On matrix symmetrization and sparse direct solvers

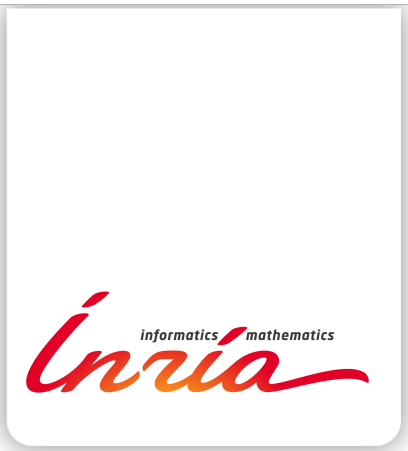
Raluca Portase, Bora Uçar

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 — 21 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 running 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. Duff and Koster [11, 12] offer efficient implementations of a set of exact algorithms for bipartite matching in the HSL [14] subroutine MC64 to permute matrices. In particular, one of the algorithms implements Olschowka and Neumaier's matching and obtains two diagonal matrices $\mathbf{D}_r$ and $\mathbf{D}_c$, and a permutation matrix $\mathbf{P}$ such that the permuted and scaled matrix $\mathbf{D}_r\mathbf{A}\mathbf{P}\mathbf{D}_c$ has all diagonal entries equal to one in absolute value, and all other entries less than or equal to one, again, 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 earlier work [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 for pattern symmetric matrices [3]. Therefore, column permutations increasing the symmetry should be useful for MUMPS and similar direct solvers as well.

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 precaution in MUMPS [3]). The heuristics for the second problem does 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 resulting matrix as candidates to be on the diagonal. The subset is chosen so that it would be helpful in numerical pivoting. By choosing all entries of $\mathbf{D}_r\mathbf{A}\mathbf{P}\mathbf{D}_c$ to be in that subset one recovers a pattern symmetrizing matching [20]; by choosing all entries of $\mathbf{D}_r\mathbf{A}\mathbf{P}\mathbf{D}_c$ 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 step for direct solvers for unsymmetric matrices computes an MC64 matching, and then orders the matrices for reducing the fill. 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 carrying out the factorization and solving the initial linear system. In particular, on a diverse set of matrices by adding only a little overhead in the standard preprocessing phase of direct solvers, 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, with respect to the

current state of the art.

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 used to achieve the two objectives in conjunction with 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 methods on MUMPS.

2 Background and notation

Given a square $(n \times n)$ matrix $\mathbf{A}$, we associate a bipartite graph $G_{\mathbf{A}} = (R \cup C, E)$ with it. Here, R and C are two disjoint vertex sets that correspond to the rows and the 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 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 simplify use $\text{mate}(v)$. Notice that 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 $\mathcal{M} \oplus \mathcal{C}$ do 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 $\textsc{SymScore}(\mathbf{A})$, 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 $\textsc{SymScore}(\mathbf{A})$. Observe that if $\mathbf{A}$ is symmetric, $\textsc{SymScore}(\mathbf{A})$ is equal to the number of nonzero entries of $\mathbf{A}$.

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

Among the algorithms implemented in MC64 [11, 12], two of them are of use in this work. 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 $\mathbf{Q}$ such that $\mathbf{AQ}$ 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 $\mathcal{O}(n\tau \log n)$, where n is the size of $\mathbf{A}$, and τ is the number of nonzeros in $\mathbf{A}$.

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 $\mathbf{Q}$ such that $\mathbf{AQ}$ 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 $\mathbf{D}_r = \mathrm{diag}(p_1, p_2, \ldots, p_n)$ and $\mathbf{D}_c = \mathrm{diag}(q_1, q_2, \ldots, q_n)$, where $p_i = \exp(u_i)$ and $q_j = \exp(v_j)/a_j$ such that the diagonal of the the permuted and scaled matrix $\mathbf{D}_r\mathbf{AQD}_c$ 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] in improving the pattern symmetry 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 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.

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

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) for a c ∈ adj(r_i).      */
```

diagonal of $\mathbf{AM}$, and two others are in the off-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 is tried to be improved by alternating the current matching with 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 this second alternative by incorporating methods to update gains of the alternating cycles (rather than recomputing the gains of cycles changing their gain value as done in the earlier work), and a few short-cuts to have improved running 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

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

Uçar's discusses that perfect matchings achieving the maximum weight according to the specified edge weights (2) help in obtaining good results in the pattern symmetry score.

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 is summarized in Algorithm 2. First, MC64 is applied to find a column permutation $\mathbf{Q}_{\text{MC64}}$ and the associated diagonal scaling matrices $\mathbf{D}_r$ and $\mathbf{D}_c$ such that $\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c$ has ones along the diagonal and all other entries no larger than one. Then, the matrix is ordered for reducing the potential fill. 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

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

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

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

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

The partial threshold pivoting scheme accepts a diagonal entry as pivot, if it is larger than a given threshold times the maximum entry (in absolute values) in the current column. We use MUMPS in describing Algorithm 2 to instantiate all components for clarity. MUMPS uses 0.01 as default value; the number typically is between 0.001 and 0.1 [3].

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 $\mathbf{A}\mathbf{Q}$ is more pattern symmetric than $\mathbf{A}\mathbf{Q}_{\text{MC64}}$. Additionally, $\mathbf{Q}$ should be numerically useful and help in avoiding pivoting for numerical issues.

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). 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, and 4 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`. At Line 3, we call ImproveSymmetry—described in the next subsection—to improve the pattern symmetry by finding the column permutation $\mathbf{Q}_{\text{Pat}}$ so that $\mathbf{Q}_{\text{MC64}}\mathbf{Q}_{\text{Pat}}$ 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 (permute and scaled) matrix $\mathbf{A}$. At Line 5, the direct solver MUMPS is called to factorize $\mathbf{A}''$ and solve the original linear system.

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 $|\mathbf{AQ}| + |\mathbf{AQ}|^T$ instead of $|\mathbf{AQ}_{\text{MC64}}| + |\mathbf{AQ}_{\text{MC64}}|^T$. With this, we expect improvements on all remaining steps of factoring $\mathbf{A}''$ and solving the initial linear system. First, since $\mathbf{AQ}$ is more symmetric in pattern than $\mathbf{AQ}_{\text{MC64}}$, the graph based ordering routines will have shorter running 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 increased iterative refinement [5] for better accuracy.

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 `/* qτ entries which are ≥ threshold are in A_f */`
3 $\mathbf{Q}_{Pat} \leftarrow$ ImproveSymmetry$(\mathbf{AQ}_{\text{MC64}}, \mathbf{A}_f)$
$\mathbf{R} \leftarrow$ fill-reducing-ordering$(\mathbf{AQ} + (\mathbf{AQ})^T)$ `/* Q as in (5) */`
$\mathbf{A}'' \leftarrow \mathbf{R}\mathbf{D}_r\mathbf{A}\mathbf{Q}\mathbf{D}_c'\mathbf{R}^T$ `/* D'_c as in (6) */`
4 $t \leftarrow \min(\text{diag}(|\mathbf{A}''|))$
5 call the solver on $\mathbf{A}''$ with the partial threshold pivoting $\frac{t}{100}$ `/* if t = 1 this becomes equivalent to the default choice for MUMPS. */`

3.1 An iterative-improvement based heuristic

In order to separate numerical concerns from the structural ones, the proposed approach constraints pattern symmetry improving matchings so as to include only large elements. Once the large elements are filtered into $\mathbf{A}_f$ at Line 2, we call the function ImproveSymmetry to improve the

symmetry of $\mathbf{AQ}_{\text{MC64}}$ by matchings using only entries 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). Since computing UB1 can be done using MC64 as a black-box, we proceed to explain the proposed iterative improvement method.

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 much improved running 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 list 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 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 of Line PASS, a priority queue is constructed on the set C_4 with the gain of alternating the cycle as the key value. For this, we compute the gain value 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 $\mathcal{C}$. 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 along $\mathcal{C}$, the gains of a set of cycles can change. We propose an efficient way to keep the gains up-to-date. 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 $c'_2 \in \operatorname{adj}_G(\{r_1, r_2\})$. Notice that 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 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 improvement that we get from r_1 is equal to the decrease that we get from r_2, and vice versa. 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 are 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 show the pattern of $\mathbf{A}(r'_2, [c_1, c_2])$, e.g., 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 Figure 2, in which we represent only the nonzero entries of the two cycles of interest and their interaction. Here, $\mathcal{C} = (r_1, c_1, r_2, c_2)$ and $\mathcal{C}' = (r'_1, c'_1, r'_2, c'_2)$ are two 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 (under the new matching). Now suppose that we alternated $\mathcal{C}$ and lost the symmetrical entry. 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

$\mathbf{A}(r'_1,[c_1,c_2])$ \ $\mathbf{A}(r'_2,[c_1,c_2])$	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) $\mathbf{A}(r_1,[c'_1,c'_2]) = \times0$ and b) $\mathbf{A}(r_1,[c'_1,c'_2]) = \times0$. 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 $\mathcal{C}$, $\text{gain}[\mathcal{C}'] = -1$. The pattern of $\mathbf{A}([r_1, r_2], [c'_1, c'_2])$ corresponds to the cell $(0\times, \times0)$ in Fig. 1a, and hence the gain of $\mathcal{C}'$ increases by 2 to be equal to 1, when $\mathcal{C}$ is alternated.

argument and once with r_2 at the same place. In the second call, Fig. 1a will be looked up for the case $\mathbf{A}(r_2,[c'_1,c'_2]) = \times0$, 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 running time of Algorithm 4. The first one is to short cut a pass (Line 5). We set a limit on the number of moves to be 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 pass which has been just finished 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%. These defaults values are justified empirically in Section 4.2.3.

3.3 Running time analysis

We investigate the running 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)

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)

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, 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 initialization steps take, in the worst case, $\mathcal{O}\left(\tau_f \log \tau_f + \sum_{u \in X}(|\operatorname{adj}_{G_f}(u)| - 1) \times (|\operatorname{adj}_G(u)| - 1)\right)$ time. There are at most $\mathcal{O}(n)$ extract operations from the heap and $\mathcal{O}(\tau_f)$ deletions in a pass from the heap, which costs a total of $\mathcal{O}((n + \tau_f) \log \tau_f)$. 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 running 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 algorithm's performance with respect to the earlier work. While doing so, we also explore the parameter space of the proposed algorithm. In the second set of experiments (Section 4.3), we evaluate the effects of the proposed algorithm in the context of the direct solver MUMPS.

4.1 Data set and environment

We created a data set as follows. From University of Florida Sparse Matrix Collection [10], we took all real, square, unsymmetric matrices with a numerical symmetry value less than 0.95, and with the following additional 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 15000000; (iii) have 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 score larger than 0.90τ for a matrix with τ nonzeros, as there is little potential improvement. This left us with 75 matrices at the end.

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 running time of the proposed 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), the pair of the ith row and the ith column of

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

Table 1 – Statistical indicators of the ratio of the pattern symmetry scores to the number of nonzeros 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 $\mathbf{A}_f$.

$\mathbf{A}$ is deemed 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 running time achieved by using the proposed gain-updates; and (iii) the impact of the other significant components of the proposed approach.

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 score. 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 cannot hope large improvements in the pattern symmetry.

We compare four thresholding schemes in Table 1 by giving the statistical indicators of the ratio $\textsc{SymScore}(\mathbf{AQ})/\tau$, where $\mathbf{Q}$ is as defined in (5). 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 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 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 to using $\mathbf{Q} = \mathbf{Q}_{\text{Pat}}$ (as all the effects of MC64 could be ignored). 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. Second, the larger the number of allowed entries, the higher the pattern symmetry score as expected. Allowing half of the entries improves the pattern symmetry score (with respect to using $\mathbf{Q}_{\text{MC64}}$ only) by 18%, on the average; allowing $1-e^{-1}$ improves the pattern symmetry score by 28%, and finally allowing all entries reaches on the average 45% improvements with respect to using $\mathbf{Q}_{\text{MC64}}$ only.

When compared to the running time of MC64, the geometric mean of the ratio of the running time of IterativeImprove to MC64 is 0.13, 0.42, 0.64 and 1.36 with the four thresholding scheme.

matrix	n	τ	Running time	
			no-gain-updates	with gain updates
ATandT/pre2	62962	5757273	6.38	1.62
Vavasis/av41092	41086	1170426	59.94	0.67
Schenk_ISEI/ohne2	181343	6858506	126.37	5.01
Fluorem/PR02R	160599	8176944	213.56	7.53
geomean (all 75 matrices)	34031	418154	0.26	0.06

Table 2 – Effect of the gain update (with respect to recomputing them) on the running time in seconds. In only these four matrices, IterativeImprove with the gain updates took more than one seconds. The last three matrices are the cases where not using the gain updates took the largest time.

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 score and the running time. Therefore, we keep the thresholding scheme $1 - e^{-1}$ as the default value (the remaining experiments use this scheme).

4.2.2 Effects of the gain updates

Using the proposed gain-updates subroutine improves the running time of earlier work [20]. As expected, the gain-update technique does not change the pattern symmetry score—ratio of the pattern symmetry scores 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 running 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 running time of IterativeImprove without the gain-updates was the largest (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 took larger than one second—we added `pre2`, which was the fourth one, to the table. As seen from these instances, IterativeImprove has sometimes large running times when UpdateGain is not used; with UpdateGain, the largest running time is 7.53 seconds. The arithmetic mean of the running time without and with gain-updates are, respectively, 8.91 and 0.36 seconds. The arithmetic and geometric mean of MC64's running time are 1.19 and 0.10 seconds, respectively. All these numbers confirm that the gain-updates has large impact in running time, and on average IterativeImprove is fast.

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

	$\mathcal{M}_0 = \mathcal{M}_{MC64}$	$\mathcal{M}_0 = \mathcal{M}_{UB1}$	
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

Table 3 – Statistical indicators of the running time of initializing ImproveSymmetry with an MC64 perfect matching (the column $\mathcal{M}_0 = \mathcal{M}_{MC64}$) and with a UB1 maximizing perfect matching; and the statistical indicators of the ratio of the pattern symmetry scores to the number of nonzeros.

as the objective on the bipartite graph of $\mathbf{A}_f$ using the edge weights (2). Any perfect matching could be used for this purpose. We compared using UB1 with that of using MC64 which is available (in other words, this 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.

As seen in Table 3, using UB1 as initial matching results in about 6% improvement with respect to not using UB1 (in this case IterativeImprove uses the matching of MC64 which is readily available). By looking at the geometric mean line, we 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 running time of MC64 to compute a maximum product perfect matching, to compute a maximum weighted perfect matching for UB1, and the running time of IterativeImprove on five matrices where the running time of MC64 was the largest while computing UB1. This table also includes two of the largest running times for MC64 while computing a maximum product perfect matching. As seen from this table, apart from `ohne2`, the running time of computing UB1 is in the same order of MC64, while being about three times slower in general (geometric mean of 0.37 versus 0.10 of MC64's). 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; the largest value of an edge could be n (but it is much smaller in our case, as we get rid of dense rows). With this observation, we have looked at algorithms whose running 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 running time complexity of $\mathcal{O}(\sqrt{n}\tau \log Wn)$, where W is the maximum edge weight. We have tested this algorithm and found it slower in general than MC64; the geometric mean of `csa_q`'s running time was 0.53. Furthermore, on an instance (the matrix `QLi/largebasis`), `csa_q` took more than an hour (MC64's running time was less than a second on this matrix). Both algorithms perform well on average, but the maximum was much worse with `csa_q`. That is why we keep MC64 as the default solver for UB1. Notice that if large running times are likely, one can skip computing UB1, and call IterativeImprove using the matching found by MC64 (at the first step of Algorithm 3).

We tested the effect of using the practical improvements discussed in Section 3.2. As the base case, we take the thresholding scheme $1 - e^{-1}$ and use UB1 for initial matching. We compare the pattern symmetry score and the running time of IterativeImprove in five iterations without any short cuts to that with the default values for the short cuts ($\min\{50, 0.005|C_4|\}$ for the window size of seeing an improvement, and 5% improvement between two passes). Without the short cuts, the geometric mean of the running time of IterativeImprove is 0.27 seconds; this is still fast but 2.75 times slower than MC64. The improvement in the final pattern symmetry score is only

matrix	MC64	UB1	IterativeImprove
Schenk_ISEI/ohne2	0.63	273.22	5.01
Muite/Chebyshev4	0.45	58.28	0.92
Fluorem/PR02R	51.96	52.02	7.53
Raju/laminar_duct3D	13.86	26.30	0.81
Norris/stomach	0.28	23.02	0.34

Table 4 – The running time of MC64 to compute a maximum product perfect matching, the time to compute a maximum weighted perfect matching for UB1 (using MC64), and the running time of IterativeImprove in seconds.

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 algorithm removed, optimized with respect to what is remaining, and put the removed rows/columns back into the matrix. There were 11 such matrices. The pattern symmetry score ratios agreed to the three significant digits; and the geometric mean of the ratio of the running times were 0.29 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 algorithm 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 before hand, 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 running time).

We compare the effects of the proposed $\mathbf{Q}$ (computed by the proposed ImproveSymmetry), with those of $\mathbf{Q}_{\text{MC64}}$ (computed by MC64) and with those of $\mathbf{Q}_{\text{Pat}}$ (computed by earlier work). We are going to look at four measurements: the running 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, factorization, and solution with iterative refinement when necessary). The running time depends on the compiler, the hardware, and the third party libraries (in particular BLAS for MUMPS), while the real-space and the operation count are absolute figures. Since this is so, we expect that the improvements in real-space and the operation count to be usually larger than the improvements in the running time.

As stated before, using $\mathbf{Q}_{\text{Pat}}$ ignores numerical issues. In our experiments with the 75 matrices, MUMPS returned with an error message for 29 matrices with the default settings. This shows that using $\mathbf{Q}_{\text{Pat}}$ is not a viable alternative for preprocessing unsymmetric matrices for direct methods. We now document the benefits of using ImproveSymmetry to compute $\mathbf{Q} = \mathbf{Q}_{\text{MC64}}\mathbf{Q}_{\text{Pat}}$ in MUMPS with respect to using $\mathbf{Q}_{\text{MC64}}$ only. By looking at all 75 matrices, we note 4% improvement in MeTiS's running time, 6% improvement in real space, 12% improvement in the operation count, and 4% improvement in the running time of MUMPS. However, we consider the improvements of

measurement	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}$
pattern symmetry score 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

Table 5 – Geometric mean of some measurements. Those of MC64 are given in absolute terms under the column $\mathbf{Q}_{\text{MC64}}$; those of ImproveSymmetry are given as the geometric mean of the ratios to MC64's results under the column $\mathbf{Q}$.

less than 10% in the pattern symmetry score to be insignificant to have impact on the direct solver, and 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 score by at least 10%. This set contained 32 matrices. We first note that using $\mathbf{Q}_{\text{MC64}}$ necessitated one step of iterative refinement in 3 matrices, whereas using $\mathbf{Q}$ required iterative refinement on 22 matrices (on 4 of them, two steps otherwise one step was seen). At the end, the backward error was always satisfactory.

The performance difference between using $\mathbf{Q}_{\text{MC64}}$ and $\mathbf{Q}$ in MUMPS is summarized in Table 5 and supplemented with the performance profiles given in Fig. 3. In a nutshell, the pattern symmetry score ratio of 0.25 is increased by 1.72 folds to be 0.43, on average, by ImproveSymmetry. This leads to 7% reduction in the running time of MeTiS (although MeTiS's running time is small, it is larger than that of MC64); 11% reduction in the actual real-space used by MUMPS (this is not symbolic, and computed after factorization); 18% reduction on the actual operation count in MUMPS (again computed after factorization); and finally 12% reduction in the total running of MUMPS (that is, the total of analysis, factorization, and the solution time using iterative refinement). The performance profile (Fig. 3) shows the percentage of test cases in which the running time of MUMPS with one permutation was no slower than its running 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 $\mathbf{Q}_{\text{MC64}}$ for all ρ. In this figure, we also see that $\mathbf{Q}$ is more robust than $\mathbf{Q}_{\text{MC64}}$ in the sense that the worst case of $\mathbf{Q}$ has a smaller ρ than that of $\mathbf{Q}_{\text{MC64}}$.

We give some detailed results with MUMPS on five matrices in Table 6 for displaying a complete picture. These five matrices are from the set where ImproveSymmetry resulted in at least 10% improvement in the pattern symmetry score, and showed diverse performance results. In the table, there is only one instance in which $\mathbf{Q}$ resulted in more fill-in than $\mathbf{Q}_{\text{MC64}}$. We looked closely to this, and saw that MUMPS reported similar numbers during the symbolic analysis phase; that is, the larger fill-in is not due to pivoting, but having a less effective ordering before the factorization. This was not a common case (the same outcome was observed only in five out of 32 instances). Otherwise, the gains in the real-space are usually in concordance with the operation count. This also reflects to the running time. In other cases (e.g., `sinc18`), the number of off-diagonal and delayed pivots are larger with $\mathbf{Q}$ which results in larger running time.

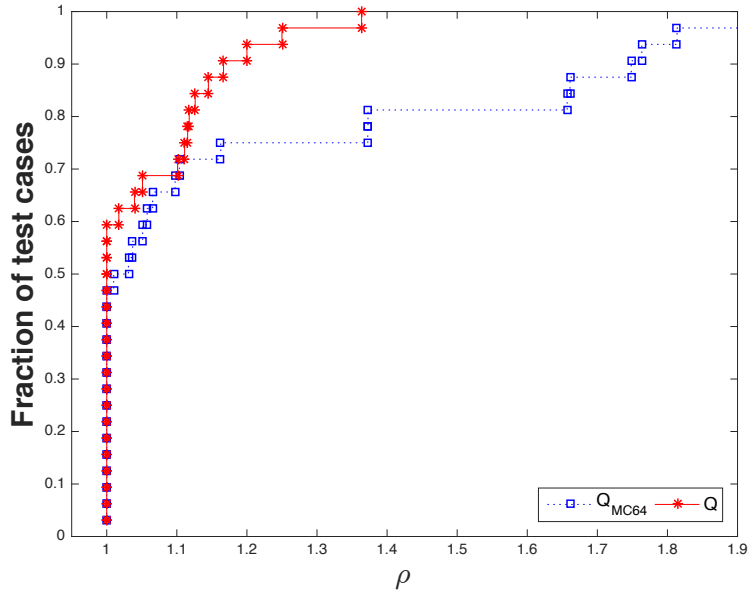


Figure 3 – Performance profiles of using $\mathbf{Q}_{\text{MC64}}$ and the proposed $\mathbf{Q}$ in the running time of MUMPS.

			Real-space		Operation count		MUMPS time	
matrix	n	τ	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}$	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}$	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}$
g7jac200sc	56474	706030	48190762	0.88	52815764548	0.85	22.58	0.86
pre2	629628	5757273	115199656	1.06	209672052762	1.12	104.94	1.15
sinc18	15650	913548	47910178	0.91	102643812100	1.01	47.67	1.10
twotone	105740	777549	12126432	0.45	11231143070	0.14	5.44	0.22
Zd_Jac3	11230	586278	9333136	0.70	5674622519	0.58	2.71	0.57

Table 6 – Detailed results on five matrices. Real-space and the operation count are reported by MUMPS after factorization (they are not symbolic results). In all metrics, those of $\mathbf{Q}_{\text{MC64}}$ are given in absolute terms, those of $\mathbf{Q}$ are given as ratios to that of $\mathbf{Q}_{\text{MC64}}$. The running time of $\mathbf{Q}_{\text{MC64}}$ is listed in seconds.

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 previously in an independent manner. 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, heuristics for the second one 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% improvements in the pattern symmetry are obtained, the memory and operation count requirement of MUMPS are improved by, respectively, 11% and 18%. This results in 12% improvement, on average, in the actual running time of MUMPS. The proposed algorithms are carefully implemented to be fast, and add only a little overhead to the standard preprocessing of sparse direct solvers for unsymmetric matrices.

Acknowledgement

We thank Jean-Yves L'Excellent and Guillaume Joslin for their help with the MUMPS experiments.

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.

[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. 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.

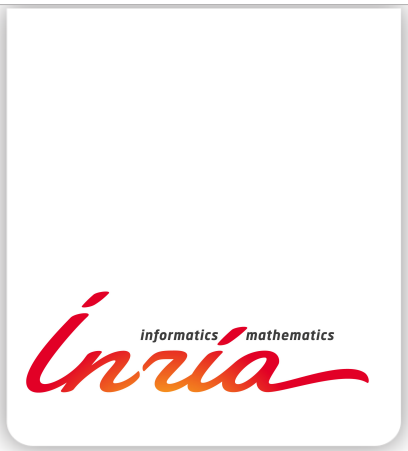
[16] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Dover, Mineola, New York (unabridged reprint of Combinatorial Optimization: Networks and Matroids, originally published by New York: Holt, Rinehart, and Wilson, c1976), 2001.

[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