Matrix symmetrization and sparse direct solvers

Raluca Portase, Bora Uçar

Matrix symmetrization and sparse direct solvers

Raluca Portase* Bora Uçar†

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. 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.

1 Introduction

We investigate bipartite matching algorithms for computing column permutations of a sparse matrix to achieve 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. Such permutations have use in combinatorial problems (in which the main concern is the pattern) and in sparse direct solvers for unsymmetric matrices that perform computations using the nonzero pattern of $|\mathbf{A}| + |\mathbf{A}|^T$ for a square matrix $\mathbf{A}$, and are exemplified by MUMPS [1, 3].

Olschowka and Neumaier [17] formulate the first objective as a maximum weighed bipartite matching problem, which is polynomial time solvable. Duff and Koster [10, 11] offer efficient implementations of a set of exact algorithms for the bipartite matching problem in the HSL [13] 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 [2, 11, 16, 17].

Uçar [19] investigates the second objective for $(0, 1)$-matrices. Referring to two earlier studies [5, 7], 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 [2]. 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. 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 by combining the algorithms from MC64 and earlier work [19]. 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 [19]; 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 the 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

*Technical University of Cluj-Napoca, Romania raluca_portase@yahoo.com

†LIP, UMR5668 (CNRS - ENS Lyon - UCBL - Université de Lyon - INRIA), Lyon, France, bora.ucar@ens-lyon.fr

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 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 [19], 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. We have engineered the main data structures and the algorithms of the earlier work [19] 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. An accompanying technical report [18] contains further experiments.

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 $(r_i, c_j) \in E$ if and only if $a_{ij} \neq 0$. When $\mathbf{A}$ is clear from the context, we use G for brevity. We refer to an edge as (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 [15]. 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 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 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. The diagonal entries contribute one, and a pair of symmetrically positioned off-diagonal entries contributes two to $\textsc{SymScore}(\mathbf{A})$; hence if $\mathbf{A}$ is symmetric, $\textsc{SymScore}(\mathbf{A})$ is equal to the number of nonzero entries of $\mathbf{A}$. In our combinatorial algorithms, we use the discrete quantity $\textsc{SymScore}(\mathbf{A})$ as the optimization metric. Since, the ratio $\textsc{SymScore}(\mathbf{A})/\tau$ measures how symmetric the matrix $\mathbf{A}$ with τ nonzeros is, we use this ratio 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 [10, 11]. 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. There are a number of polynomial time algorithms for this purpose [6, Ch. 4]; the one that is implemented in MC64 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 a minimum weight perfect matching problem. The matching algorithms used in MC64 also construct two diagonal matrices $\mathbf{D}_r$ and $\mathbf{D}_c$ such that the diagonal of the permuted and scaled matrix $\mathbf{D}_r\mathbf{APD}_c$ 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 [19] for

improving the pattern symmetry score of matrices. This heuristic works on the bipartite graph associated with $\mathbf{A}$. 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.

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

$$\textsc{SymScore}(\mathbf{AM}) = n + 2 \times |C_4| \ , \tag{2.1}$$

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

By observing the role of the alternating cycles of length four in (2.1), Uçar [19] 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 [19] 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, 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

$$\min\{|\operatorname{adj}(r_i)|, |\operatorname{adj}(c_j)|\} \ . \tag{2.2}$$

To see why UB1 is an upper bound, let us count the potentially symmetric entries from the rows, one row at a time. When the row r_i and the column c_j are put in the corresponding positions, the maximum number of nonzeros in 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 (2.2) 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 1. 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, 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. After the preprocessing, the direct solver effectively factorizes

$$\mathbf{A}' = \mathbf{P}\mathbf{D}_r\mathbf{A}\mathbf{Q}_{\text{MC64}}\mathbf{D}_c\mathbf{P}^T \ . \tag{2.3}$$

Algorithm 1: 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

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 [2].

3 Matrix symmetrization for direct solvers

Our aim is to find another column permutation $\mathbf{Q}$ instead of $\mathbf{Q}_{\text{MC64}}$ in (2.3) so that $\mathbf{AQ}$ is more pattern symmetric than $\mathbf{AQ}_{\text{MC64}}$. 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, 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

$$\mathbf{A}'' = \mathbf{R}\mathbf{D}_r\mathbf{A}\mathbf{Q}\mathbf{D}_c'\mathbf{R}^T\,, \tag{3.4}$$

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

$$\mathbf{Q} = \mathbf{Q}_{\text{MC64}}\mathbf{Q}_{\text{Pat}} \tag{3.5}$$

is a permutation matrix, and

$$\mathbf{D}_c' = \mathbf{Q}_{\text{Pat}}{}'\mathbf{D}_c\mathbf{Q}_{\text{Pat}} \tag{3.6}$$

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 2, where the numbered lines contain the differences with respect to the existing framework shown in Algorithm 1. 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 [8, 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, and in the second substep we iteratively improve this initial permutation for better pattern symmetry score, obtaining $\mathbf{Q}_{\text{Pat}}$. 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. The value $t/100$ is used in order to recover the behavior of Algorithm 1, if $\mathbf{Q}_{\text{Pat}}$ is identity, or if $\mathbf{Q}$ 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.

Algorithm 2: 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
$\mathbf{Q}_{Pat} \leftarrow$ IterativeImprove($\mathbf{AQ}_{\text{MC64}}, \mathbf{A}_f, \mathbf{M}_0$)

$\mathbf{R} \leftarrow$ fill-red.-ordering($|\mathbf{AQ}| + |\mathbf{AQ}|^T$)`/* see (3.5) */`

$\mathbf{A}'' \leftarrow \mathbf{R}\mathbf{D}_r\mathbf{A}\mathbf{Q}\mathbf{D}_c'\mathbf{R}^T$ `/* $\mathbf{D}_c'$ as in (3.6) */`
4 $t \leftarrow \min(\text{diag}(|\mathbf{A}''|))$
5 call MUMPS on $\mathbf{A}''$ with 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, the graph based ordering routines will have shorter run time and will be more effective, as $|\mathbf{AQ}|+|\mathbf{AQ}|^T$ is more 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 [4] taken to recover the accuracy.

3.1 An iterative-improvement based heuristic.

In order to separate the 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 2, 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 3. 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 (2.2). 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 in Section 4.3). 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 3 are standard. The significant differences with respect to the earlier work [19] which enable the incorporation of the numerical concerns and improved run time are shown at the numbered lines.

The Lines 1 and 2 of Algorithm 3 are related. The first line builds a bipartite graph from the entries allowed to be in the diagonal. 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 (labeled Pass in Algorithm 3), 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

Algorithm 3: IterativeImprove($\mathbf{A}, \mathbf{A}_f, \mathbf{M}_0$)

Input : $\mathbf{A}$, a matrix.
$\mathbf{A}_f$, entries allowed to be in the matching.
$\mathbf{M}_0$, a permutation matrix corresponding to a perfect matching $\mathcal{M}_0$ on $\mathbf{A}$

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 $\mathbf{A}_f$
$\mathcal{M}_1 \leftarrow \mathcal{M}_0$
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 $>$ bestSymm **then**
      update bestSymm
    $\mathcal{M}_1 \leftarrow \mathcal{M}_1 \oplus \mathcal{C}$ `/* match` (r_1, c_2) `and` (r_2, c_1) `*/`
4   HeapDelete($\mathcal{C}'$) for $\mathcal{C}' \in \text{adj}_{G_o}(\{r_1, c_1, r_2, c_2\})$
5   **if** *no gain since a long time* **then** break
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
  initSymmScore $\leftarrow$ bestSymm

$\mathcal{C}$ can be computed as the difference

$$\text{gain}[\mathcal{C}] = s_1 - s_0 \ , \tag{3.7}$$

where

$$s_0 = 2(|\,\text{mate}(\text{adj}_G(c_1)) \cap \text{adj}_G(r_1)| + |\,\text{mate}(\text{adj}_G(c_2)) \cap \text{adj}_G(r_2)|) \tag{3.8}$$

$$s_1 = 2(|\,\text{mate}(\text{adj}_G(c_2)) \cap \text{adj}_G(r_1)| + |\,\text{mate}(\text{adj}_G(c_1)) \cap \text{adj}_G(r_2)|) \ . \tag{3.9}$$

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 original matrix $\mathbf{A}$. 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 [19]. Since all gains are even (symmetric pairs add two to the pattern symmetry score), we simplify the factor two from (3.8)–(3.9) 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 $c'_1 \in \text{adj}_G(\{r_1, r_2\})$ or $c'_2 \in \text{adj}_G(\{r_1, r_2\})$. 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) $\in \text{adj}_G(r_1)$ and (c'_1 or c'_2) $\in \text{adj}_G(r_2)$. 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 suffer 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 $c'_2 \in \text{adj}_G(r_1)$ 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 $c_1 \notin (\text{adj}_G(r'_2))$ and $c_2 \in (\text{adj}_G(r'_2))$. Similarly, the header of the rows shows the pattern of $\mathbf{A}(r'_2, [c_1, c_2])$ and $\mathbf{A}(r'_1, [c_1, c_2])$. The observations from above are translated into the pseudocode shown in Algorithm 4, where Algorithm 3 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 3. The first one is to short cut a pass (Line 5). We set a limit on the number of moves (tentatively realized) between the

$\mathbf{A}(r'_1, [c_1, c_2])$	$\mathbf{A}(r'_2, [c_1, c_2])$ 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.

Algorithm 4: UPDATEGAINS($G, \mathcal{C}, r$, cycleHeap)

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

$mark(c') \leftarrow r$ for all $c' \in \text{adj}_G(r)$
$mark(r') \leftarrow c_1$ for all $r' \in \text{adj}_G(c_1)$
$mark(r') \leftarrow c_2$ for all $r' \in \text{adj}_G(c_2)$
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 = r1, 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=r1, 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)

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 3 returns. Our default setting starts another pass, if the finishing pass has improved the pattern symmetry score by at least 5%. As expected, these shortcuts were always very helpful in reducing the run time, without tangible difference in the obtained symmetry score (see the accompanying report [18, Sec. 4.2.3]).

3.3 Run time analysis. We investigate the run time of Algorithm 3. The initialization steps up to Line 3 take time in $\mathcal{O}(n+\tau)$. Computing the gain of a cycle (r_1, c_1, r_2, c_2) using Eqs. (3.7), (3.8), and (3.9) 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 $\mathcal{O}\left(\sum_{u\in X}(|\operatorname{adj}_{G_f}(u)|-1) \times (\operatorname{adj}_G(u)-1)\right)$, where $X = R \cup C$. 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 $\mathcal{O}((n+\tau_f)\log \tau_f)$. 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 comparison, the gain computations at each pass in the earlier work [19] 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 Experimental result

We present a set of results to highlight the improvements with respect to earlier work [19] and to show the effects within a direct solver. The whole set of experiments can be found in the accompanying report [18]. The previous work [19] also contains some experiments regarding the effect of the number of iterations.

4.1 Comparison with the earlier work. Using the proposed gain-updates subroutine improves the run time of earlier work [19]. In order to document this, we use a set of 75 square matrices from the SuiteSparse Matrix Collection (formerly UFL) [9] 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. On these matrices, we took the largest irreducible block and worked on these blocks. 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 1 (in only the last three matrices the gain updates took longer than one second). 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 (geometric means are 0.06 vs 0.10 [18, Table 9]).

Table 1: Effect of the gain update (with respect to re-computing them) on the run time of IterativeImprove (Algorithm 3) 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

4.2 Investigations with a direct method. The thresholding scheme at Line 1 of Algorithm 2 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 (3.4). 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 2) 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. 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. By conducting a large set of experiments (see the report [18, Sec. 4.2.1]), we concluded that allowing around 0.63τ nonzeros in $\mathbf{A}_f$ strikes a good balance for symmetry and run time. We use this parameter in the experiments of this subsection.

We present results with MUMPS 5.0.1 [1, 3]. For the fill-reducing ordering step, we used MeTiS [14] 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 2 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 first tested using the permutations computed in the earlier study [19], and observed that it is not a viable alternative: MUMPS returned with an error message (related to memory allocation) for 29 out of 75 matrices of the previous subsection with the default parameters. We then compared the effects of $\mathbf{Q}$ (computed by ImproveSymmetry), with those of $\mathbf{Q}_{\text{MC64}}$ (computed by MC64). In both 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. We present results with 32 matrices on which ImproveSymmetry delivered more than 10% improvement with respect to $\mathbf{Q}_{\text{MC64}}$. Table 2 contains the results for these matrices.

From Table 2, we observe that the use of $\mathbf{Q}$ brings the following improvements in comparison to the use of $\mathbf{Q}_{\text{MC64}}$: (i) the run time of MeTiS is reduced by 7%; (ii) the real space used by MUMPS (reported after the factorization) is reduced by 11%; (iii) the operation count in MUMPS (after the factorization) is reduced by 18%; (iv) the run time of MUMPS (analysis, factorization, and the solution time with potentially the iterative refinement) is reduced by 12%. The results show that considerable reductions in the real space and operation count is possible by increasing the symmetry, on top of potential gains inside MUMPS due to increased symmetry. This entails reductions in the run time of the remaining steps in solving a linear system (see a remarkable result for PR02R, whose symmetry ratio is increased to 0.94, which was 0.72 for $\mathbf{Q}_{\text{MC64}}$, as seen in Table 8 of the technical report [18]).

4.3 Run time. We now investigate the run time briefly (the technical report [18] contains detailed results). As discussed earlier, 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 $\mathbf{A}_f$). 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 $\mathbf{A}\mathbf{Q}_{\text{MC64}}$ 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, 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 for computing UB1. In Table 4, we give the run time of MC64 to compute a maximum product perfect matching MC64_π, 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. 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 algorithm of time complexity $\mathcal{O}(n\tau \log n)$. We have integer edge weights coming from a small set for computing UB1. With this observation,

Table 2: Results on 32 matrices for the four metrics (MeTiS time, real space, operation count, and MUMPS time)—those with $\mathbf{Q}_{\text{MC64}}$ are given in absolute terms, and those with $\mathbf{Q}$ are given as ratios to that with $\mathbf{Q}_{\text{MC64}}$.

	MeTiS time		Real space		Op. count		MUMPS time	
matrix	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}/\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}/\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}/\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}_{\text{MC64}}$	$\mathbf{Q}/\mathbf{Q}_{\text{MC64}}$
av41092	1.98	0.99	1.7E+07	0.94	9.2E+09	0.99	5.33	0.95
bbmat	1.81	0.99	3.9E+07	0.97	2.9E+10	0.94	13.76	0.95
circuit_4	0.35	1.03	5.0E+05	0.97	1.3E+07	1.04	0.28	0.94
cz10228	0.09	0.99	3.9E+05	0.93	7.7E+06	0.92	0.04	1.15
cz20468	0.19	0.97	7.7E+05	0.93	1.5E+07	0.92	0.09	1.13
cz40948	0.40	1.00	1.5E+06	0.93	3.1E+07	0.93	0.17	1.11
fd15	0.10	0.99	8.3E+05	1.00	6.7E+07	1.03	0.08	1.04
g7jac160	1.03	0.98	3.5E+07	1.00	3.5E+10	1.04	15.10	1.05
g7jac180	1.22	0.98	4.2E+07	0.97	4.4E+10	1.03	18.98	1.04
g7jac180sc	1.20	1.00	4.2E+07	0.89	4.4E+10	0.88	18.98	0.91
g7jac200	1.34	1.00	4.8E+07	0.98	5.3E+10	1.00	22.59	1.02
g7jac200sc	1.35	1.00	4.8E+07	0.88	5.3E+10	0.85	22.58	0.86
largebasis	8.33	0.98	3.4E+07	0.98	1.3E+09	0.95	2.14	1.12
lhr17c	0.08	1.00	6.1E+05	0.86	5.8E+07	0.90	0.05	1.13
lhr34c	0.18	1.03	1.2E+06	1.12	1.1E+08	1.28	0.11	1.33
lhr71c	0.18	1.01	1.2E+06	1.12	1.1E+08	1.29	0.12	1.18
mac_econ_fwd500	4.32	0.99	7.1E+07	0.94	3.3E+10	0.91	15.94	0.94
matrix_9	1.96	0.96	1.0E+08	1.04	2.1E+11	1.08	104.29	1.12
ohne2	8.22	0.68	2.4E+08	0.97	3.1E+11	0.99	148.09	0.99
onetone1	0.67	0.77	4.8E+06	0.90	1.9E+09	0.90	1.00	0.97
powersim	0.06	0.97	1.0E+05	0.93	6.0E+05	0.83	0.04	1.03
PR02R	7.72	0.62	1.8E+08	0.72	1.9E+11	0.60	91.23	0.60
pre2	10.46	0.97	1.2E+08	1.06	2.1E+11	1.12	104.94	1.15
sinc15	0.59	1.01	2.4E+07	0.87	3.9E+10	0.89	17.44	0.91
sinc18	1.02	0.95	4.8E+07	0.91	1.0E+11	1.01	47.67	1.10
std1_Jac2	0.40	0.85	6.0E+06	0.73	3.4E+09	0.56	1.64	0.57
std1_Jac3	0.43	0.80	6.2E+06	0.72	3.6E+09	0.56	1.85	0.55
twotone	1.27	0.94	1.2E+07	0.45	1.1E+10	0.14	5.44	0.22
viscoplastic2	0.88	0.92	6.5E+06	0.64	1.5E+09	0.40	0.98	0.60
Zd_Jac2	0.58	0.88	8.9E+06	0.89	5.7E+09	0.73	2.69	0.73
Zd_Jac3	0.62	0.81	9.3E+06	0.70	5.7E+09	0.58	2.71	0.57
Zd_Jac6	0.61	0.88	8.5E+06	0.85	4.9E+09	0.74	2.32	0.73
geomean	0.77	0.93	9.6E+06	0.89	3.0E+09	0.82	2.73	0.88

Table 3: Statistical indicators of the pattern symmetry ratios obtained by IterativeImprove initialized with $\mathcal{M}_0 = \mathcal{M}_{MC64}$; the run time of computing a perfect matching attaining UB1, and the pattern symmetry ratio obtained by IterativeImprove when initialized with $\mathcal{M}_0 = \mathcal{M}_{UB1}$.

	$\mathcal{M}_0 = \mathcal{M}_{MC64}$	$\mathcal{M}_0 = \mathcal{M}_{UB1}$	
statistics	sym. ratio	UB1 time (s)	sym. ratio
min	0.10	0.01	0.12
max	0.94	273.22	0.94
geomean	0.48	0.37	0.51

we have looked at one of the fastest algorithms for this case. The algorithm, called `csa_q` [12], 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 of the report [18] 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 2). With all these, we note that the total time for solving linear system might increase, if we include the run time of all the preprocessing steps. In particular, when we add all times (MeTiS + UB1 + ItImp + MUMPS) with $\mathbf{Q}$ and all times (MeTiS + MUMPS) with $\mathbf{Q}_{\text{MC64}}$, we have a total time (on average) of 3.56 in the first case and 3.50 in the second case. Fortunately, all steps (ordering, factorization, and solve time) involved in solving the linear system after the application of IterativeImprove see reduced run time, on top of the improved memory requirement and operation count.

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. Duff and Koster address the first objective within

Table 4: The run time of MC64 to compute a maximum product perfect matching ($MC64_\pi$), 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.

		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

MC64 [11] with the maximum product perfect matching. Uçar [19] addresses the second objective and highlights that the problem is NP-complete. Therefore, 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 this paper we focused on the use of the symmetry increasing heuristics for the direct methods. Previous work [19] and the accompanying technical report [18, Sec. 4.4] contain results on the combinatorial problems of, respectively, graph partitioning by vertex separators and profile reduction. In short, 43% smaller separators and 8% smaller profiles are obtained on a set of matrices thanks to the symmetrization heuristics.

The proposed method has two components: initialization and iterative improvement. The first component uses a maximum weighted perfect matching (UB1) to initialize for which we used MC64 as a black-box. 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. We tried one promising algorithm but could not get a consistent improvement. A mechanism to make an automatic choice between these two matching codes will be very useful. To speed up the first component, one can use fast heuristics to approximate UB1, or can come up with initialization methods that are not related to UB1.

References

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

[2] 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 T. Math. Software*, 27(4):388–421, 2001.

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

[4] M. Arioli, J. W. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Math. Anal.*, 10(2):165–190, 1989.

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

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

[7] C. J. Colbourn and B. D. McKay. A correction to Colbourn's paper on the complexity of matrix symmetrizability. *Inform. Process. Lett.*, 11:96–97, 1980.

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

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

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

[11] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Math. Anal.*, 22:973–996, 2001.

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

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

[14] 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.

[15] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Dover, Mineola, New York, 2001.

[16] 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.

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

[18] R. Portase and B. Uçar. On matrix symmetrization and sparse direct solvers. Research Report RR-8977, Inria Grenoble – Rhône-Alpes, November 2016.

[19] B. Uçar. Heuristics for a matrix symmetrization problem. In *Proceedings of Parallel Processing and Applied Mathematics (PPAM'07)*, pages 718–727, 2008.