Science & Technology Facilities Council

Preprint

RAL-P-2014-002

On the efficient scaling of sparse symmetric matrices using an auction algorithm

JD Hogg, JA Scott

February 2014

Submitted for publication in Numerical Linear Algebra with Applications

RAL Library
STFC Rutherford Appleton Laboratory
R61
Harwell Oxford
Didcot
OX11 0QX

Tel: +44(0)1235 445384
Fax: +44(0)1235 446403
email: libraryral@stfc.ac.uk

Science and Technology Facilities Council preprints are available online at: http://epubs.stfc.ac.uk

ISSN 1361- 4762

On the efficient scaling of sparse symmetric matrices using an auction algorithm

Jonathan Hogg and Jennifer Scott

January 28, 2014

Abstract

The well-known HSL software package `MC64` is a powerful tool for scaling sparse matrices prior to the application of direct and iterative methods to solve linear systems $Ax = b$. It computes a scaling by using the Hungarian algorithm to solve the maximum weight maximum cardinality matching problem. However, with the parallelization of the factorization and solve phases of direct solvers, the serial Hungarian algorithm can represent an unacceptably large proportion of the total solution time for such solvers.

Recently, auction algorithms and approximation algorithms have been suggested as alternatives for achieving near-optimal solutions for the maximum weight maximum cardinality matching problem. In this paper, the efficacy of auction and approximation algorithms as replacements for the Hungarian algorithm is assessed in the context of sparse symmetric direct solvers when used on problems arising from a range of practical applications. High cardinality sub-optimal matchings are shown to be as effective as optimal matchings for the purposes of scaling. However, a higher degree of optimality is required to effectively use matching-based ordering techniques. The auction algorithm is demonstrated to be capable of finding such matchings significantly faster than the Hungarian algorithm, but the 1/2-approximation matching fails to consistently achieve a sufficient cardinality.

1 Introduction

Our aim is to efficiently solve the large sparse linear system

$$Ax = b.$$

Our main interest is the use of direct solvers when A is symmetric indefinite. In this case, it is necessary to incorporate numerical pivoting to maintain stability. This can mean that the pivot sequence chosen during the analyse phase on the basis of sparsity has to be modified as the numerical factorization proceeds. In particular, some pivots have to be delayed until they satisfy the stability criteria. Our recent studies [13, 17] demonstrate that the number of delayed pivots provides a good predictor of the effectiveness of a scaling at reducing the wall clock time for the factorization. This is because the number of delayed pivots corresponds to the amount of additional work performed due to the requirements for numerical pivoting. Our work also demonstrates that, for tough indefinite problems, the widely-used `MC64` scaling algorithm [8] is particularly effective compared to other scalings and techniques tested.

`MC64` seeks to find an ordering such that the product of the entries on the diagonal of the reordered matrix is maximized. The problem is only well defined if a permutation exists such that the diagonal is zero-free, however the degenerate case can also be treated, see for example our recent work [16]. Stated mathematically, given a sparse matrix $A = \{a_{ij}\}$, we associate a bipartite graph having vertex sets V_r, V_c corresponding to the rows and columns, and an edge $(i, j) \in E$ joining row i to column j if a_{ij} is nonzero. An edge subset $\mathcal{M} \subseteq E$ is called a *matching* if no two edges in $\mathcal{M}$ are incident to the same vertex. We seek a matching $\mathcal{M}$ of the row vertices to the column vertices such that the cardinality $|\mathcal{M}|$ is maximized and the product of the entries $|a_{ij}|$ for each edge in the matching is maximized. If σ is an indicator function such that $\sigma_{ij} = 1$ if edge $(i,j) \in \mathcal{M}$ and is zero otherwise, then we aim to solve the following problem:

$$\begin{aligned} \max \quad & \prod_{(i,j)\in E} |a_{ij}|\sigma_{ij} & (1.1) \\ \text{s.t.} \quad & \sum_{j\in V_c} \sigma_{ij} = 1, \qquad \forall i \in V_r & (1.2) \\ & \sum_{i\in V_r} \sigma_{ij} = 1, \qquad \forall j \in V_c & (1.3) \\ & \sigma_{ij} \in \{0,1\}. & (1.4) \end{aligned}$$

The MC64 algorithm applies a transformation to a_{ij} to give an associated (positive) edge weight

$$w_{ij} = \log c_j - \log |a_{ij}|, \tag{1.5}$$

where $c_j = \max_i |a_{ij}|$. The maximization in (1.1) is then equivalent to

$$\min \sum_{(i,j)\in E} w_{ij}\sigma_{ij} \tag{1.6}$$

By way of a sign change, this is classified as a maximum weight maximum cardinality matching problem (also known as an assignment problem). In MC64, this is solved using the Hungarian algorithm [18]. From standard theory, at optimality the following conditions are satisfied for some row and column dual variables u and v:

$$\begin{aligned} w_{ij} - u_i - v_j = 0, \qquad & \forall (i,j) \in \mathcal{M}, & (1.7) \\ w_{ij} - u_i - v_j \geq 0, \qquad & \text{otherwise}. & (1.8) \end{aligned}$$

The matching $\mathcal{M}$ provides a permutation that, in the unsymmetric case, can be used to achieve a zero-free diagonal with large entries. In the symmetric case it can be used to permute large entries on to the subdiagonal [9, 11, 24, 25]. The dual variables u and v can be used to calculate a scaling as follows. Define the diagonal scaling matrices D_r, D_c and S with diagonal entries

$$\begin{aligned} d_i^r &= \exp(u_i), \\ d_j^c &= \exp(v_j - c_j), \\ s_i &= \sqrt{d_i^r d_i^c}. \end{aligned}$$

Then $D_1 A D_2$ is such that the largest entry in each row and column is exactly one and all other entries are less than or equal to one. If A is symmetric, SAS has the same property. The permutation and scaling can be used independently if required, and it is especially common in the symmetric case to use only the scaling.

There are two potential problems with MC64: (i) the run time is hard to predict and can vary significantly when the data is permuted; and (ii) an application of MC64 can represent a significant fraction of the total factorization time when using a direct solver, particularly when the solver is run in parallel (see Table 4.5 in Section 4). The latter point is compounded by Amdahl's Law, as MC64 is a serial code whilst the factorization obtains good parallel speedups on a modest number of cores. The main issue lies with the Hungarian algorithm that MC64 uses to solve the assignment problem. This seeks optimal augmenting paths through the matrix from an unmatched row to an unmatched column. In those cases where performance is poor it is because of the need to scan a significant portion of the entire matrix while proving optimality for each augmenting path.

We note that it may be possible to parallelize the Hungarian algorithm using similar techniques to those used for the unweighted case [2, 7]. However, we expect the speedups to be significantly more limited because at each stage optimal independent augmenting paths must be found, whereas in the unweighted case any augmenting path will do.

In this paper, we relax the requirement for a maximum cardinality matching to allow us to use algorithms that deliver near-optimal results in weight and cardinality. The solution hence does not provide the zero-free diagonal often desired by unsymmetric solvers, but does allow the majority of large entries to be permuted to the subdiagonal for symmetric matrices and, as we shall demonstrate, still provides a high quality scaling for most of our test matrices which are taken from practical applications.

The main contribution of this paper is a comparison of the performance and effectiveness of two alternative algorithms for the relaxed maximum weight maximum cardinality matching problem with that of the `MC64` implementation of the Hungarian algorithm when the resulting scaling is used prior to the factorization of sparse symmetric matrices. These alternatives are an auction algorithm and a $\frac{1}{2}$-approximation algorithm. Both solve the problem approximately whilst claiming to offer significantly better parallel speedups than the Hungarian algorithm for large problems [12, 23]. We assess performance in terms of time to find the matching and its effectiveness when used as a scaling and/or ordering heuristic for a sparse direct symmetric linear solver.

The remainder of this paper is laid out as follows. In Section 2, we describe the auction algorithm and associated work; both serial and parallel versions are discussed. Then, in Section 3, we describe the approximation algorithm. Section 4 provides a comparison of the effectiveness of these algorithms, both in terms of performance and numerical improvement to the scaled matrix; comparisons are made with the Hungarian algorithm. Finally, in Section 5, our conclusions are presented.

2 The Auction Algorithm

The auction algorithm for the maximum weight matching problem was first proposed by Bertsekas [3] and since then has been studied in a number of papers, including [4, 5, 22]. Most recently, Sathe et al. [23] showed that the algorithm can quickly find high quality matchings and is readily parallelizable.

The auction algorithm solves the following maximum weight matching problem

$$\begin{aligned} \max \quad & \sum_{(i,j)\in E} w_{ij}\sigma_{ij} && (2.1)\\ \text{s.t.} \quad & \sum_{j\in V_c} \sigma_{ij} \le 1, \qquad \forall i \in V_r && (2.2)\\ & \sum_{i\in V_r} \sigma_{ij} \le 1, \qquad \forall j \in V_c && (2.3)\\ & \sigma_{ij} \in \{0,1\}. && (2.4) \end{aligned}$$

To transform (1.1) to (2.1), in place of (1.5), we use the related transformation

$$w_{ij} = \alpha + \log|a_{ij}| + (\alpha - c_j), \qquad (2.5)$$

where $c_j = \max_i \log|a_{ij}|$ and $\alpha = \max_{i,j}\{c_j - \log|a_{ij}|\}$. The transformation $\log|a_{ij}| + (\alpha - c_j)$ is sufficient to transform the maximum product into a scaled maximum sum over positive weights. The extra α term transforms the objective from a maximum weight to maximum weight maximum cardinality because α is greater than each individual $\log|a_{ij}|+(\alpha-c_j)$ term, only a maximum cardinality solution can be optimal (this trick has been used by other authors previously, for example in LEDA [20]). A corresponding unsymmetric scaling of A is then given by

$$\begin{aligned} d_i^r &= \exp(\alpha - u_i),\\ d_j^c &= \exp(\alpha - v_j - c_j), \end{aligned}$$

which can again be symmetrized using

$$s_i = \sqrt{d_i^r d_i^c}$$

.

For each nonzero entry a_{ij} of A, $w_{ij} - u_i$ is the increase in the objective obtained by augmenting $\mathcal{M}$ with (i, j), displacing any edge currently in $\mathcal{M}$ that contains row i or column j. The auction algorithm starts with the row dual variables, $u = \{u_i\}$, initialised to zero and proceeds by scanning each unmatched column j to find the row index i such that

$$i = \arg\max_k \{w_{kj} - u_k\}.$$

If $w_{ij} - u_i > 0$, column j "bids" for row i. The highest bid for row i wins, say in column j_1, the matching $\mathcal{M}$ is augmented by adding the edge (i, j_1) and any column previously matched with row i is returned to the pool of unmatched columns. The dual variable u_i is updated to be the cost of using the second best row: that is, u_i is the (first order) reduction in the objective if j_1 was not matched to i. For this reason, the dual vector u is also referred to in some contexts as the vector of "reduced costs". By adding $\epsilon > 0$ to u_i, a minimum increase in the objective can optionally be required. This accelerates convergence of the algorithm by ignoring opportunities for trivial improvement. ϵ is chosen to be small but much larger than machine precision and it is increased as the algorithm proceeds.

We note that for the serial algorithm, the (average) number of iterations and cost per iteration can be reduced by treating every bid as immediately winning. This reduces the cost per iteration, as there is no longer the need to determine the highest bid (each bid wins) and the data needed for the resulting update are already in cache. Further, if a bid by column j for row i wins immediately, any existing k such that $(i, k) \in \mathcal{M}$ becomes available for rematching in the current iteration.

We have implemented both the serial and OpenMP versions of the auction algorithm, which we outline as Algorithms 1 and 2, respectively. The serial algorithm declares a bid to have won immediately (as previously described), whilst the parallel version must split the work into separate bid generation and reconciliation phases, as bids must now be communicated between threads. As the process is memory bound, this additional phase essentially doubles the time, so significant speedup is required for the parallel code to outperform the serial code.

The termination conditions for both algorithms are the same, and the basis for these is illustrated in Figures 2.1 and 2.2. These show the convergence of the serial auction algorithm for two symmetric problems taken from our test set (see Section 4 for details) that are chosen to demonstrate typical behaviour (one converges almost immediately while the other takes a substantial number of iterations). We define the effectiveness of a matching $\mathcal{M}$ as the reduction in the number of delayed pivots compared to the reduction for an optimal matching $\mathcal{M}^*$ calculated using `MC64`. That is, we use the following formula

$$\%\text{effectiveness} = 100 \times \frac{\text{ndelay}_\phi - \text{ndelay}_\mathcal{M}}{\text{ndelay}_\phi - \text{ndelay}_{\mathcal{M}^*}},$$

where ndelay is the number of delayed pivots and ϕ is the empty set and denotes no scaling. The figures demonstrate that $|\mathcal{M}|$ and the effectiveness of the matching do not correlate well. However, in all our tests we observed that a matching of high cardinality was sufficient to achieve high effectiveness (but we could stop much earlier in some cases). With the further observation that later iterations are much cheaper than earlier iterations as they involve many fewer unmatched columns, convergence to a high cardinality matching is a good stopping criterion.

Our preliminary experiments showed the quality of the matching to be sensitive to the choice of ϵ at each iteration. The strategy described in the algorithms is based on one used by Sathe et al. [23] (which differs by initializing $\epsilon = 16/(n + 1)$) and was found to be effective.

Finally, the question arises as to the scaling to apply to unmatched rows and columns. We first comment that the transform (2.5) can lead to large entries in D_c and small entries in D_r that (by construction) cancel to give a moderate scaling of A. However, this prevents us from merely working in the a_{ij} space and choosing $u = 0$ or $v = 0$ for unmatched rows or columns: if the unmatched row contains an entry in a matched column (or vice versa) it ends up very poorly scaled. Thus, values for unmatched rows and columns must be specified in w_{ij} space and transformed back to a_{ij} space.

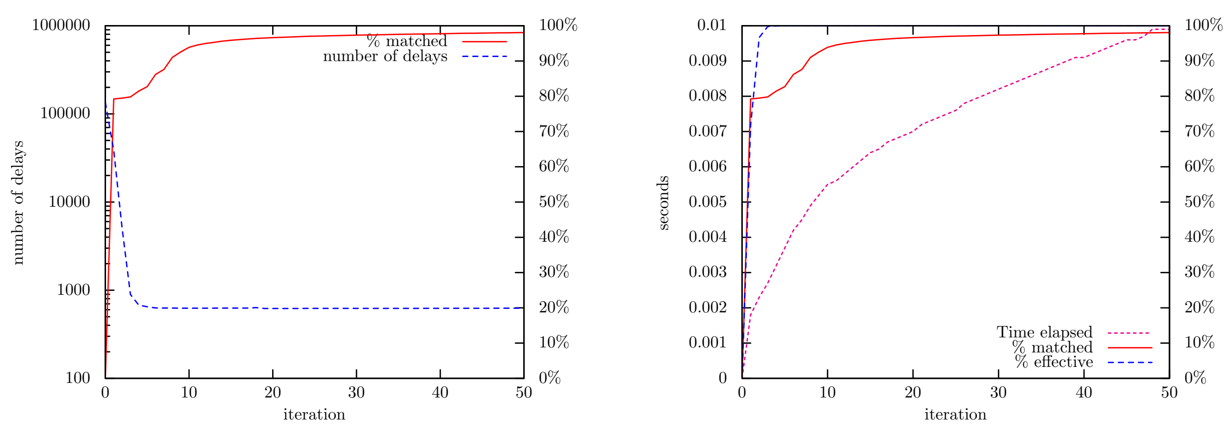
We observe that the $pval = w_{ij} - u_j$ value calculated most recently for a column j makes a good guess for v_j, as it corresponds to the value of the dual variable for a recent trial matching. Each column has such

Algorithm 1 Serial auction algorithm

Input: Size n, positive weights w_{ij}, iteration limit *maxitr*
Output: Matching $\mathcal{M}$, dual variables u
Initialise: $\mathcal{M} = \phi$; $u = 0$; $\epsilon = 0.01$
for $itr = 1, maxitr$ **do**
  if (terminate()) **exit**
  $\epsilon = \min(1.0, \epsilon + 1/(n+1))$
  for each unmatched column j that is not unmatchable **do**
    Find $i = \arg\max_k \{w_{kj} - u_k\}$, $pval = w_{ij} - u_i$ and $qval = \max_{k \neq i} \{w_{kj} - u_k\}$
    if $(pval > 0)$ **then**
      ! Bid for row i and win immediately
      $u_i = u_i + pval - qval + \epsilon$
      Add (i, j) to $\mathcal{M}$
      if ($(i, k) \in \mathcal{M}$ for some k) mark k as unmatched and remove (i, k) from $\mathcal{M}$
    else
      ! No bid is worthwhile
      Mark column j as unmatchable
    end if
  end for
end for

terminate(): *! Returns true if algorithm should terminate*
if $(|\mathcal{M}| = n)$ **return** true
if ($|\mathcal{M}|$ unchanged for 10 iterations **and** $|\mathcal{M}|/n > 0.9$) **return** true
if ($|\mathcal{M}|$ unchanged for 100 iterations) **return** true
return false

Figure 2.1: Convergence behaviour of the serial auction algorithm on the Schenk_IBMNA/c-62 matrix.



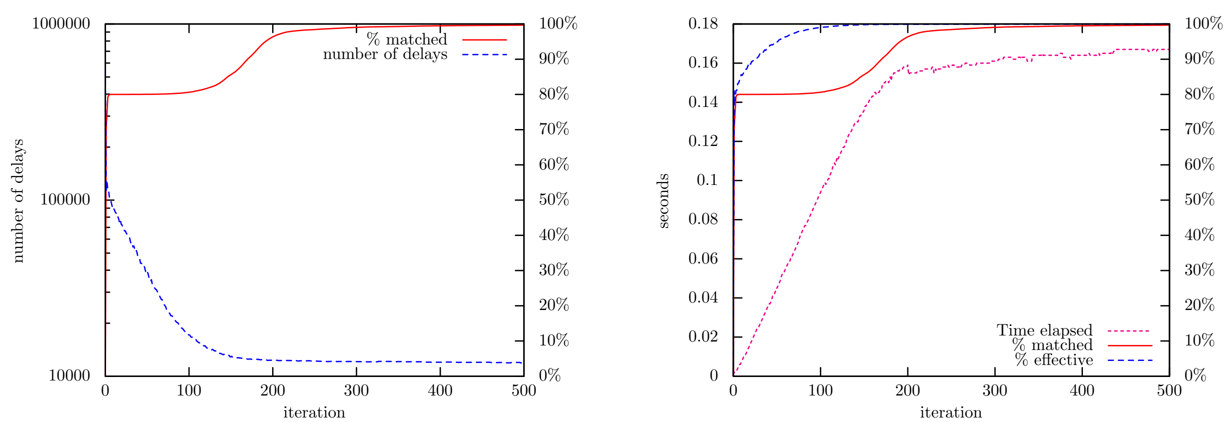
Algorithm 2 Parallel auction algorithm

Input: Size n, positive weights w_{ij}, iteration limit $maxitr$
Output: Matching $\mathcal{M}$, dual variables u
Running on P threads, DEFAULT(private), SHARED(n, w_{ij}, $maxitr$, $\mathcal{M}$)
Initialise: $\mathcal{M} = \phi$; $u = 0$; $\epsilon = 0.01$
Partition the columns equally between the threads
for $itr = 1, maxitr$ **do**
  if (terminate()) **exit**
  $\epsilon = \min(1.0, \epsilon + 1/(n+1))$
  generate_bids()
  —**BARRIER**—
  determine_winners()
  —**BARRIER**—
  Reassign columns so that each thread has approximately $(n - |\mathcal{M}|)/P$ columns
end for

generate_bids():
for each unmatched column j owned by this thread **do**
  Find $i = \arg\max_k \{w_{kj} - u_k\}$, $pval = w_{ij} - u_i$ and $qval = \max_{k \neq i}\{w_{kj} - u_k\}$
  if $(pval > 0)$ **then**
    $u_i = u_i + pval - qval + \epsilon$
    Delete any existing bid by this thread for i.
    Record bid (i, j) and u_i.
  end if
end for

determine_winners():
for all rows i **do**
  Find highest bid (i, j) with value $pval$ among all threads
  Add (i, j) to $\mathcal{M}$
  if($(i, k) \in \mathcal{M}$ for some k) mark k as unmatched and remove (i, k) from $\mathcal{M}$
  Update local $u_i = pval$
end for

Figure 2.2: Convergence behaviour of the serial auction algorithm on the GHS_indef/ncvxqp5 matrix.



a *pval* calculated at least once during the execution of the algorithm (unless the column is empty, in which case its scaling is irrelevant). However, storing this value creates additional memory traffic that may not be desirable, particularly in the parallel case where false sharing may be an issue (v_j can always be calculated during post processing for matched columns). Regardless, u_i for unmatched rows cannot be found as a side-effect of execution in this fashion. We found that most reasonable values for such dual variables work, but for simplicity our code initialises $u_i = 0$ and $v_j = \max_i w_{ij}$ (note that we found $u_i = 0, v_j = 0$ performed considerably less well in our tests).

3 The Approximation Algorithm

We start with some definitions that we require in our description of the approximation algorithm. We assume that all edge weights w_{ij} are positive and define $W(\mathcal{M}) = \sum_{(i,j)\in\mathcal{M}} w_{ij}$ be the *total weight* of a matching $\mathcal{M}$ for a graph $\mathcal{G}$. Let $\mathcal{M}^*$ be an optimal matching, then an *α-approximation matching* algorithm is defined to be a matching algorithm that guarantees to find $\mathcal{M}$ such that $W(\mathcal{M}) \geq \alpha W(\mathcal{M}^*)$. An edge (i, j) of $\mathcal{G}$ with weight w_{ij} is defined to be locally dominant if $\arg\max_k \{w_{kj}\} = \arg\max_k \{w_{ki}\} = w_{ij}$.

The greedy approach of Avis [1] provides a simple $\frac{1}{2}$-approximation algorithm. This matches the heaviest edges in decreasing order if they are locally dominant (this is roughly equivalent to a single round of the auction algorithm). In this paper, we use the more advanced parallel implementation of Halappanavar et al. [12]. Rather than the sorting-based approach of Avis, a queue-based mechanism is used, as originally proposed by Preis [21]. The central concept is that, at each iteration, locally dominant edges are added to the matching; the matched edges and their vertices are removed and the resulting reduced graph is considered at the next iteration. The algorithm is outlined as Algorithm 3. Here *adj*(j) denotes the set of vertices that are neighbours of the vertex j.

The algorithm has two phases. In the first, a list of the locally dominant edges in $\mathcal{G}$ is made. This is done in parallel by passing through the graph data twice. On the first pass, for each vertex j the neighbour p_j that maximises w_{kj} is found (ties are broken by taking the lowest such p_j). The edge (j, p_j) is held as the candidate locally dominant edge for vertex j. A second pass confirms whether a candidate edge is a locally dominant edge by checking if $p_{p_j} = j$. Each locally dominant edge (j, p_j) is added to the matching, and the vertex j is added to the list $\mathcal{Q}$ of vertices to be removed from $\mathcal{G}$ for the next iteration. Observe that $k = p_j$ will be added to this list when vertex k is processed.

The second phase of the algorithm consists of a number of iterations, each of which removes vertices from $\mathcal{G}$ and, for each remaining vertex i that is a neighbour of a vertex in $\mathcal{Q}$, updates its candidate locally dominant edge. The list of vertices for removal can be iterated over in parallel as long as updates to the list of candidate edges and additions to the vertex removal list $\mathcal{Q}'$ for the next iteration are performed atomically. The unmatched neighbours of each vertex j that is to be removed must be checked. Specifically, any unmatched neighbour i for which (i, j) was the candidate edge must have its candidate edge updated to the edge (i, p_i), where p_i is the unmatched neighbour of i that maximises w_{ki}. If edge (i, p_i) is locally optimal in the reduced graph, it is added to the matching and vertices i and p_i are included in the removal list that is to be used on the next iteration.

A simple example to illustrate the approximation algorithm in given in Figure 3.1. Here the vertices are A to F and the integers are the edge weights. The arrows from each vertex indicate which is the candidate locally dominant edge. Thus, on the first pass of phase 1, the edge (A, B) is the candidate for vertices A and B, (C, A) is the candidate for vertex C, (F, C) is the candidate for F, and so on. On the second pass of phase 1, the candidate edge (A, B) is confirmed as a locally dominant edge (since it is the local candidate for both A and B). This edge is added to the matching $\mathcal{M}$ (denoted by the double line) and the vertices A and B are added to the vertex removal list. On the first iteration of phase 2, the candidate edges for vertices C and E are recomputed; the edge (C, F) is found to be locally dominant and added to $\mathcal{M}$, with C and F added to the removal list. Finally, on the second iteration, the candidate edge for vertex D is recomputed, the edge (D, E) is locally dominant and added to $\mathcal{M}$. The final matching has total $W(\mathcal{M}) = 11+9+3 = 23$. Note that this is not the optimal matching, which has $W(\mathcal{M}) = 10 + 7 + 9 = 26$.

Recall that our interest is in the problem (1.1) but the approximation algorithm addresses the maximum

Algorithm 3 Parallel $\frac{1}{2}$-approximation algorithm

Input: Graph $\mathcal{G}$ with positive edge weights w_{ij}
Output: Matching $\mathcal{M}$
Running on P threads, DEFAULT(private), SHARED(w_{ij}, $\mathcal{M}$, $\mathcal{Q}$, $\mathcal{Q}'$, p)
Initialise: $\mathcal{M} = \phi$; $\mathcal{Q} = \phi$
Partition vertices equally between threads
! Phase 1: Identify locally dominant edges in original graph
! First establish candidate locally dominant edges (in parallel)
for each vertex j **do**
  Find $p_j = \arg\max_i \{w_{ij}\}$ *! Tie break by lowest index*
end for
—BARRIER—
! Confirm choice where candidates agree (in parallel)
for each vertex j **do**
  if $p_{p_j} = j$ **then**
    ! Edge (j, p_j) is locally dominant
    Add (j, p_j) to the matching $\mathcal{M}$
    Add j to the vertex removal list $\mathcal{Q}$
  end if
end for
! Phase 2: Reduce graph by removing vertices in $\mathcal{Q}$, finding new locally dominant edges as we go
while $\mathcal{Q} \neq \phi$ **do**
  $\mathcal{Q}' = \phi$! Removal list for next iteration
  —BARRIER—
  for all $j \in \mathcal{Q}$ **do**
    ! Remove vertex j and update candidate locally dominant edges of its neighbours (done in parallel)
    for each $i \in adj(j)$ such that $p_i = j$ **do**
      ProcessVertex($i, \mathcal{Q}'$)
    end for
  end for
  —BARRIER—
  $\mathcal{Q} \leftarrow \mathcal{Q}'$ *! Set removal list for next iteration*
end while

ProcessVertex($i, \mathcal{Q}'$):
! Update candidate edge for vertex i
Find $p_i = \arg\max_{k \notin \mathcal{M}} \{w_{ki}\}$ *! Tie break by lowest index*
if $p_{p_i} = i$ **then**
  ! After update, edge (i, p_i) is now locally dominant
  Add (i, p_i) to the matching $\mathcal{M}$
  Add i and p_i to the removal list $\mathcal{Q}'$
end if

Figure 3.1: Approximation algorithm example (on a general graph $\mathcal{G}$ rather than a bipartite graph).

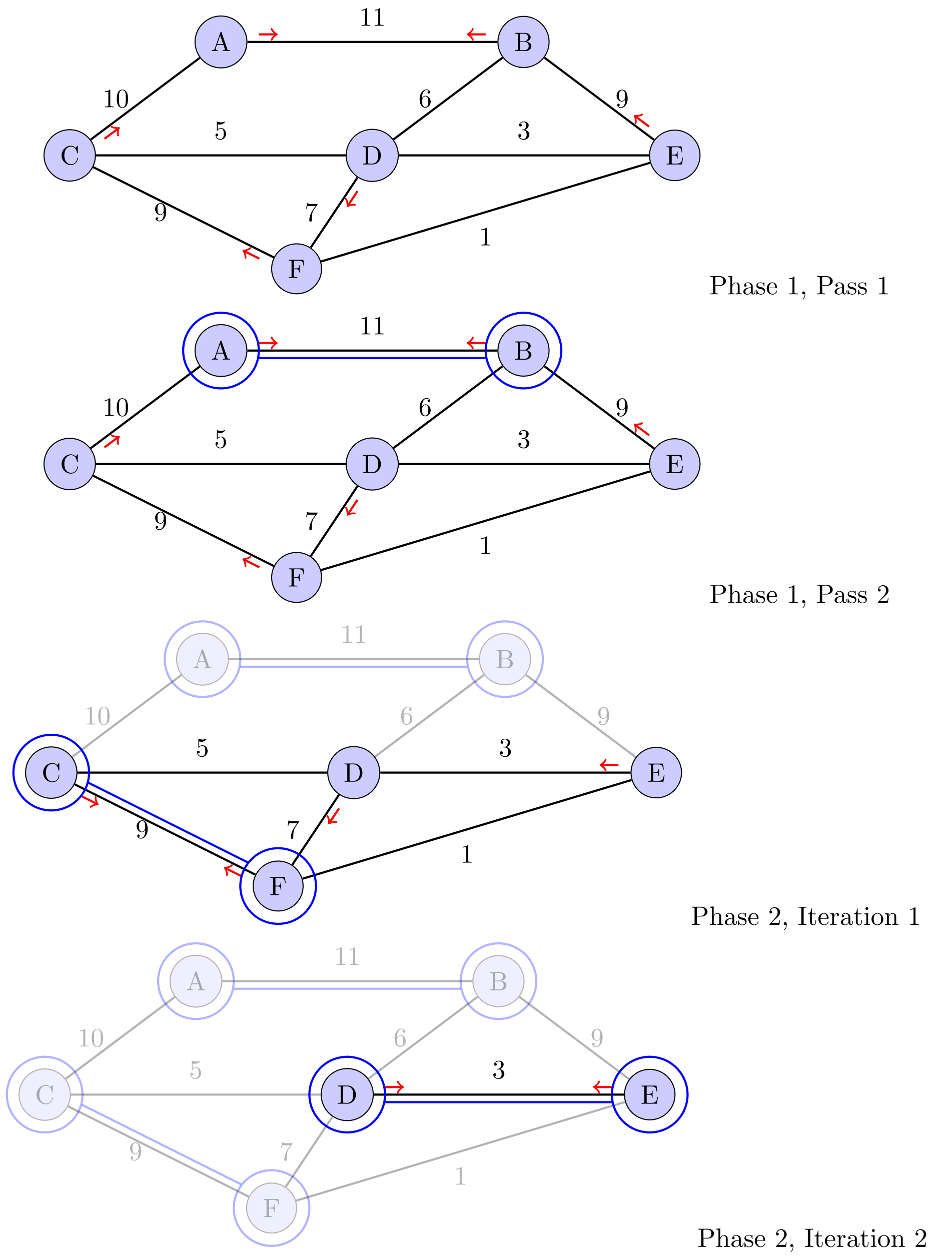


Figure 4.1: Description of machine used for numerical experiments

Processor	2 × Intel Xeon E5-2687W
Physical Cores	16
Memory	64GB
Compiler	ifort 12.1.0
BLAS	MKL 10.3 update 6
L1/L2 cache (per core)	32KB / 256KB
L3 cache (shared)	20MB
Compiler flags	`ifort -O3 -xAVX -no-prec-div -ip -openmp`

weight maximum cardinality problem. Applying the transformation

$$w_{ij} = \log |a_{ij}| + (\alpha - c_j),$$

where $\alpha = \max_{i,j}\{c_j - \log |a_{ij}|\}$ and $c_j = \max_i \log |a_{ij}|$, we obtain $w_{ij} > 0$ and the final matching provides an approximate solution to (1.1). As the approximation algorithm does not use dual variables, we define

$$u_i = 0 \qquad \forall\, i,$$
$$v_j = \begin{cases} w_{ij}, & (i,j) \in \mathcal{M}, \\ c_j, & \text{otherwise.} \end{cases}$$

This guarantees the equality condition $w_{ij} - u_i - v_j = 0$ for edges in $\mathcal{M}$. The choice $v_j = c_j$ for unmatched columns ensures that the largest entry in the column is scaled towards 1, and that the scaling is appropriate after the w_{ij} transformation is reversed.

4 Computational Experiments

For the purposes of our experiments, we use four sets of of symmetric indefinite test problems drawn from the University of Florida Sparse Matrix Collection [6] and detailed in Table 4.1. Test Sets 1 and 2 are matrices that do not significantly benefit from an `MC64` scaling compared to no scaling or the application of a cheap norm equilibration algorithm. The purpose of these sets is to assess the cost of applying a scaling algorithm when scaling is not actually needed. For the problems in Test Set 1, the time to run `MC64` is high, while for those in Test Set 2, `MC64` represents a much smaller overhead in the solver time. Test Sets 3 and 4 are drawn from our recent paper on pivoting techniques for difficult problems [17]. Test Set 3 is a set of problems for which using the `MC64` scaling is sufficient to reduce the number of delayed pivots to reasonable levels, while Test Set 4 comprises those problems that require a matching-based ordering and scaling to achieve this (further details on matching-based orderings are given in Section 4.3).

All our tests are performed on the 16 core machine detailed in Figure 4.1. All times and results are based on `HSL_MC64` version 2.4.0 and the sparse direct solver `HSL_MA97` version 2.2.0 [14, 15]. Both are run with default settings, except where otherwise stated. We use the letters OOM to indicate a problem ran out of memory during the factorization phase of `HSL_MA97` because of the generation of too many delayed pivots.

4.1 Scalability

Figure 4.2 shows the speedup of our implementation of the parallel auction algorithm against our implementation of the serial auction algorithm. The problems in the four test sets have been amalgamated into a single set and then rearranged in order of increasing number of entries in A. It is clear that the matrix must have a large number of entries before parallelization is worthwhile (a significant speedup is required to overcome the overhead of the separate bid generation and reconciliation phases). Based on these results, we only recommend the parallel algorithm if $nz(A) > 2 \times 10^6$.

We found that no appreciable parallel speedup was achieved by running the approximation algorithm in parallel (however, slowdown was observed on the smallest problems).

Table 4.1: Test sets used for testing. $nz(A)$ denotes the number of entries in the lower triangular part of A; $nz(L)$ and $nflops$ denote the number of entries and floating-point operations, respectively, returned by the analyse phase of the HSL_MA97 solver.

Test Set 1

Identifier	n	$nz(A)$	$nz(L)$	$nflops$	Description/Application
Schenk_IBMNA/c-54	31793	385987	1.0374×10^6	7.6222×10^7	Non-linear optimization
Boeing/pcrystk02	13965	968583	4.3969×10^6	1.9128×10^9	Crystal vibration
HB/bcsstk30	28924	1036208	3.9946×10^6	9.3470×10^8	Off-shore generator platform
GHS_indef/boyd1	93279	1211231	6.5262×10^5	4.6722×10^6	Convex QP
Rothberg/gearbox	153746	9080404	3.8829×10^7	2.1001×10^{10}	Aircraft flap actuator
Gupta/gupta3	16783	9323427	6.3516×10^6	3.1067×10^9	Linear programming
Andrianov/mip1	66463	10352819	4.5317×10^7	1.4552×10^{11}	Mixed integer programming
DNVS/fullb	199187	11708077	7.6518×10^7	1.0081×10^{11}	Full-breadth barge
DNVS/troll	213453	11985111	6.6466×10^7	5.6414×10^{10}	Structural problem
Chen/pkustk14	151926	14836504	1.0945×10^8	1.4796×10^{11}	Tall building

Test Set 2

Identifier	n	$nz(A)$	$nz(L)$	$nflops$	Description/Application
GHS_indef/copter2	55476	759952	1.0444×10^7	5.4949×10^9	CFD helicopter rotor blade
Cunningham/qa8fk	66127	1660579	2.4259×10^7	2.1322×10^{10}	3D acoustic FE stiffness matrix
Boeing/crystk03	24696	1751178	9.8413×10^6	5.7087×10^9	Crystal vibration
Lin/Lin	256000	1766400	1.1359×10^8	2.7918×10^{11}	Structural eigenvalue problem
Boeing/bcsstk39	46772	2060662	7.0169×10^6	1.6613×10^9	Rocket booster
Boeing/pct20stif	52329	2698463	1.1952×10^7	9.1960×10^9	Engine block
Oberwolfach/filter3D	106437	2707179	2.0099×10^7	7.6986×10^9	3D heat transfer PDE
Oberwolfach/t3dh	79171	4352105	4.8137×10^7	6.9077×10^{10}	Micropyros thruster
Koutsovasilis/F2	71505	5294285	2.1290×10^7	1.1450×10^{10}	Piston rod
PARSEC/Ge99H100	112985	8451395	6.5419×10^8	7.0120×10^{12}	Density function theory

Test Set 3

Identifier	n	$nz(A)$	$nz(L)$	$nflops$	Description/Application
GHS_indef/ncvxqp1	12111	73963	1.6839×10^6	7.2793×10^8	Non-convex QP
GHS_indef/cvxqp3	17500	122462	3.1398×10^6	1.7670×10^9	Convex QP
GHS_indef/ncvxqp5	62500	424966	1.2052×10^7	9.7223×10^9	Non-convex QP
GHS_indef/ncvxqp3	75000	499964	1.9007×10^7	2.0692×10^{10}	Non-convex QP
GHS_indef/stokes128	49666	558594	2.9813×10^6	3.6881×10^8	FE model Stokes problem
Schenk_IBMNA/c-62	41731	559341	8.4562×10^6	8.0164×10^9	Optimization problem
Schenk_IBMNA/c-64	51035	707985	1.6971×10^6	1.3801×10^8	Optimization problem
GHS_indef/boyd2	466316	1500397	2.5854×10^6	1.5582×10^7	Optimization problem

Test Set 4

Identifier	n	$nz(A)$	$nz(L)$	$nflops$	Description/Application
GHS_indef/bratu3d	27792	173796	6.2769×10^6	4.4174×10^9	Optimization
GHS_indef/cont-201	80595	438795	4.7815×10^6	8.6513×10^8	Convex QP
GHS_indef/ncvxqp7	87500	574962	2.4731×10^7	3.0939×10^{10}	Non-convex QP
GHS_indef/cont-300	180895	988195	1.1744×10^7	2.9559×10^9	Convex QP
GHS_indef/darcy003	389874	2101242	8.1587×10^6	5.5664×10^8	Mixed FE model Darcy's equation
TSOPF/TSOPF_FS_b300_c2	56814	8767466	2.1433×10^7	8.9629×10^9	Optimal power flow

Figure 4.2: Speedup of the parallel auction algorithm against the serial auction algorithm. Matrices from all four test sets are ordered by increasing $nz(A)$.

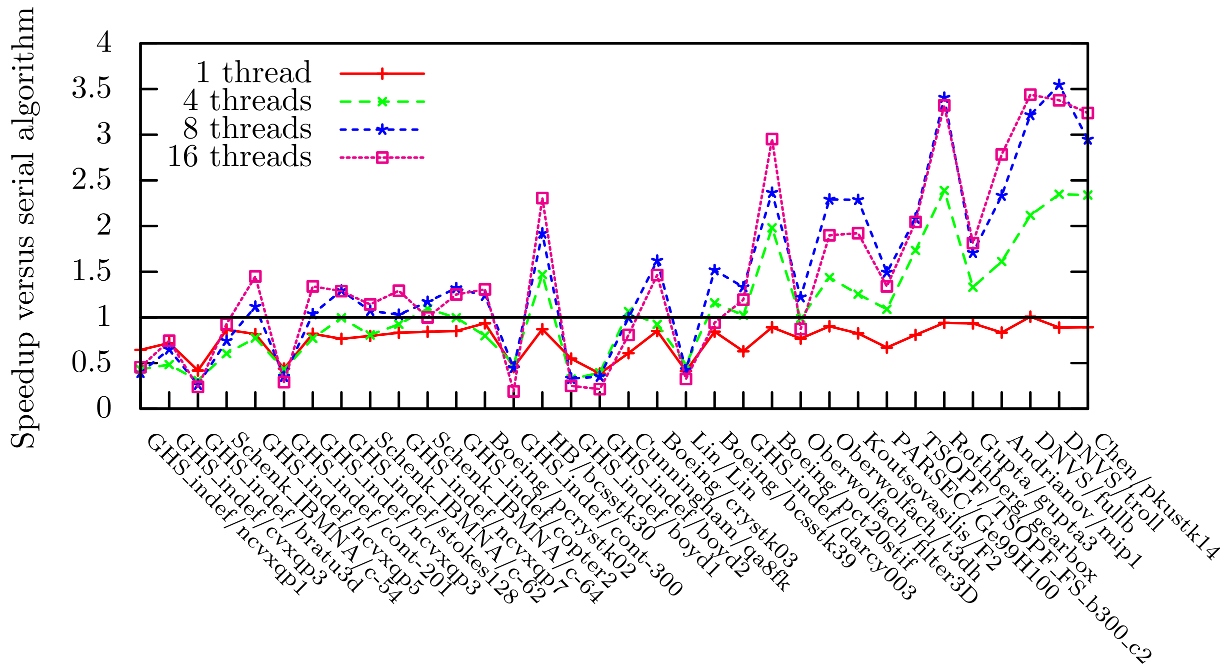


Table 4.2: Cardinality of the matching obtained using each algorithm as a percentage of the entries matched. Cardinalities less than 99% are in bold. Numbers in brackets give deficiencies.

Problem	Hungarian	sAuction	pAuction	Approx
Schenk_IBMNA/c-54	100	**98.89** (353)	**98.85** (367)	99.33 (213)
Boeing/pcrystk02	100	99.68 (44)	99.69 (43)	**98.22** (249)
HB/bcsstk30	100	99.57 (124)	99.64 (105)	**97.22** (803)
GHS_indef/boyd1	100	99.99 (8)	99.99 (9)	99.99 (11)
Rothberg/gearbox	100	99.88 (180)	99.84 (252)	**97.85** (3303)
Gupta/gupta3	100	99.79 (36)	99.65 (58)	**96.57** (576)
Andrianov/mip1	100	99.60 (263)	99.67 (217)	**97.01** (1985)
DNVS/fullb	100	99.89 (218)	99.89 (217)	**97.63** (4726)
DNVS/troll	100	99.89 (242)	99.91 (195)	**97.85** (4593)
Chen/pkustk14	100	99.84 (249)	99.81 (287)	**98.33** (2533)
GHS_indef/copter2	100	99.78 (123)	99.77 (127)	**94.79** (2890)
Cunningham/qa8fk	100	100	100	100
Boeing/crystk03	100	100	100	100
Lin/Lin	100	100	100	100
Boeing/bcsstk39	100	100	100	99.54 (214)
Boeing/pct20stif	100	99.61 (202)	99.75 (129)	**97.33** (1399)
Oberwolfach/filter3D	100	100	100	100
Oberwolfach/t3dh	100	100	100	100
Koutsovasilis/F2	100	100	100	100
PARSEC/Ge99H100	100	100	100	100
GHS_indef/ncvxqp1	100	**96.28** (450)	**96.28** (450)	**61.08** (4714)
GHS_indef/cvxqp3	100	**98.06** (340)	**98.05** (341)	**57.14** (7500)
GHS_indef/ncvxqp5	100	99.84 (100)	99.82 (112)	**83.50** (10315)
GHS_indef/ncvxqp3	100	**96.00** (3000)	**96.00** (3000)	**68.84** (23367)
GHS_indef/stokes128	100	99.87 (67)	99.88 (62)	**67.01** (16384)
Schenk_IBMNA/c-62	100	99.68 (133)	99.74 (109)	**84.42** (6502)
Schenk_IBMNA/c-64	100	99.04 (488)	99.06 (479)	**93.78** (3173)
GHS_indef/boyd2	100	100	100	**93.38** (30857)
GHS_indef/bratu3d	100	100	100	**94.53** (1519)
GHS_indef/cont-201	100	100	100	99.75 (199)
GHS_indef/ncvxqp7	100	**98.06** (1700)	**98.06** (1700)	**61.02** (34109)
GHS_indef/cont-300	100	100	100	99.83 (299)
GHS_indef/darcy003	100	99.98 (63)	99.98 (77)	99.90 (383)
TSOPF/TSOPF_FS_b300_c2	100	99.93 (38)	99.93 (40)	**75.03** (14187)

4.2 Effectiveness of algorithms: scaling only

Table 4.2 provides results on the quality of the matching achieved by each of the matching algorithms in terms of cardinality, while Table 4.3 measures the effectiveness of the associated scaling by counting the number of delayed pivots when the orderings are run with the `HSL_MA97` solver. Table 4.4 compares the runtime of each algorithm to achieve this. For the parallel auction algorithm, results are given for running on 16 threads.

These tables show that while the approximation algorithm is the fastest, it fails to provide an alternative to the Hungarian algorithm, both in terms of finding a high cardinality matching and reducing the number of delayed pivots. On the other hand, both our serial and parallel auction codes lead to a similar number of delayed pivots as for the Hungarian algorithm on all but one problem (GHS_indef/ncvxqp1), where they perform slightly worse.

The ncvxqp1 discrepancy is an example where our stopping conditions for the auction algorithm cause it to terminate with a 96.3% match after 101 iterations, taking approximately 0.004 seconds. If we instead run for 383 iterations (which takes 0.007 seconds), we achieve a 96.6% match resulting in only 10,986 delayed pivots, which is comparable to the Hungarian algorithm. However, this run includes 268 iterations where the matching is stuck at 96.3%. Note that, for this problem, a complete matching requires 12,368 iterations and takes 0.013 seconds.

Table 4.5 summarises the numbers in Table 4.4 by showing the fraction of the total factorization time spent in the scaling for each algorithm. The total factorization is taken to be the time to compute the scaling and then to factorize the scaled matrix (the time for pre- and post-processing the matrix data is not included, but is relatively small and easily parallelized). It shows that the use of the auction algorithm generally reduces the proportion of the time spent in scaling the matrix, especially for problems in Test Sets

Table 4.3: Number of delayed pivots reported by HSL_MA97 with different scalings

Problem	None	Hungarian	sAuction	pAuction	Approx
Schenk_IBMNA/c-54	6355	1281	2566	2615	11203
Boeing/pcrystk02	11	11	11	11	11
HB/bcsstk30	16	16	16	16	16
GHS_indef/boyd1	OOM	0	0	0	43671
Rothberg/gearbox	102	101	103	103	110
Gupta/gupta3	41	33	34	33	36
Andrianov/mip1	122	50	51	52	100
DNVS/fullb	145	145	145	144	150
DNVS/troll	150	147	146	146	167
Chen/pkustk14	102	102	101	100	113
GHS_indef/copter2	87	86	72	75	80
Cunningham/qa8fk	0	0	0	0	0
Boeing/crystk03	0	0	0	0	0
Lin/Lin	0	0	0	0	0
Boeing/bcsstk39	0	0	0	0	32
Boeing/pct20stif	43	40	41	41	40
Oberwolfach/filter3D	0	0	0	0	0
Oberwolfach/t3dh	0	0	0	0	0
Koutsovasilis/F2	0	0	0	0	0
PARSEC/Ge99H100	3	3	3	3	1
GHS_indef/ncvxqp1	124018	10303	31462	31462	76197
GHS_indef/cvxqp3	312033	26039	26058	26051	120608
GHS_indef/ncvxqp5	544291	11858	11944	11798	522545
GHS_indef/ncvxqp3	1446194	65161	66027	66014	0
GHS_indef/stokes128	30509	5502	5502	5502	5502
Schenk_IBMNA/c-62	135154	594	669	692	180979
Schenk_IBMNA/c-64	32356	574	570	570	120103
GHS_indef/boyd2	27077	0	0	0	39339
GHS_indef/bratu3d	59569	59657	59590	59657	59650
GHS_indef/cont-201	88299	88276	88276	88276	88284
GHS_indef/ncvxqp7	1697334	272146	273327	273371	1673909
GHS_indef/cont-300	148526	148509	148509	148509	148512
GHS_indef/darcy003	44900	44900	44900	44900	44900
TSOPF/TSOPF_FS_b300_c2	100652	45306	46175	48642	97031

Table 4.4: Time (in seconds) to compute different scalings and the HSL_MA97 time to compute the factorization on 16 cores.

	Scaling				Factor				
	Hungarian	sAuction	pAuction	Approx	None	Hungarian	sAuction	pAuction	Approx
Schenk_IBMNA/c-54	0.20	0.01	0.01	0.00	0.09	0.06	0.05	0.05	0.09
Boeing/pcrystk02	0.16	0.01	0.01	0.00	0.07	0.07	0.05	0.05	0.07
HB/bcsstk30	0.70	0.03	0.01	0.01	0.12	0.12	0.11	0.10	0.12
GHS_indef/boyd1	1.85	0.00	0.01	0.00	OOM	0.06	0.06	0.06	145
Rothberg/gearbox	1.96	0.19	0.06	0.04	0.42	0.42	0.40	0.41	0.42
Gupta/gupta3	1.61	0.06	0.03	0.03	0.39	0.37	0.35	0.35	0.38
Andrianov/mip1	4.71	0.25	0.09	0.03	2.34	2.28	2.26	2.26	2.33
DNVS/fullb	2.27	0.27	0.08	0.05	1.82	1.87	1.78	1.80	1.82
DNVS/troll	2.05	0.26	0.08	0.05	0.68	0.68	0.65	0.66	0.67
Chen/pkustk14	7.61	0.25	0.08	0.06	1.61	1.59	1.57	1.51	1.60
GHS_indef/copter2	0.06	0.02	0.02	0.01	0.15	0.14	0.14	0.14	0.14
Cunningham/qa8fk	0.00	0.00	0.00	0.00	0.36	0.36	0.34	0.34	0.35
Boeing/crystk03	0.00	0.00	0.00	0.00	0.14	0.14	0.12	0.11	0.13
Lin/Lin	0.01	0.00	0.02	0.01	4.57	4.64	4.44	4.62	4.63
Boeing/bcsstk39	0.01	0.00	0.00	0.01	0.17	0.16	0.15	0.14	0.16
Boeing/pct20stif	0.69	0.05	0.02	0.01	0.29	0.29	0.26	0.26	0.28
Oberwolfach/filter3D	0.01	0.01	0.01	0.01	0.14	0.14	0.13	0.13	0.14
Oberwolfach/t3dh	0.01	0.01	0.01	0.01	0.83	0.80	0.80	0.79	0.82
Koutsovasilis/F2	0.01	0.01	0.01	0.01	0.62	0.63	0.59	0.59	0.63
PARSEC/Ge99H100	0.02	0.01	0.01	0.02	259	262	259	261	261
GHS_indef/ncvxqp1	0.09	0.00	0.01	0.00	2.84	0.17	0.37	0.37	0.97
GHS_indef/cvxqp3	0.17	0.02	0.03	0.00	18.2	0.43	0.39	0.39	2.15
GHS_indef/ncvxqp5	0.41	0.17	0.12	0.00	69.1	0.77	0.71	0.73	90.5
GHS_indef/ncvxqp3	3.76	0.20	0.15	0.00	329	2.48	2.53	2.45	408
GHS_indef/stokes128	0.05	0.01	0.01	0.00	0.08	0.05	0.03	0.03	0.05
Schenk_IBMNA/c-62	0.05	0.02	0.02	0.00	6.85	0.44	0.39	0.39	17.0
Schenk_IBMNA/c-64	0.16	0.02	0.02	0.01	1.02	0.05	0.04	0.04	16.1
GHS_indef/boyd2	0.03	0.01	0.05	0.01	15.5	0.09	0.09	0.09	57.6
GHS_indef/bratu3d	0.00	0.00	0.00	0.00	0.86	0.86	0.83	0.85	0.86
GHS_indef/cont-201	0.00	0.00	0.00	0.00	0.19	0.19	0.16	0.16	0.16
GHS_indef/ncvxqp7	2.23	0.21	0.16	0.00	146	14.4	14.4	14.6	174
GHS_indef/cont-300	0.01	0.00	0.02	0.01	0.31	0.31	0.30	0.29	0.31
GHS_indef/darcy003	0.18	0.26	0.22	0.02	0.10	0.10	0.08	0.08	0.11
TSOPF/TSOPF_FS_b300_c2	0.23	0.23	0.11	0.02	1.62	0.48	0.39	0.37	1.54

Table 4.5: Percentage of the total factorization time spent in the scaling algorithm on 16 cores.

Problem	Hungarian	sAuction	pAuction	Approx
Schenk_IBMNA/c-54	76.1	10.4	12.4	1.7
Boeing/pcrystk02	69.0	12.4	9.8	5.9
HB/bcsstk30	85.4	19.3	10.0	6.6
GHS_indef/boyd1	96.9	5.4	17.3	<0.1
Rothberg/gearbox	82.2	32.0	12.3	8.8
Gupta/gupta3	81.2	13.7	8.0	6.8
Andrianov/mip1	67.4	9.8	3.7	1.4
DNVS/fullb	54.9	13.3	4.2	3.0
DNVS/troll	75.2	28.2	10.3	7.6
Chen/pkustk14	82.7	13.6	4.8	3.5
GHS_indef/copter2	30.4	14.9	12.4	3.9
Cunningham/qa8fk	1.0	0.7	0.9	1.3
Boeing/crystk03	2.2	2.0	1.4	3.1
Lin/Lin	0.2	0.1	0.3	0.2
Boeing/bcsstk39	3.5	2.8	3.1	3.9
Boeing/pct20stif	70.6	16.0	6.0	4.3
Oberwolfach/filter3D	7.2	6.7	7.3	6.1
Oberwolfach/t3dh	1.5	1.3	0.7	1.3
Koutsovasilis/F2	2.1	2.0	1.0	2.0
PARSEC/Ge99H100	<0.1	<0.1	<0.1	<0.1
GHS_indef/ncvxqp1	33.7	1.0	2.2	<0.1
GHS_indef/cvxqp3	28.9	5.7	7.6	<0.1
GHS_indef/ncvxqp5	34.9	19.4	13.9	<0.1
GHS_indef/ncvxqp3	60.3	7.4	5.8	<0.1
GHS_indef/stokes128	53.0	22.6	18.8	4.0
Schenk_IBMNA/c-62	9.5	4.3	3.8	<0.1
Schenk_IBMNA/c-64	75.1	33.3	33.8	<0.1
GHS_indef/boyd2	24.7	10.0	33.9	<0.1
GHS_indef/bratu3d	0.1	0.1	0.2	0.1
GHS_indef/cont-201	1.1	0.8	2.8	1.4
GHS_indef/ncvxqp7	13.4	1.4	1.1	<0.1
GHS_indef/cont-300	1.7	1.0	5.1	1.7
GHS_indef/darcy003	64.2	76.8	73.7	13.2
TSOPF/TSOPF_FS_b300_c2	32.9	36.9	23.1	1.2

1 and 3. The approximation algorithm spends a very small proportion of its time in scaling because the factorization time is so much larger.

4.3 Effectiveness of algorithms: ordering and scaling

We now consider the effectiveness of using a matching-based ordering combined with the matching-based scaling. As these techniques are known to be expensive [17], we only consider their application to problems in Tests Sets 3 and 4.

A matching-based ordering involves using a matching to identify 2×2 pivots, compressing the adjacency graph of the matrix such that the sparsity patterns of both members of the 2×2 pivot are merged into a single column before running a fill-reducing ordering on the compressed graph [10, 11]. There are thus three times to consider, (i) the time to run the matching algorithm (given in Table 4.4 of the previous section), (ii) the time to run the whole matching-based ordering routine, including the preprocessing, matching algorithm, graph compression and ordering and (iii) the factorization time using the calculated scaling and ordering. Table 4.6 reports the latter two times, while Table 4.7 demonstrates their ability to reduce the number of delayed pivots required during factorization.

We again see the approximation algorithm does not provide a sufficiently good matching for this approach to be effective. For most of our test problems, the Hungarian algorithm and the serial and parallel auction algorithms give comparable results and are extremely effective in substantially reducing the delayed pivots. However, for the ncvxqp/cvxqp problems, the Hungarian algorithm gives the best results, even for those problems for which the auction algorithms gave quality scalings of comparable quality (Table 4.3). These ncvxqp/cvxqp problems correspond exactly to those for which the cardinality of the auction algorithm matching was less than 99% (Table 4.2). Additional experiments show that by running the serial auction algorithm until a 100% cardinality matching is reached, results comparable to the Hungarian algorithm can

Table 4.6: Matching-based ordering and scaling: Time (in seconds) to compute the ordering and to compute the factorization on 16 cores.

Problem	Ordering and scaling				Factor			
	Hungarian	sAuction	pAuction	Approx	Hungarian	sAuction	pAuction	Approx
GHS_indef/ncvxqp1	0.12	0.04	0.04	0.07	0.42	0.62	0.69	1.45
GHS_indef/cvxqp3	0.23	0.08	0.09	0.09	0.99	0.86	0.90	2.15
GHS_indef/ncvxqp5	0.66	0.41	0.36	0.29	1.66	1.97	1.86	177
GHS_indef/ncvxqp3	4.00	0.49	0.45	0.39	6.67	12.5	11.8	OOM
GHS_indef/stokes128	0.23	0.19	0.19	0.26	0.04	0.03	0.03	0.02
Schenk_IBMNA/c-62	0.25	0.22	0.21	0.26	2.42	1.87	2.13	130
Schenk_IBMNA/c-64	0.37	0.22	0.22	0.28	0.14	0.11	0.11	74.4
GHS_indef/boyd2	17.4	17.6	17.5	17.2	0.10	0.10	0.11	81.2
GHS_indef/bratu3d	0.06	0.05	0.05	0.06	0.18	0.16	0.16	0.16
GHS_indef/cont-201	0.12	0.12	0.12	0.16	0.05	0.04	0.04	0.03
GHS_indef/ncvxqp7	2.51	0.53	0.49	0.50	8.68	10.5	8.02	OOM
GHS_indef/cont-300	0.28	0.27	0.29	0.38	0.10	0.09	0.09	0.08
GHS_indef/darcy003	1.07	1.15	1.12	1.19	0.11	0.10	0.10	0.11
TSOPF/TSOPF_FS_b300_c2	1.68	1.67	1.57	2.91	0.41	0.38	0.36	1.38

Table 4.7: Matching-based ordering and scaling: Number of delayed pivots returned by `HSL_MA97`.

Problem	Hungarian	sAuction	pAuction	Approx
GHS_indef/ncvxqp1	40	28034	29010	82297
GHS_indef/cvxqp3	69	1675	1290	120608
GHS_indef/ncvxqp5	100	130	246	796466
GHS_indef/ncvxqp3	260	15722	16578	OOM
GHS_indef/stokes128	7	4	15	9322
Schenk_IBMNA/c-62	1	0	0	617535
Schenk_IBMNA/c-64	1	22	0	313480
GHS_indef/boyd2	0	0	0	41748
GHS_indef/bratu3d	340	340	340	1450
GHS_indef/cont-201	0	0	0	0
GHS_indef/ncvxqp7	226	9866	7987	OOM
GHS_indef/cont-300	0	0	0	1
GHS_indef/darcy003	121	105	95	339
TSOPF/TSOPF_FS_b300_c2	2429	1403	779	148164

be obtained, while still offering a substantial time saving.

5 Conclusions

We have demonstrated that the auction algorithm fulfills its promise and provides comparable quality to the Hungarian algorithm in the context of scaling and ordering sparse symmetric matrices for use with direct solvers while being significantly faster. By contrast, the very fast $\frac{1}{2}$-approximation algorithm does not at present represent a reasonable alternative.

Our results further show that high quality scalings can be obtained using a sub-optimal matching. However, the matching-based orderings generally require the matching to be of high cardinality to be fully effective in limiting the number of delayed pivots.

As the parallel auction algorithm requires additional work compared to the serial version, we recommend that the user is asked to choose which to use. In our tests, we were able to achieve consistent speedups with the parallel version on matrices that have in excess of two million entries; for smaller problems, it is more efficient to use the serial code.

In this paper, the emphasis has been on sparse symmetric systems. However, matchings are commonly used in the unsymmetric case to permute the matrix in order to obtain a zero-free diagonal of large elements to reduce the need for pivoting (see, for example, the parallel solver SuperLU_DIST [19]). We suspect that this will have similar behavior to that found in the symmetric case when permuting large entries to the subdiagonal: specifically that the sub-optimal termination required to obtain a small run time of the matching algorithm may not provide a matching of sufficient quality to avoid pivoting. Further, the failure to obtain a matching of maximum cardinality could necessitate some additional manipulation to ensure pivot candidates exist at the end of the factorization. A future objective is to investigate how effective the auction algorithm is for unsymmetric solvers and, in particular, whether a parallel implementation can reduce the time for the scaling and ordering without having a detrimental effect on the subsequent factorization (see also [22]).

Finally, we remark that an efficient implementation of the Hungarian algorithm is complicated whereas that of both the serial and parallel versions of the auction algorithm are much more straightforward. We plan to include such implementations within our mathematical software libraries.

Acknowledgements

We would like to thank Mahantesh Halappanavar and Alex Pothen for providing access to their MatchBox Software used for the $\frac{1}{2}$-approximation matching. Thanks also to Sherry Li from Lawrence Berkeley National Laboratory and our colleague Iain Duff for commenting on a draft of this report.

References

[1] D. Avis, *A survey of heuristics for the weighted matching problem*, Networks, 13 (1983), pp. 475–493.

[2] A. Azad, M. Halappanavar, S. Rajamanickam, E. G. Boman, A. Khan, and A. Pothen, *Multithreaded algorithms for maximum matching in bipartite graphs*, in International Parallel and Distributed Processing Symposium, IEEE Computer Society, 2012, pp. 860–872.

[3] D. P. Bertsekas, *A distributed asynchronous relaxation algorithm for the assignment problem*, in 24th IEEE Conference on Decision and Control, vol. 24, IEEE, 1985, pp. 1703–1704.

[4] D. P. Bertsekas and D. A. Castañon, *Parallel synchronous and asynchronous implementations of the auction algorithm*, Parallel Computing, 17 (1991), pp. 707–732.

[5] L. Buš and P. Tvrdík, *Towards auction algorithms for large dense assignment problems*, Computer Optimization and Applications, 43 (2009), pp. 411–436.

[6] T. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Transactions on Mathematical Software, 38 (2011), pp. Article 1, 25 pages.

[7] M. Deveci, K. Kaya, B. Uçar, and U. V. Çatalyürek, *GPU accelerated maximum cardinality matching algorithms for bipartite graphs*, in Euro-Par 2013 Parallel Processing, F. Wolf, B. Mohr, and D. Mey, eds., vol. 8097 of Lecture Notes in Computer Science, Springer, 2013, pp. 850–861.

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

[9] I. Duff and S. Pralet, *Strategies for scaling and pivoting for sparse symmetric indefinite problems*, SIAM J. Matrix Analysis and Applications, 27 (2005), pp. 313–340.

[10] ——, *Strategies for scaling and pivoting for sparse symmetric indefinite problems*, SIAM J. Matrix Analysis and Applications, 27 (2005), pp. 313 – 340.

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

[12] M. Halappanavar, J. Feo, O. Villa, A. Tumeo, and A. Pothen, *Approximate weighted matching on emerging manycore and multithreaded architectures*, International Journal of High Performance Computing Applications, 26 (2012), pp. 413–430.

[13] J. Hogg and J. Scott, *The effects of scalings on the performance of a sparse symmetric indefinite solver*, Technical Report RAL-TR-2008-007, Rutherford Appleton Laboratory, 2008.

[14] ——, *HSL_MA97: a bit-compatible multifrontal code for sparse symmetric systems*, Technical Report RAL-TR-2011-024, Rutherford Appleton Laboratory, 2011.

[15] ——, *New parallel sparse direct solvers for multicore architectures*, Algorithms, 6 (2013), pp. 702–725. Special issue: Algorithms for Multi Core Parallel Computation.

[16] ——, *Optimal weighted matchings for rank-deficient sparse matrices*, SIAM J. Matrix Analysis and Applications, 34 (2013), pp. 1431–1447.

[17] ——, *Pivoting strategies for tough sparse indefinite systems*, ACM Transactions on Mathematical Software, 40 (2013). Article 4, 19 pages.

[18] H. Kuhn, *The Hungarian method for the assignment problem*, Naval Research Logistics Quarterly, 2 (1955), pp. 83–97.

[19] X. Li and J. Demmel, *SuperLU_DIST: A scalable distributed-memory sparse direct solver or unsymmetric linear systems*, ACM Transactions on Mathematical Software, 29 (2003), pp. 110–140.

[20] K. Mehlhorn and S. Näher, *The LEDA Platform of Combinatorial and Geometric Computing*, Cambridge University Press, 1999.

[21] R. Preis, *Linear time $\frac{1}{2}$-approximation algorithm for maximum weighted matching in general graphs*, in 16th Symposium on Theoretical Aspects of Computer Science (STACS), 1999, pp. 259–269.

[22] J. Riedy, *Making Static Pivoting Scalable and Dependable*, PhD thesis, EECS Department, University of California, Berkeley, Dec 2010.

[23] M. Sathe, O. Schenk, and H. Burkhart, *An auction-based weighted matching implementation on massively parallel architectures*, Parallel Computing, 38 (2012), pp. 595–614.

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

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