. We used MUMPS in such a way that all preprocessing were done before hand. This results in 12% improvement, on average, in the run time of MUMPS (analysis with no preprocessing, factorization, and solution with possibly iterative improvement). By including all the preprocessing steps, we see that the proposed method needs improvements in its run time for realizing the gains in the total run time of solving a linear system.

The proposed method has two components: initialization and iterative improvement. The first component uses a maximum weighted perfect matching (UB1) to initialize the second component. We used MC64 for this purpose. The second component is carefully implemented to be fast. MC64's performance for UB1 turned out to be inferior than what is observed for solving the

maximum product perfect matching problem. Other algorithms could be used for computing UB1 attaining perfect matchings. We tried one such algorithm, `csa_q`, but could not get a consistent improvement with respect to MC64's run time, and also observed a very long run time for an instance. A mechanism to make an automatic choice between these two matching codes would be very useful. Since using UB1 attaining matchings is a heuristic, perhaps fast heuristics can be used to compute a perfect matching, approximating UB1. Alternatively, one can come up with initialization algorithms that are not related to UB1 for improved run time. We leave the identification and the use of better algorithms for initializing the iterative improvement method as a future work.

The proposed method is hard to parallelize like many weighted matching algorithms in parallel computing systems. Furthermore, the proposed method is used under the premise that a maximum product perfect matching has already been computed with the approach used in MC64. Dependence on such a matching sets up another obstacle in parallelizing the preprocessing step with the proposed method.

Appendix: Detailed results

In this appendix, we give some large tables in order to facilitate the discussion in the experiments.

First we give the list of matrices used in the experiments in the alphabetical order in Tables 7 and 8. In these tables, we give the order n of the largest irreducible block which are used in the experiments, the number of nonzeros τ, the original pattern symmetry ratio, the pattern symmetry ratio after MC64, and the final pattern symmetry ratio obtained by the proposed algorithm.

Table 9 gives the run time of various perfect matching codes and also that of IterativeImprove for all 75 matrices.

Tables 10 and 11 contain raw results with MUMPS on 32 matrices for which using $\mathbf{Q}$ resulted in more than 10% improvement in the pattern symmetry ratio with respect to using $\mathbf{Q}_{\mathrm{MC64}}$. Table 10 contains the four metrics in which $\mathbf{Q}$ should provide improvements, and Table 11 give further measurements returned by MUMPS (the number of iterative refinement steps, the number of off-diagonal and delayed pivots, and the ratio of the estimated space to the real space).

Acknowledgement

We thank Jean-Yves L'Excellent and Guillaume Joslin for their help with the MUMPS experiments. This work is supported in part by French National Research Agency (ANR) project SOLHAR (ANR-13-MONU-0007). We also thank Kamer Kaya for his feedback on an earlier version of this paper.

References

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[2] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

Table 7 – Matrices used in the experiments in alphabetical order (first 38 matrices). **A** is the original matrix, permuted with MC64 to obtain $\mathbf{AQ}_{\mathrm{MC64}}$ and with the proposed approach to obtain **AQ**. The parameters for **Q** are set as in the numerical experiments of Section 4.3.

			Symmetry ratio		
matrix	n	τ	**A**	$\mathbf{AQ}_{\mathrm{MC64}}$	**AQ**
2D_54019_highK	46863	477931	0.73	0.73	0.75
3D_51448_3D	44822	521160	0.68	0.69	0.74
av41092	41086	1683890	0.10	0.08	0.12
barrier2-10	115190	2155351	0.65	0.81	0.81
barrier2-11	115190	2155351	0.65	0.81	0.81
barrier2-12	115190	2155351	0.65	0.81	0.81
bayer01	48603	237759	0.27	0.26	0.27
bayer02	11710	54254	0.28	0.28	0.28
bayer04	12359	62900	0.30	0.31	0.33
bayer10	10803	62238	0.24	0.22	0.24
bbmat	38744	1771722	0.54	0.50	0.59
bips07_1693	9918	42120	0.56	0.81	0.84
bips07_1998	11421	54517	0.67	0.87	0.90
Chebyshev4	68121	5377761	0.31	0.31	0.31
circuit_3	7607	34024	0.66	0.86	0.86
circuit_4	52005	255068	0.91	0.78	0.90
crashbasis	160000	1750416	0.59	0.59	0.59
cz10228	10227	102863	0.49	0.49	0.65
cz20468	20467	206063	0.49	0.49	0.65
cz40948	40947	412136	0.49	0.49	0.65
e40r0100	17281	553562	0.33	0.89	0.89
epb1	14734	95053	0.77	0.77	0.77
epb2	25228	175027	0.72	0.72	0.72
epb3	84617	463625	0.73	0.73	0.73
fd15	10645	41767	0.31	0.29	0.31
fd18	15367	60487	0.30	0.28	0.31
g7jac160	45154	555601	0.10	0.10	0.19
g7jac180	50814	630821	0.10	0.10	0.19
g7jac180sc	50814	630821	0.10	0.10	0.19
g7jac200	56474	706030	0.10	0.10	0.19
g7jac200sc	56474	706030	0.10	0.10	0.18
garon2	13535	373235	1.00	0.83	0.83
ibm_matrix_2	44822	521160	0.68	0.69	0.74
Ill_Stokes	20896	191368	0.99	0.30	0.30
inlet	11400	322253	0.63	0.63	0.64
invextr1_new	30412	1793881	0.97	0.86	0.89
laminar_duct3D	54149	3410525	0.24	0.56	0.57
largebasis	440020	5240084	1.00	0.11	0.19

Table 8 – Matrices used in the experiments in alphabetical order (last 37 matrices), continuing from Table 7.

			Symmetry ratio		
matrix	n	τ	**A**	$\mathbf{AQ}_{\text{MC64}}$	**AQ**
lhr17c	3664	76936	0.44	0.12	0.38
lhr34	4586	53502	0.32	0.21	0.23
lhr34c	7663	173683	0.42	0.29	0.32
lhr71	4586	53502	0.32	0.21	0.23
lhr71c	7663	173683	0.45	0.29	0.32
mac_econ_fwd500	206467	1273187	0.22	0.19	0.21
majorbasis	160000	1750416	0.59	0.59	0.59
matrix-new_3	78672	830476	0.73	0.73	0.78
matrix_9	99372	1193348	0.68	0.68	0.76
mc2depi	525825	2100225	0.25	0.25	0.25
mimo8x8_system	7823	37380	0.60	0.85	0.88
ohne2	181343	6869939	0.84	0.65	0.85
onetone1	32211	326852	0.37	0.44	0.51
onetone2	32211	213896	0.46	0.59	0.65
para-4	153226	2930882	0.67	0.83	0.83
powersim	12239	53479	0.79	0.70	0.79
PR02R	160599	8176944	0.95	0.72	0.94
pre2	629628	5757273	0.34	0.59	0.74
rajat21	397153	1820287	0.64	0.79	0.79
rajat29	629328	3733109	0.41	0.88	0.88
rim	22559	1014938	0.28	0.54	0.55
shermanACb	17583	82497	0.44	0.87	0.87
shyy161	25440	152001	0.83	0.83	0.83
sinc15	10880	526566	0.27	0.27	0.41
sinc18	15650	913548	0.27	0.27	0.40
std1_Jac2	9093	389158	0.25	0.16	0.61
std1_Jac3	9093	414305	0.26	0.15	0.66
stomach	213360	3021648	0.86	0.86	0.86
torso1	116158	8516500	0.43	0.43	0.43
twotone	105740	777549	0.41	0.64	0.72
viscoplastic2	32769	381326	0.61	0.13	0.44
ww_vref_6405	7781	37272	0.60	0.85	0.88
Zd_Jac2	11230	555374	0.27	0.14	0.60
Zd_Jac3	11230	586278	0.27	0.12	0.63
Zd_Jac6	11230	573470	0.28	0.13	0.59
zeros_nopss_13k	7081	35566	0.58	0.86	0.89
Zhao2	33861	166453	0.94	0.27	0.27
geomean (over all 75 matrices)			0.43	0.40	0.51

Table 9 – The run time of MC64 for computing a maximum product matching ("$MC64_\pi$."), for computing a perfect matching for UB1, the run time of csa_q for computing UB1, and the run time of IterativeImprove under the column"ItImp."

		UB1					UB1		
matrix	$MC64_\pi$	MC64	csa_q	ItImp.	matrix	$MC64_\pi$	MC64	csa_q	ItImp.
2D_54019_highK	0.09	0.69	3.82	0.04	lhr17c	0.01	0.04	0.08	0.01
3D_51448_3D	0.43	1.37	0.34	0.09	lhr34	0.01	0.02	0.03	0.00
av41092	2.06	1.79	0.75	0.67	lhr34c	0.11	0.06	0.13	0.02
barrier2-10	0.27	14.99	4.08	0.40	lhr71	0.01	0.02	0.02	0.00
barrier2-11	0.26	14.96	4.18	0.45	lhr71c	0.12	0.05	0.15	0.02
barrier2-12	0.26	15.02	3.94	0.41	mac_econ_fwd500	1.68	2.50	24.38	0.09
bayer01	0.06	0.06	0.11	0.01	majorbasis	0.15	3.93	12.48	0.11
bayer02	0.01	0.01	0.02	0.00	matrix_9	0.53	5.62	2.10	0.26
bayer04	0.02	0.02	0.05	0.00	matrix-new_3	0.58	1.14	1.86	0.13
bayer10	0.02	0.02	0.11	0.00	mc2depi	0.25	7.20	2.81	0.14
bbmat	0.17	6.86	2.23	0.99	mimo8x8_system	0.01	0.01	0.01	0.01
bips07_1693	0.01	0.01	0.01	0.00	ohne2	0.63	273.22	14.83	5.01
bips07_1998	0.01	0.01	0.01	0.00	onetone1	0.04	0.41	0.14	0.07
Chebyshev4	0.45	58.28	8.62	0.92	onetone2	0.03	0.12	0.11	0.03
circuit_3	0.00	0.01	0.01	0.00	para-4	0.30	16.58	6.47	0.56
circuit_4	0.02	0.04	0.08	0.03	powersim	0.01	0.02	0.02	0.01
crashbasis	0.14	7.27	6.72	0.39	PR02R	51.96	52.02	17.35	7.53
cz10228	0.01	0.03	0.14	0.02	pre2	0.71	8.54	7.93	1.62
cz20468	0.02	0.05	0.62	0.04	rajat21	0.18	0.41	1.16	0.24
cz40948	0.04	0.10	2.78	0.08	rajat29	0.36	0.50	0.83	0.25
e40r0100	0.11	0.28	0.26	0.09	rim	0.15	0.56	0.50	0.29
epb1	0.01	0.03	0.57	0.00	shermanACb	0.01	0.03	0.03	0.01
epb2	0.02	0.05	0.19	0.01	shyy161	0.01	0.12	2.27	0.01
epb3	0.04	0.19	4.46	0.02	sinc15	0.09	0.19	0.09	0.37
fd15	0.01	0.02	0.02	0.00	sinc18	0.17	0.44	0.19	0.75
fd18	0.01	0.03	0.03	0.00	std1_Jac2	0.05	0.37	0.31	0.21
g7jac160	0.31	0.52	1.67	0.07	std1_Jac3	0.06	0.36	0.58	0.26
g7jac180	0.36	0.60	2.62	0.08	stomach	0.28	23.02	10.16	0.34
g7jac180sc	0.84	0.95	3.96	0.12	torso1	4.25	0.77	4.96	0.08
g7jac200	0.41	0.69	3.21	0.09	twotone	0.09	0.22	0.27	0.21
g7jac200sc	1.05	0.97	3.91	0.12	viscoplastic2	0.05	0.10	0.08	0.04
garon2	0.03	0.22	0.20	0.08	ww_vref_6405	0.01	0.01	0.01	0.00
ibm_matrix_2	0.40	1.21	0.36	0.10	Zd_Jac2	0.07	0.55	1.17	0.29
Ill_Stokes	0.03	0.19	0.15	0.01	Zd_Jac3	0.07	0.54	0.95	0.37
inlet	0.12	0.19	0.72	0.05	Zd_Jac6	0.07	0.55	1.32	0.32
invextr1_new	1.81	5.73	2.17	0.90	zeros_nopss_13k	0.01	0.01	0.01	0.00
laminar_duct3D	13.86	26.30	2.22	0.81	Zhao2	1.38	0.40	0.45	0.01
largebasis	0.66	1.36	4620.32	0.48					
geomean (all 75 matrices)						0.10	0.37	0.53	0.06

Table 10 – Individual results for the 32 matrices for the four metrics (MeTiS time, real space, operation count, and MUMPS time) for which we expect improvements with the proposed $\mathbf{Q}$ with respect to $\mathbf{Q}_{\text{MC64}}$. "MeTiS time" refers to the ordering time with MeTiS; "Real space" and "Op. count" (the operation count) are reported by MUMPS after factorization (they are not symbolic results), and the run time.

	MeTiS time		Real space		Op. count		MUMPS time	
matrix	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}$	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}$	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}$	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}$
av41092	1.98	1.96	1.7E+07	1.6E+07	9.2E+09	9.1E+09	5.33	5.07
bbmat	1.81	1.78	3.9E+07	3.8E+07	2.9E+10	2.7E+10	13.76	13.01
circuit_4	0.35	0.36	5.0E+05	4.8E+05	1.3E+07	1.4E+07	0.28	0.27
cz10228	0.09	0.09	3.9E+05	3.6E+05	7.7E+06	7.0E+06	0.04	0.05
cz20468	0.19	0.19	7.7E+05	7.2E+05	1.5E+07	1.4E+07	0.09	0.10
cz40948	0.40	0.40	1.5E+06	1.4E+06	3.1E+07	2.8E+07	0.17	0.19
fd15	0.10	0.10	8.3E+05	8.3E+05	6.7E+07	6.9E+07	0.08	0.08
g7jac160	1.03	1.01	3.5E+07	3.5E+07	3.5E+10	3.7E+10	15.10	15.87
g7jac180	1.22	1.20	4.2E+07	4.1E+07	4.4E+10	4.5E+10	18.98	19.73
g7jac180sc	1.20	1.20	4.2E+07	3.8E+07	4.4E+10	3.9E+10	18.98	17.20
g7jac200	1.34	1.34	4.8E+07	4.7E+07	5.3E+10	5.3E+10	22.59	22.98
g7jac200sc	1.35	1.34	4.8E+07	4.2E+07	5.3E+10	4.5E+10	22.58	19.42
largebasis	8.33	8.13	3.4E+07	3.3E+07	1.3E+09	1.3E+09	2.14	2.41
lhr17c	0.08	0.08	6.1E+05	5.3E+05	5.8E+07	5.2E+07	0.05	0.06
lhr34c	0.18	0.19	1.2E+06	1.3E+06	1.1E+08	1.4E+08	0.11	0.15
lhr71c	0.18	0.18	1.2E+06	1.3E+06	1.1E+08	1.4E+08	0.12	0.14
mac_econ_fwd500	4.32	4.28	7.1E+07	6.7E+07	3.3E+10	3.0E+10	15.94	14.96
matrix_9	1.96	1.89	1.0E+08	1.1E+08	2.1E+11	2.2E+11	104.29	116.33
ohne2	8.22	5.62	2.4E+08	2.3E+08	3.1E+11	3.1E+11	148.09	146.63
onetone1	0.67	0.51	4.8E+06	4.3E+06	1.9E+09	1.7E+09	1.00	0.97
powersim	0.06	0.05	1.0E+05	9.7E+04	6.0E+05	5.0E+05	0.04	0.04
PR02R	7.72	4.78	1.8E+08	1.3E+08	1.9E+11	1.2E+11	91.23	55.06
pre2	10.46	10.14	1.2E+08	1.2E+08	2.1E+11	2.3E+11	104.94	120.16
sinc15	0.59	0.59	2.4E+07	2.1E+07	3.9E+10	3.5E+10	17.44	15.89
sinc18	1.02	0.97	4.8E+07	4.4E+07	1.0E+11	1.0E+11	47.67	52.49
std1_Jac2	0.40	0.34	6.0E+06	4.4E+06	3.4E+09	1.9E+09	1.64	0.93
std1_Jac3	0.43	0.34	6.2E+06	4.5E+06	3.6E+09	2.0E+09	1.85	1.02
twotone	1.27	1.20	1.2E+07	5.5E+06	1.1E+10	1.5E+09	5.44	1.17
viscoplastic2	0.88	0.81	6.5E+06	4.1E+06	1.5E+09	5.8E+08	0.98	0.59
Zd_Jac2	0.58	0.51	8.9E+06	7.9E+06	5.7E+09	4.2E+09	2.69	1.96
Zd_Jac3	0.62	0.51	9.3E+06	6.5E+06	5.7E+09	3.3E+09	2.71	1.55
Zd_Jac6	0.61	0.53	8.5E+06	7.2E+06	4.9E+09	3.6E+09	2.32	1.69
geomean	0.77	0.72	9.6E+06	8.5E+06	3.0E+09	2.5E+09	2.73	2.42

Table 11 – Measurements reported by MUMPS with $\mathbf{Q}_{\text{MC64}}$ and the proposed $\mathbf{Q}$. "Iter. ref." refers to the number of iterative refinement steps automatically performaned by MUMPS. "Off. diag. piv." and "Delayed piv." refer to the number of off-diagonal pivots and the delayed pivots during factorization. "$\frac{\text{real space}}{\text{est space}}$" refers to the ratio of the real space to the estimated space.

	Iter. ref.		Off. diag. piv.		Delayed piv.		$\frac{\text{real space}}{\text{est space}}$	
matrix	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}$	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}$	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}$	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}$
av41092	1	1	2782	5149	562	1366	1.00	1.00
bbmat	0	1	258	1	55	0	1.00	1.00
circuit_4	0	0	663	38	0	0	1.00	1.00
cz10228	0	0	0	15	0	0	1.00	1.00
cz20468	0	0	0	19	0	0	1.00	1.00
cz40948	0	0	0	28	0	0	1.00	1.00
fd15	0	0	74	64	208	24	1.00	1.01
g7jac160	0	1	252	1707	490	266	1.01	1.01
g7jac180	0	1	288	1850	466	247	1.01	1.01
g7jac180sc	0	1	407	3208	467	373	1.01	1.01
g7jac200	0	1	269	2143	582	302	1.01	1.02
g7jac200sc	0	1	398	3241	578	509	1.01	1.01
largebasis	0	0	0	1	0	0	1.00	1.00
lhr17c	0	2	14	520	17	48	1.02	1.06
lhr34c	0	1	887	950	272	158	1.00	1.00
lhr71c	1	1	889	841	274	133	1.00	1.00
mac_econ_fwd500	0	1	2198	6279	467	881	1.00	1.00
matrix_9	0	1	839	605	183	25	1.00	1.00
ohne2	0	1	48	0	0	0	1.00	1.00
onetone1	0	0	611	19	0	0	1.01	1.00
powersim	0	0	20	3	0	0	1.00	1.00
PR02R	1	2	4860	823	1434	33	1.01	1.00
pre2	0	0	3619	621	3589	545	1.00	1.00
sinc15	0	1	40	1127	34	48	1.00	1.00
sinc18	0	1	109	1298	20	94	1.00	1.00
std1_Jac2	0	1	2	294	0	17	1.00	1.00
std1_Jac3	0	2	0	118	0	5	1.00	1.00
twotone	0	0	1182	46	20	5	1.00	1.00
viscoplastic2	0	1	403	169	0	27	1.02	1.01
Zd_Jac2	0	2	2	305	0	13	1.00	1.00
Zd_Jac3	0	1	0	127	0	9	1.02	1.01
Zd_Jac6	0	1	0	77	0	0	1.02	1.00

[3] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Transactions on Mathematical Software*, 27(4):388–421, 2001.

[4] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[5] M. Arioli, J. W. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM Journal on Matrix Analysis and Applications*, 10(2):165–190, 1989.

[6] E. Boros, V. Gurvich, and I. Zverovich. Neighborhood hypergraphs of bipartite graphs. *Journal of Graph Theory*, 58(1):69–95, 2008.

[7] R. Burkard, M. Dell'Amico, and S. Martello. *Assignment Problems*. SIAM, Philadelphia, PA, USA, 2009.

[8] C. J. Colbourn and B. D. McKay. A correction to Colbourn's paper on the complexity of matrix symmetrizability. *Information Processing Letters*, 11:96–97, 1980.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 3rd edition, 2009.

[10] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.

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

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

[13] A. V. Goldberg and R. Kennedy. An efficient cost scaling algorithm for the assignment problem. *Mathematical Programming*, 71(2):153–177, 1995.

[14] HSL. HSL: A collection of Fortran codes for large-scale scientific computation. http://www.hsl.rl.ac.uk/, 2016.

[15] 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*. University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 55455, 1998.

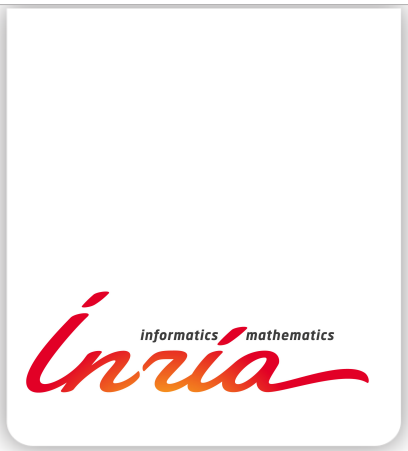
[16] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Dover, Mineola, New York (unabridged reprint of Combinatorial Optimization: Networks and Matroids, originally published by New York: Holt, Rinehart, and Wilson, c1976), 2001.

[17] X. S. Li and J. W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 1–17, Washington, DC, USA, 1998. IEEE Computer Society.

[18] M. Olschowka and A. Neumaier. A new pivoting strategy for Gaussian elimination. *Linear Algebra and Its Applications*, 240:131–151, 1996.

[19] F. Pellegrini. *SCOTCH 5.1 User's Guide*. Laboratoire Bordelais de Recherche en Informatique (LaBRI), 2008.

[20] B. Uçar. Heuristics for a matrix symmetrization problem. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Proceedings of Parallel Processing and Applied Mathematics (PPAM'07)*, volume 4967 of *Lecture Notes in Computer Science*, pages 718–727, 2008.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399