Bipartite Matching Heuristics with Quality Guarantees on Shared Memory Parallel Computers

Fanny Dufossé, Kamer Kaya, Bora Uçar

To cite this version:
Fanny Dufossé, Kamer Kaya, Bora Uçar. Bipartite Matching Heuristics with Quality Guarantees on Shared Memory Parallel Computers. 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS), May 2014, Phoenix, Arizona, United States. 10.1109/IPDPS.2014.63. hal-01235162

HAL Id: hal-01235162
https://inria.hal.science/hal-01235162

Submitted on 28 Nov 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Abstract—We propose two heuristics for the bipartite matching problem that are amenable to shared-memory parallelization. The first heuristic is very intriguing from parallelization perspective. It has no significant algorithmic synchronization overhead and no conflict resolution is needed across threads. We show that this heuristic has an approximation ratio of around 0.632. The second heuristic is designed to obtain a larger matching by employing the well-known Karp-Sipser heuristic on a judiciously chosen subgraph of the original graph. We show that the Karp-Sipser heuristic always finds a maximum cardinality matching in the chosen subgraph. Although the Karp-Sipser heuristic is hard to parallelize for general graphs, we exploit the structure of the selected subgraphs to propose a specialized implementation which demonstrates a very good scalability. Based on our experiments and theoretical evidence, we conjecture that this second heuristic obtains matchings with cardinality of at least 0.866 of the maximum cardinality. We discuss parallel implementations of the proposed heuristics on shared memory systems. Experimental results, for demonstrating speed-ups and verifying the theoretical results in practice, are provided.

I. INTRODUCTION

We consider the maximum cardinality bipartite matching problem. A matching in a graph is a set of edges no two of which share a common vertex. The maximum cardinality matching problem asks for a matching of maximum size. There are a number of polynomial time algorithms to solve this problem exactly. The smallest worst-case time complexity of the known algorithms is \( O(\sqrt{n}\tau) \) for a bipartite graph with \( n \) vertices and \( \tau \) edges—the first of such algorithms is described by Hopcroft and Karp [17]. There is considerable interest in simpler and faster algorithms that have some approximation guarantee. Such cheap algorithms are used as a jump-start routine by the current state of the art matching algorithms [11], [24]. Most of these heuristics obtain good results in practice, but their worst-case guarantee is only around 1/2. A well-known heuristic, called Karp-Sipser (KS) heuristic [19], finds maximum cardinality matchings in highly sparse (random) graphs but does not have a constant ratio approximation for denser ones (this algorithm will be reviewed later in Section II). Algorithms that achieve an approximation ratio of \( 1 - 1/e \), where \( e \) is the base of the natural logarithm are designed for the online case [20]. Many of these algorithms are sequential in nature in that a sequence of greedy decisions are made in the light of the previously made decisions.

Let \( G = (V_R \cup V_C, E) \) be a bipartite graph, where \( V_R \) and \( V_C \) are two vertex classes and \( E \) is the edge set. \( G \) can be represented as a sparse matrix \( A \). Each row (column) of \( A \) corresponds to a unique vertex in \( V_R \) (in \( V_C \)) so that \( a_{ij} = 1 \) if and only if \( (v_i, v_j) \in E \). We first assume that bipartite graphs have two properties: (i) the same number of vertices in both vertex classes; (ii) each edge appears in a matching that contains all vertices. These bipartite graphs correspond to square, fully indecomposable matrices, or block diagonal matrices with each block being fully indecomposable. These assumptions simplify the theoretical analysis. Later on, we discuss the bipartite graphs without these properties and demonstrate (in Section IV) that the proposed heuristics deliver results that concur with those of the assumed case.
II. Notation and Preliminaries

Using the analogy between a matrix and a bipartite graph, we refer to the vertices in the two classes as row and column vertices. The number of edges incident on a vertex is called its degree. A path in a graph is a sequence of vertices such that each consecutive vertex pair share an edge. A vertex is reachable from another one, if there is a path between them. The connected components of a graph are the equivalence classes of vertices under the “is reachable from” relation. A cycle in a graph is a path whose start and end vertices are the same. A simple cycle is a cycle with no vertex repetitions. A tree is a connected graph with no cycles. A spanning tree of a connected graph $G$ is a tree containing all vertices of $G$.

A. Matching

A matching $M$ in a graph $G = (V_R \cup V_C, E)$ is a subset of edges $E$ where a vertex in $V_R \cup V_C$ is in at most one edge in $M$. Given a matching $M$, a vertex $v$ is said to be matched by $M$ if $v$ is in an edge of $M$, otherwise $v$ is called unmatched. If all the vertices are matched by $M$, then $M$ is said to be a perfect matching. The cardinality of a matching $M$, denoted by $|M|$, is the number of edges in $M$. The maximum cardinality matching problem asks for a matching of maximum size. There are a number of well-known, exact, and polynomial-time algorithms for this problem. A recent paper [21] gives a classification of those algorithms.

Parallel (exact) matching algorithms on modern architectures have been recently investigated. Azad et al. [4] study the implementations of a set of known bipartite graph matching algorithms on shared memory systems. Deveci et al. [9], [10] investigate the implementation variants of some matching algorithms on GPU. There are quite good speedups reported in these implementations, yet there are non-trivial instances where parallelism does not help in any of them.

Our focus is on matching heuristics that have linear running time complexity and good quality guarantees on the size of the matching. A survey of matching heuristics is given by Kaya et al. [11] and Langguth et al. [24]. Recent studies focusing on approximate matching algorithms on parallel systems include heuristics for graph matching problem [6], [15], [16] and also heuristics used for initializing bipartite matching algorithms [4].

The simplest heuristic is called the cheap matching that has two variants in the literature. The first variant randomly visits the edges and matches the two endpoints of an edge if they are both available. The theoretical performance guarantee of this heuristic is $1/2$, i.e., the heuristic delivers matchings of size at least half of the maximum matching cardinality. This is analyzed theoretically [13] and shown to obtain results that are near the worst-case on certain classes of graphs. The second variant of the cheap matching heuristic repeatedly selects a random vertex and matches it with a random neighbor. The matched vertices, along with the ones which become isolated, are removed from the graph and the process continues until the whole graph is consumed. This variant also has a $1/2$ worst-case approximation guarantee (see for example a proof by Pothen and Fan [27]), and it is somewhat better ($0.5 + \epsilon$ for $\epsilon \geq 0.000025$ [2] which has been recently improved to $\epsilon \geq 1/256$ [26]).

We make use of a heuristic algorithm, called Karp-Sipser (KS). We summarize it and refer the reader to original paper [19]. The theoretical foundation of the KS heuristic is that if there is a vertex $v$ with exactly one neighbor ($v$ is called degree-one), then there is a maximum cardinality matching in which $v$ is matched with its neighbor. That is, matching $v$ with its neighbor is an optimal decision. Using this, the KS heuristic runs as follows. Check whether there is a degree-one vertex; if so then match the vertex with its unique neighbor and delete both vertices (and the edges incident on them) from the graph. Continue this way until the graph has no edges (in which case we are done) or all remaining vertices have degree larger than one. In the latter case, pick a random edge, match the two endpoints of this edge, and delete those vertices and the edges incident on them. Then repeat the whole process on the remaining graph. The phase before the first random choice of edges made by the KS algorithm is called Phase 1, and the rest is called Phase 2 (where new degree-one vertices may arise). The running time of this heuristic is linear. This heuristic matches all but $O(n^{1/5})$ vertices of a random undirected graph [3]. One disadvantage of KS is that because of the degree dependencies of the vertices to the already matched vertices, an efficient parallelism is hard to achieve (a list of degree-one vertices needs to be maintained). That is probably why some inflicted forms (successful but without any known quality guarantee) of this heuristic were used in recent studies [4].

B. Scaling matrices to doubly stochastic form

An $n \times n$ matrix $A$ is said to have a support if there is a perfect matching in the associated bipartite graph. An $n \times n$ matrix $A$ is said to have total support if all its nonzero entries can be put into a perfect matching. Any nonnegative matrix $A$ with total support can be scaled with two (unique) positive diagonal matrices $D_R$ and $D_C$ such that $D_RAD_C$ is doubly stochastic (that is, the sum of entries in any row and in any column of $D_RAD_C$ is equal to one). If $A$ has a support but not a total support then $A$ can be scaled to a doubly stochastic matrix but not with two positive diagonal matrices (see [30] or more recent treatments [22], [23], [29]).

Here we review two algorithms for doubly-stochastic matrix scaling from the literature. The Sinkhorn-Knopp scaling algorithm [30] generates a sequence of matrices (whose limit is doubly stochastic) by normalizing the columns and the rows of the sequence of matrices alternately. That is, the initial matrix is normalized such that each column has sum one. Then, the resulting matrix is normalized so that each row has sum one and so on so forth. This algorithm has been recently analyzed where most of the known results are summarized [22].

Another scaling algorithm is proposed by Ruiz [29], parallelized for distributed [1] and shared memory [7] parallel systems, and its properties are investigated [23]. This algorithm also builds a sequence of matrices converging to a double
stochastic matrix. Instead of normalizing the rows and columns of the matrices alternately, this algorithm scales the rows and the columns of each matrix in the sequence. It has been shown that [23] this algorithm converges slower than the Sinkhorn-Knopp algorithm for unsymmetric matrices. For the symmetric matrices, there is no such a clear cut distinction.

We use a parallel implementation of the Sinkhorn-Knopp scaling method, shown in Algorithm 1, but other doubly stochastic scaling methods can also be used. In Algorithm 1, $A_+$ and $A_{i,j}$ are the sets of columns and row indices of the nonzeros at the $i$th row and $j$th column of $A$, respectively. Instead of the diagonal scaling matrices $D_r$ and $D_c$, we use two arrays $d_r$ and $d_c$ to store the (diagonal) entries of the scaling matrices. As is seen, the method runs until convergence, where we want to stop when both the row sums and column sums are sufficiently close to one. At each iteration, we first balance the columns and then the rows, at which point the row sums are one (modulo round-off errors), but the column sums are not. The stopping criteria for convergence is therefore to have the maximum difference between the column sums and one as small as possible. At the end, $d_r[i] \times a_{ij} \times d_c[j]$ gives the scaled entry. There are techniques to improve the parallel performance of Algorithm 1. For example, in case of skewness in degree distributions, one assign multiple threads to a single row with many nonzeros. However, we do not focus on this issues here.

Algorithm 1 SCALESK: Parallel Sinkhorn-Knopp scaling

Input: $A$: an $n \times n$ matrix with total support
Output: $d_r, d_c$: row/column scaling arrays
1: for $i = 1$ to $n$ in parallel do
2: $d_r[i] \leftarrow 1$
3: $d_c[i] \leftarrow 1$
4: while not converged do
5: for $j = 1$ to $n$ in parallel do
6: $c_{sum} \leftarrow \sum_{i \in A_{i,j}} d_r[i] \times a_{ij}$
7: $d_r[i] \leftarrow 1/c_{sum}$
8: for $i = 1$ to $n$ in parallel do
9: $r_{sum} \leftarrow \sum_{j \in A_{i,j}} a_{ij} \times d_c[j]$
10: $d_c[j] \leftarrow 1/r_{sum}$

III. TWO MATCHING HEURISTICS

We propose two simple matching heuristics for the maximum cardinality bipartite matching problem that are highly parallelizable and have guaranteed approximation ratios. The first heuristic does not require synchronization or conflict resolution. This heuristic and its approximation guarantee of around 0.632 are discussed in the following subsection. The second heuristic is designed to obtain larger matchings compared to those obtained by the first one. This heuristic employs KS on a judiciously chosen subgraph of the input graph. We show that for this subgraph, the KS heuristic is an exact maximum cardinality matching algorithm. Based on our experiments and theoretical evidence, we conjecture that the second heuristic obtains matchings of size around 0.866 of the maximum matching cardinality.

A. One-sided matching

The first matching heuristic we propose, ONE_SIDED-MATCH, scales the given adjacency matrix $A$ (each $a_{ij}$ is originally either 0 or 1) and uses the scaled entries to randomly choose a column as a match for each row. The pseudocode of the heuristic is shown in Algorithm 2.

Algorithm 2 ONE_SIDEDMATCH

Input: $A$: an $n \times n$, $(0,1)$-matrix with total support
Output: $c_{match}$: the rows matched to columns
1: $(d_r, d_c) \leftarrow \text{SCALESK}(A)$
2: for $j = 1$ to $n$ in parallel do
3: $c_{match}[j] \leftarrow \text{NIL}$
4: for $i = 1$ to $n$ in parallel do
5: Pick a random column $j \in A_{i, +}$ by using the probability density function
6: $c_{match}[j] \leftarrow i$

ONE_SIDEDMATCH first obtains the scaling vectors $d_r$ and $d_c$ corresponding to a doubly stochastic matrix $S$ (line 1). After initializing the $c_{match}$ array, for each row $i$ of $A$, the heuristic randomly chooses a column $j \in A_{i, +}$ based on the probabilities computed by using corresponding scaled entries of row $i$. It then matches $i$ and $j$. Clearly multiple rows can choose the same column and write to the same entry in $c_{match}$. In a parallel, shared-memory setting, one of the write operation survives, and the $c_{match}$ array defines a valid matching, i.e., $\{c_{match}[j], j : c_{match}[j] \neq \text{NIL}\}$. We now analyze its approximation guarantee in terms of the matching cardinality.

Theorem 1. Let $A$ be an $n \times n$, $(0,1)$-matrix with total support. Then, ONE_SIDEDMATCH obtains a matching of size at least $n(1 - 1/e) \approx 0.632n$.

Proof: To compute the matching cardinality, we will count the columns that are not picked by any row and subtract it from $n$. Since $\sum_{k \in A_{s,j}} s_{ik} = 1$ for each row $i$ of $S$, the probability that a column $j$ is not picked by any of the rows in $A_{s,j}$ is equal to $\prod_{i \in A_{s,j}} (1 - s_{ij})$. By applying the arithmetic-geometric mean inequality, we obtain

$$a_j \prod_{i \in A_{s,j}} (1 - s_{ij}) \leq \frac{1}{d_j} \sum_{i \in A_{s,j}} s_{ij},$$

where $d_j = |A_{s,j}|$ is the degree of column vertex $j$. Therefore,

$$\prod_{i \in A_{s,j}} (1 - s_{ij}) \leq \left(1 - \frac{1}{d_j} \sum_{i \in A_{s,j}} s_{ij}\right)^{d_j}.$$

Since $S$ is doubly stochastic, we have $\sum_{i \in A_{s,j}} s_{ij} = 1$ and

$$\prod_{i \in A_{s,j}} (1 - s_{ij}) \leq \left(1 - \frac{1}{d_j}\right)^{d_j}.$$
The function on the right hand side above is an increasing one, and has the limit
\[
\lim_{d_j \to \infty} \left( 1 - \frac{1}{d_j} \right)^{d_j} = \frac{1}{e},
\]
where \( e \) is the base of the natural logarithm. By the linearity of expectation, the expected number of unmatched columns is no larger than \( \frac{n}{2e} \). Hence, the cardinality of the matching is no smaller than \( n \left( 1 - 1/e \right) \).

In Algorithm 2, we split the rows among the threads with a parallel for construct. For each row \( i \), the corresponding thread chooses a random number \( r \) from a uniform distribution with range \( (0, \sum_{k \in A_i} s_{ik}) \). Then the smallest column index \( j \) for which \( \sum_{k \in A_i} s_{ik} \leq r \) is found and \( \text{match}[j] \) is set to \( i \). Since no synchronization or conflict detection is required, the heuristic promises significant speedups.

**B. Two-sided matching**

**ONESIDEDMATCH**'s approximation guarantee and suitable structure for parallel architectures make it a good cheap matching heuristic. The natural question that follows is whether a heuristic with a better guarantee while without being too complicated to parallelize exists. We asked: "what happens if we repeat the process for the other (column) side of the bipartite graph"? The question led us to the following algorithm. Let each row select a column, and let each column select a row. Take all these \( 2n \) choices to construct a bipartite graph \( G \) (a subgraph of the input) with \( 2n \) vertices and at most \( 2n \) edges (if \( i \) chooses \( j \) and \( j \) chooses \( i \), we have one edge), and seek a maximum cardinality matching in \( G \). Since the number of edges is at most \( 2n \), any exact matching algorithm on this graph would be fast—in particular the worst case running time would be \( O(n^{1.5}) \) [17]. Yet, we can do better and obtain a maximum cardinality matching in linear time by running the Karp-Sipser heuristic on \( G \), as we display in Algorithm 3.

The most interesting component of **TWOSIDEDMATCH** is the incorporation of the Karp-Sipser heuristic for two reasons. First, although it is only a heuristic, KS computes a maximum cardinality matching on the bipartite graph \( G \) constructed in Algorithm 3. Second, although KS has a sequential nature, we can obtain good speedups with a specialized, parallel implementation. In general, it is hard to parallelize (non-trivial) graph algorithms, and it is even harder when the overall cost is \( O(n) \) which is the case for KS on \( G \). We give a series of lemmas below which enables us to use KS as an exact algorithm with a good shared-memory parallel performance.

The first lemma describes the structure of \( G \) constructed at line 8 of **TWOSIDEDMATCH**.

**Lemma 1.** Each connected component of \( G \) constructed in Algorithm 3 contains at most one simple cycle.

**Proof:** A connected component \( M \subseteq G \) with \( n \) vertices can have at most \( n \) edges. Let \( T \) be a spanning tree of \( M \). Since \( T \) contains \( n - 1 \) edges, the remaining edge in \( M \) can create at most one cycle when added to \( T \).

**Algorithm 3 TWOSIDEDMATCH**

**Input:** \( A \): an \( n \times n \), (0,1)-matrix with total support

**Output:** \( \text{match}[:]: \) the mate of each vertex or NIL

1: \( (d_i, d_c) \leftarrow \text{SCALEKS}(A) \)
2: for \( i = 1 \) to \( n \) in parallel do
3: Pick a random column \( j \in A_{i*} \) by using the probability density function
4: \( p_i(k) = \frac{s_{ik}}{\sum_{l \in A_{i*}} s_{il}} \), for all \( k \in A_{i*} \)
5: \( r\text{choice}[i] \leftarrow j \)
6: for \( j = 1 \) to \( n \) in parallel do
7: \( c\text{choice}[j] \leftarrow i \)
8: Construct a bipartite graph \( G = (V_R \cup V_C, E) \) where
9: \( E = \{(i, r\text{choice}[i]) : i \in \{1, \ldots, n\}\} \cup \{(c\text{choice}[j], j) : j \in \{1, \ldots, n\}\} \)

10: \( \text{match} \leftarrow \text{KARPSIPSER}(G) \)

Lemma 1 explains why KS is an exact algorithm on \( G \). If a component does not contain a cycle, KS consumes all its vertices in Phase 1. Therefore, all of the matching decisions given by KS are optimal for this component. Assume a component contains a simple cycle. After Phase 1, the component is either consumed, or due to Lemma 1, it is reduced to a simple cycle. In the former case, the matching is a maximum cardinality one. In the latter case, an arbitrary edge of the cycle can be used to match a pair of vertices. This decision necessarily leads to a unique perfect matching in the remaining simple path. These two arguments can be repeated for all the connected components to see that the KS heuristic finds a maximum cardinality matching in \( G \).

Algorithm 4 describes our parallel KS implementation **KARPSIPSERMT**. The graph is represented using a single array choice, where \( \text{choice}[u] \) is the vertex randomly chosen by \( u \in V_R \cup V_C \). The choice array is a concatenation of the arrays rchoice and cchoice set in **TWOSIDEDMATCH**. Hence, an explicit graph construction for \( G \) (line 8 of Algorithm 3) is not required, and a transformation of the selected edges to a graph storage scheme is avoided. **KARPSIPSERMT** uses three atomic operations for synchronization. The first operation \_ADD(memory, value) atomically adds a value to a memory location. It is used to compute the vertex degrees in the initial graph (line 9). The second operation \_COMPANDSWAP(memory, value, replace) first checks whether the memory location has the value. If so, its content is replaced. The final content is returned. The third operation \_ADDANDFETCH(memory, value) atomically adds a given value to a memory location and the final content is returned. We will describe the use of these two operations later.

**KARPSIPSERMT** has two phases which correspond to the two phases of KS. The first phase of **KARPSIPSERMT** is
Algorithm 4 KARPSIPSERMT

Input: \( G = (V, \text{choice}) \): the chosen vertex for each \( u \in V \)

Output: \( \text{match}[\cdot] \): the match array for \( u \in V \)

1: for all \( u \in V \) in parallel do
2: \( \text{mark}[u] \leftarrow 1 \)
3: \( \text{deg}[u] \leftarrow 1 \)
4: \( \text{match}[u] \leftarrow \text{NIL} \)
5: for all \( u \in V \) in parallel do
6: \( v \leftarrow \text{choice}[u] \)
7: \( \text{mark}[v] \leftarrow 0 \)
8: if \( \text{choice}[v] \neq u \) then
9: \( \text{ADD}(\text{deg}[v], 1) \)
10: end if
11: end for
12: end for
13: while \( \text{curr} \neq \text{NIL} \) do
14: \( \text{nbr} \leftarrow \text{choice}[\text{curr}] \)
15: if \( \text{COMPANDSWAP}(\text{match}[^{\text{nbr}}], \text{NIL}, \text{curr}) = \text{curr} \) then
16: \( \text{match}[\text{curr}] \leftarrow \text{nbr} \)
17: \( \text{curr} \leftarrow \text{NIL} \)
18: next \( \leftarrow \text{choice}[\text{nbr}] \)
19: if \( \text{match}[\text{next}] = \text{NIL} \) then
20: \( \text{ADDANDFETCH}(\text{deg}[\text{next}], -1) = 1 \) then
21: \( \text{curr} \leftarrow \text{next} \)
22: else
23: \( \text{curr} \leftarrow \text{NIL} \)
24: end if
25: end if
26: for each column vertex \( u \in \text{parallel do} \) //Phase 2: the rest
27: \( \text{match}[u] \leftarrow v \)
28: \( \text{match}[v] \leftarrow u \)
29: end for

similar to that of KS in that optimal matching decisions are made about some degree-one vertices. The second phase of KARPSIPSERMT handles remaining vertices very efficiently, without bothering with their degrees. The following definitions are used to clarify the difference between an original KS implementation and KARPSIPSERMT.

Definition 1. Given a matching and the array choice, let \( u \) be an unmatched vertex and \( v = \text{choice}[u] \). Then \( u \) is called:
- **out-one**, if \( v \) is unmatched, and no unmatched vertex \( w \) with \( \text{choice}[w] = u \) exists.
- **in-one**, if \( v \) is matched, and only a single unmatched vertex \( w \) with \( \text{choice}[w] = u \) exists.

The first phase of KARPSIPSERMT (for loop of line 10) does not track and match all degree-one vertices. Instead, only the out-one vertices are taken into account. For each such vertex \( u \) that is already out-one before Phase 1, we have \( \text{mark}[u] = 1 \). KARPSIPSERMT visits these vertices (lines 10-11). Newly arising out-one vertices are consumed right away without maintaining a list. The second phase of KARPSIPSERMT (for loop of line 24) is much simpler than that of KS as the degrees of the vertices are not tracked/updated. We now discuss how these simplifications are possible while ensuring a maximum cardinality matching in \( G \).

**Observation 1.** An out-one or an in-one vertex is a degree-one vertex according to KS.

**Observation 2.** A degree-one vertex (of KS) is either an out-one or an in-one vertex, or it is one of the two vertices \( u \) and \( v \) in a 2-clique such that \( v = \text{choice}[u] \) and \( u = \text{choice}[v] \).

**Lemma 2.** If there exists an in-one vertex in \( G \) at any time during the execution of KARPSIPSERMT, an out-one vertex also exists.

**Proof:** Let \( u \) be an in-one vertex and let \( v \) be the unmatched vertex such that \( \text{choice}[v] = u \). Let \( \mathcal{P} \) be the longest vertex sequence \( w_1, w_2, \ldots, w_k, u \) such that all \( w_i \)s are unmatched, \( \text{choice}[w_i] = w_{i+1} \) for \( 1 \leq i < k \), and \( v = w_k \). If \( \mathcal{P} \) has a finite length, then \( w_1 \) is an out-one vertex and we are done. On the other hand, if \( \mathcal{P} \) has an infinite length it must contain a cycle. Furthermore, \( u \) must be in this cycle, since each \( w_i \)'s next vertex, which also needs to be in the cycle, is uniquely defined by choice. But \( u \) is an in-one vertex and \( \text{choice}[u] \) is already matched. Thus \( \mathcal{P} \) has a finite length, and an out-one vertex \( (w_1) \) always exists.

According to Observation 1, all the matching decisions given by KARPSIPSERMT in Phase 1 are optimal, since an out-one vertex is a degree-one vertex. Observation 2 implies that among all the degree-one vertices, KARPSIPSERMT ignores only the in-ones and 2-cliques. According to Lemma 2, an in-one vertex cannot exist without an out-one vertex, therefore they are handled in the same phase. The 2-cliques that survive Phase 1 are handled in KARPSIPSERMT's Phase 2, since they can be considered as cycles.

To analyze the second phase of KARPSIPSERMT, we will use the following lemma.

**Lemma 3.** Let \( G' = (V'_R \cup V'_C, E') \) be the graph induced by the remaining vertices after the first phase of KARPSIPSERMT. Then, the set

\[ \{ (u, \text{choice}[u]) : u \in V'_R, \text{choice}[u] \in V'_C \} \]

is a maximum cardinality matching in \( G' \).

**Proof:** Apart from 2-cliques, no out-one or in-one (that is no degree-one) vertex remains after Phase 1. A component of \( G' \) can be a trivial (a singleton vertex), a 2-clique, or a simple cycle, according to Lemma 1. Let \( \mathcal{P} \) be a non-trivial component. Since the original graph is bipartite, if \( \mathcal{P} \) is a cycle it has the edges \( \{ u, \text{choice}[u] \} \) and \( \{ \text{choice}[v], v \} \) for \( u \in V'_R \cap \mathcal{P} \) and \( v \in V'_C \cap \mathcal{P} \). The edge set \( \{ (u, \text{choice}[u]) : u \in V'_R \cap \mathcal{P}, \text{choice}[u] \in V'_C \} \) defines a maximum cardinality matching for \( \mathcal{P} \). The union of these edge sets matches all the vertices except those in the trivial components, hence it is a maximum matching in \( G' \).

In the light of Observations 1 and 2 and Lemmas 1–3, KARPSIPSERMT is an exact algorithm on the graphs created in Algorithm 3. The worst case (sequential) running time of our implementation of KS is linear.

KARPSIPSERMT tracks and consumes only the out-one vertices. This brings high flexibility while executing KARPSIPSERMT in multi-threaded environments. Consider the example in Figure 1. Here, after matching a pair of vertices and removing them from the graph, multiple degree-one vertices can be generated. The standard KS uses a list to store these new degree-one vertices. Such a list is necessary to obtain
larger matching, but the associated synchronizations while updating it in parallel will be an obstacle for parallel efficiency. The synchronization can be avoided up to some level if one sacrifices the approximation quality by not making all optimal decisions (as in [4]). We continue with the following lemma to take advantage of the special structure of the graphs in TwoSidedMatch for parallel efficiency in Phase 1.

Lemma 4. Consuming an out-one vertex creates at most one new out-one vertex.

Proof: Let $u$ be the out-one vertex that is selected by KarpSipserMT and let $v$ be choice$_{[u]}$. Since $u$ is a degree-one vertex, its removal will only affect $v$. On the other hand, although a number of in-one vertices may appear, $v$'s removal can only make the vertex $w = \text{choice}_v$ out-one. This happens iff $w$ is still unmatched and there is no other unmatched vertex $y$ such that choice$_v y = w$.

The second phase of KarpSipserMT is efficiently parallelized by using the idea in Lemma 3. That is, a maximum cardinality matching for the graph remaining after the first phase of KarpSipserMT can be obtained via a simple parallel for construct (see line 24 of Algorithm 4).

Quality of approximation: We do not have a proof on the approximation guarantee of TwoSidedMatch. However, we have the following claim.

Conjecture 1. Let $A$ be an $n \times n$ matrix with total support. Then, TwoSidedMatch obtains, asymptotically always surely, a matching of size 0.866n.

Here is some supporting evidence for the conjecture. Let the initial matrix $A$ be an $n \times n$ matrix of 1s; that is $a_{ij} = 1$ for all $1 \leq i, j \leq n$. Then, the doubly stochastic matrix $S$ is such that $s_{ij} = \frac{1}{n}$ for all $1 \leq i, j \leq n$. In this case, the graph $G$ created by Algorithm 3 is a random 1-out bipartite graph [31]. Referring to a study by Meir and Moon [25], Karpinski and Pittel [18] argue that the maximum cardinality of a matching in a random 1-out bipartite graph is $2(1 - \rho)n \approx 0.866n \rho \approx 0.567$ is the unique solution of the equation $xe^x = 1$. We also present some experimental results in the next section to support the claim.

The proof of Conjecture 1 will contribute to the known results about the Karp-Sipser heuristic (recall that it is known to leave out $\tilde{O}(n^{1/5})$ vertices) by showing a constant approximation ratio with some preprocessing. The existence of a total support does not seem to be necessary for the conjecture to hold (see the next subsection).

C. Further discussions

The Sinkhorn-Knopp scaling algorithm converges linearly (when $A$ has total support) where the rate is equivalent to the square of the second largest singular value of the resulting, doubly stochastic matrix [22]. Although convergence is not required, we can bound the theoretical running time of OneSidedMatch and TwoSidedMatch as linear (hiding the number of iterations to convergence behind the big-o notation). If the matrix does not have (total) support, less is known about the Sinkhorn-Knopp scaling algorithm, in which case, we are not able to bound the running time of the scaling step. However, the scaling algorithms should be run only a few iterations (see also below), in which case the practical running time of our heuristics become linear (in edges and vertices).

We have discussed the proposed matching heuristics while assuming that $A$ has total support. This can be relaxed in two ways to render the overall approach practical in any bipartite graph. First, we do not need to run the scaling algorithms until convergence. If $\sum_{i \in A_j} s_{ij} \geq \alpha$ instead of $\sum_{i \in A_j} s_{ij} = 1$ for all $j \in \{1, \ldots, n\}$, then $\lim_{j \to \infty} (1 - \frac{\alpha}{j})^j = e^{-\alpha}$. In other words, if we apply the scaling algorithms a few iterations, or until some relatively large error tolerance, we can still derive similar results. For example, if $\alpha = 0.92$, we will have a matching of size $n \left(1 - \frac{1}{0.92}\right) \approx 0.6015n$ (column sums that are larger than one give improved ratios; but there are columns whose sum is less than one, when

![Image](image.png)
the convergence is not achieved) for Algorithm 2. In our experiments, the number of iterations were always a few, where ONEsidedMATCH’s proven approximation guarantee and TWOsidedMATCH’s conjectured guarantee were always observed. The second relaxation is that we do not need total support; we do not even need support nor equal number of vertices in the two vertex classes. We note that most theoretical studies on randomized matching heuristics concentrates on graphs with perfect matching, as this is enough to present approximation guarantees [26, Section 2]. Since little is known about the scaling methods on such matrices, we do not dwell into the subject (scaling algorithms are not our focus), but we mention some facts and observations, and later on, present some experiments to demonstrate the practicality of the proposed ONEsidedMATCH and TWOsidedMATCH heuristics.

A sparse matrix (not necessarily square) can be permuted into a block upper triangular form using the canonical Dulmage-Mendelsohn (DM) decomposition [12]

\[
A = \begin{pmatrix}
H & * & * \\
O & S & * \\
O & O & V
\end{pmatrix},
S = \begin{pmatrix}
S_1 & * \\
O & S_2
\end{pmatrix}
\]

where, \(H\) (horizontal) has more columns than rows and has a matching covering all rows; \(S\) is square and has a perfect matching; and \(V\) (vertical) has more rows than columns and a matching covering all columns. The following facts about the matching; and

\[
A = \begin{pmatrix}
H & * & * \\
O & S & * \\
O & O & V
\end{pmatrix},
S = \begin{pmatrix}
S_1 & * \\
O & S_2
\end{pmatrix}
\]

In the sequel, a matrix whose corresponding bipartite graph

\[
A = \begin{pmatrix}
H & * & * \\
O & S & * \\
O & O & V
\end{pmatrix},
S = \begin{pmatrix}
S_1 & * \\
O & S_2
\end{pmatrix}
\]

A generic full-sprank matrix structure that is bad for KS.

\[
A = \begin{pmatrix}
H & * & * \\
O & S & * \\
O & O & V
\end{pmatrix},
S = \begin{pmatrix}
S_1 & * \\
O & S_2
\end{pmatrix}
\]

where, \(H\) (horizontal) has more columns than rows and has a matching covering all rows; \(S\) is square and has a perfect matching; and \(V\) (vertical) has more rows than columns and a matching covering all columns. The following facts about the DM decomposition are well known [27], [28]. Any of these three blocks can be void. If \(H\) is not connected, then it is block diagonal with horizontal blocks. If \(V\) is not connected, then it is block diagonal with vertical blocks. If \(S\) does not have total support, then it is in block upper triangular form, shown on the right, where \(S_1\) and \(S_2\) have the same structure recursively, until each block \(S_i\) is has total support. The entries in the blocks shown by “*” cannot be put into a maximum cardinality matching. When the presented scaling methods are applied to a matrix, the entries in “*” blocks will tend to zero (the case of \(S\) is well documented [30]). Furthermore, the row sums of the blocks of \(H\) will be a multiple of the column sums in the same block; a similar statement holds for \(V\); finally \(S\) will be doubly stochastic. That is, the scaling algorithms applied to bipartite graphs without perfect matchings will zero out the entries in the irrelevant parts and identify the entries that can be put into a maximum cardinality matching.

\[
A = \begin{pmatrix}
H & * & * \\
O & S & * \\
O & O & V
\end{pmatrix},
S = \begin{pmatrix}
S_1 & * \\
O & S_2
\end{pmatrix}
\]

IV. EXPERIMENTS

The experiments were carried out on a machine equipped with two Intel Sandybridge-EP CPUs clocked at 2.00Ghz and 256GB of memory split across the two NUMA domains. Each CPU has eight-cores (16 cores in total) and HyperThreading is enabled. Each core has its own 32kB L1 cache and 256kB L2 cache. The 8 cores on a CPU share a 20MB L3 cache. The machine runs 64-bit Debian with Linux 2.6.39-19-amd64. All the codes are compiled with gcc 4.4.5 with the -O2 optimization flag. All algorithms are implemented using C and OpenMP parallelism. The (dynamic, 512) OpenMP scheduling policy is employed while running all

the algorithms except KARP-SIPSER-MT for which we used (guided). Parallel runs are performed with 2, 4, 8, 16 threads. For atomic operations, gcc’s built-in functions are used.

A. Experimental verification of theoretical results

1) Matching quality: We investigate the matching quality of the proposed heuristics on all square, fully indecomposable matrices from the UFL Sparse Matrix Collection [8] having at least 1000 non-empty rows/columns and at most 20000000 nonzeros (some of the matrices are also in [5]). For all but 37 of these matrices (there were 743 of them), the quality guarantees 0.632 and 0.866 were surpassed with 10 iterations of the scaling method. Making 10 more scaling iterations smoothed out those 37 problematic instances.

2) Comparisons with KARP-SIPSER: Next, we analyze the performance of the proposed heuristics with respect to KS on a matrix class which we designed as a bad case for KS. Let \(A\) be an \(n \times n\) matrix, \(R_1 (C_1)\) be the set of \(A\)’s first \(n/2\) rows (columns), and \(R_2 (C_2)\) be the set of \(A\)’s last \(n/2\) rows (columns). As Figure 2 shows, \(A\) has a full \(R_1 \times C_1\) block and an empty \(R_2 \times C_2\) block. The last \(k \ll n\) rows and columns of \(R_1\) and \(C_1\), respectively, are full. Each of the blocks \(R_3 \times C_2\) and \(R_2 \times C_1\) has a nonzero diagonal.

Those diagonals form a perfect matching when combined. In the sequel, a matrix whose corresponding bipartite graph has a perfect matching will be called full-sprank, and sprank-deficient otherwise.

When \(k \leq 1\), the KS heuristic consumes the whole graph during Phase 1 and finds a maximum cardinality matching. When \(k > 1\), Phase 1 immediately ends, since there is no degree-one vertex. In Phase 2, the first edge (nonzero) consumed by KS is selected from a uniform distribution over the nonzeros whose corresponding rows and columns are still unmatched. Since the block \(R_1 \times C_1\) is full, it is more likely that the nonzero will be chosen from this block. Thus, a row in \(R_1\) will be matched with a column in \(C_1\), which is a bad decision since the block \(R_2 \times C_2\) is completely empty. Hence, we expect a decrease on the performance of KS as \(k\) increases.

On the other hand the probability that \(T\) chooses an edge from that block goes to zero, as those entries were surpassed with 10 iterations of the scaling method. Making 10 more scaling iterations smoothed out those 37 problematic instances.

The results of the experiments are in Table I. The first column shows the \(k\) value. Then the matching quality obtained by KS, and by TWOsidedMATCH with different number of scaling iterations (0, 1, 5, 10), as well as the scaling error are given. The scaling error is the maximum difference between 1
and each row/column sum of the scaled matrix (for 0 iterations it is equal to \( n - 1 \) for all cases). The quality of a matching is computed by dividing its cardinality to the maximum one, which is \( n = 3200 \) for these experiments. To obtain the values in each cell of the table, we run the programs 10 times and give the minimum quality (as we are investigating the worst-case behavior). The highest variance for KS and TwoSidedMatch were (up to four significant digits) 0.0041 and 0.0001, respectively. As expected, when \( k \) increases, the KS heuristic performs worse, and the matching quality drops to 0.67 for \( k = 32 \). TwoSidedMatch’s performance increases with the number of scaling iterations. As the experiment shows, only 5 scaling iterations are sufficient to make the proposed two-sided matching heuristic significantly better than KS. However, this number is not enough to reach 0.866 for the matrix with \( k = 32 \). On this matrix, with 10 iterations, only 2% of the rows/columns remain unmatched.

3) Matching quality on bipartite graphs without perfect matchings: We analyze the proposed heuristics on a class of random sprank-deficient square (\( n = 100000 \)) and rectangular (\( m = 100000 \) and \( n = 120000 \)) matrices with a uniform nonzero distribution (two more sprank-deficient matrices are used in the scalability tests as well). These matrices are generated by MATLAB’s sprand command (generating Erdős-Rényi random matrices [14]). The total nonzeros is set to be around \( d \times n \) for \( d \in \{2, 3, 4, 5\} \). Table II presents the results of this experiment with square matrices (rectangular case is not shown). As in the previous experiments, the matching qualities in the table is the minimum of 10 executions for the corresponding instances. As Table II shows, when the deficiency is high (correlated with small \( d \)), it is easier for our algorithms to approximate the maximum cardinality. However, when \( d \) gets larger, the algorithms require more scaling iterations. Even in this case, 5 iterations are sufficient to achieve the guaranteed qualities. In the rectangular case, the minimum quality achieved by OneSidedMatch and TwoSidedMatch were 0.753 and 0.930, respectively, with 5 scaling iterations.

### B. OneSidedMatch and TwoSidedMatch in parallel

To analyze the scalability of the proposed heuristics in parallel, we use 12 large bipartite graphs corresponding to real-life matrices from UFL collection arising in different application domains. The names hugebubbles and channel refer to the matrices hugebubbles-00020 and channel-500x100x100-b050, respectively. The properties of the bipartite graphs are given in Table III along with the sequential running times (the running time of OneSidedMatch includes that of ScalesK, and TwoSidedMatch includes those of ScalesK and KarpsisperMT). All the executions in this experiment are repeated 20 times and the first five are ignored. The times are computed by using the geometric mean of the remaining 15 executions for each instance. No significant variances were observed among the remaining individual running times. The speedup values are computed with respect to the execution with a single-thread.

Figures 3(a) and 3(b) show the individual speedup values for ScalesK and the proposed matching heuristic OneSidedMatch, respectively. When executed with 16 threads, ScalesK obtains a speedup value around 8 or more for all matrices. The maximum speedup of 10.6 is obtained for hugebubbles. The scalability of OneSidedMatch is better. For 10 matrices, a speedup value around 10 or more is obtained. The maximum speedup of 11.4 is obtained for the matrix europe_osm with 16 threads.

The structure of a matrix can affect the scalability. Both for ScalesK and OneSidedMatch, the minimum speedups (7.7 and 8.4, respectively) are obtained on torsol. As Table III shows, torsol and audikw_1 are the two smallest matrices with less than \( 10^6 \) rows and columns. As Figure 3(b) shows, OneSidedMatch obtains its worst speedups on these matrices. This is not a coincidence. When the variance of the number of nonzeros per row is high, the effects of load imbalance can be significant. For torsol and audikw_1, the variances (computed in Matlab) are 170056 and 1802, respectively. Among the 12 matrices, the next largest variance is 42 (kkt_power). Although we conducted a set of preliminary experiments on OpenMP scheduling policies, we did not fine tune it to have the best one. A different policy may work better especially for these matrices with a high variance on the number of nonzeros per row.

We repeated the scalability experiment for KarpsisperMT and TwoSidedMatch on the same matrix set. The results can be seen in Figures 4(a) and 4(b), respectively. On average (geometric mean), KarpsisperMT obtains a speedup of 11.1 with 16 threads. The maximum speedup of 12.6 is obtained on the matrix channel. These results show that the

---

### TABLE I

<table>
<thead>
<tr>
<th>( k )</th>
<th>Karpsisper</th>
<th>Scaled</th>
<th>number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.782</td>
<td>0.522</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>0.704</td>
<td>0.489</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>0.707</td>
<td>0.466</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>0.685</td>
<td>0.448</td>
<td>10</td>
</tr>
<tr>
<td>32</td>
<td>0.670</td>
<td>0.447</td>
<td>10</td>
</tr>
</tbody>
</table>

### TABLE II

<table>
<thead>
<tr>
<th>( d )</th>
<th>( \text{iter} )</th>
<th>( \text{spank} )</th>
<th>( \text{OneSidedMatch} )</th>
<th>( \text{TwoSidedMatch} )</th>
<th>( \text{iter} )</th>
<th>( \text{spank} )</th>
<th>( \text{OneSidedMatch} )</th>
<th>( \text{TwoSidedMatch} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>78.225</td>
<td>0.70</td>
<td>0.912</td>
<td>4</td>
<td>0</td>
<td>91.787</td>
<td>0.644</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>78.225</td>
<td>0.797</td>
<td>0.917</td>
<td>4</td>
<td>1</td>
<td>91.787</td>
<td>0.673</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>78.225</td>
<td>0.850</td>
<td>0.939</td>
<td>4</td>
<td>5</td>
<td>91.787</td>
<td>0.719</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>78.225</td>
<td>0.879</td>
<td>0.954</td>
<td>4</td>
<td>10</td>
<td>91.787</td>
<td>0.740</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>92.786</td>
<td>0.673</td>
<td>0.851</td>
<td>5</td>
<td>0</td>
<td>99.223</td>
<td>0.655</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>92.786</td>
<td>0.703</td>
<td>0.857</td>
<td>5</td>
<td>1</td>
<td>99.223</td>
<td>0.662</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>92.786</td>
<td>0.756</td>
<td>0.884</td>
<td>5</td>
<td>5</td>
<td>99.223</td>
<td>0.701</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>92.786</td>
<td>0.784</td>
<td>0.902</td>
<td>5</td>
<td>10</td>
<td>99.223</td>
<td>0.716</td>
</tr>
</tbody>
</table>
TABLE III
SPRANK: THE MAXIMUM CARDINALITY OF A MATCHING; SCALING ERROR: THE MAXIMUM DIFFERENCE BETWEEN ONE AND COLUMN SUMS; SEQUENTIAL EXECUTION TIMES OF SCALESK (ONE ITERATION), ONESIDEDMATCH, KARPSIPSERMT, AND TWOSIDEDMATCH.

<table>
<thead>
<tr>
<th>Name</th>
<th>n</th>
<th># of edges</th>
<th>Avg. deg.</th>
<th>sprank/n</th>
<th>Execution times with single thread (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>ONE SIDED MATCH SK</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With 16 threads, TWOSIDEDMATCH obtains a speedup value of 10.6 on the average. Compared to the average of ONESIDEDMATCH (10.1), it is slightly better. However, in a sequential setting, TWOSIDEDMATCH is 2.6 times slower. To further compare these two heuristics, we present Figures 5(a) and 5(b) that show the qualities on our test matrices. In the figures, the first columns represent the case that the neighbors are picked from a uniform distribution over the adjacency lists, i.e., the case with no scaling, hence no guarantee. The quality guarantees are achieved with only 5 iterations for almost all the cases except TWOSIDEDMATCH on nlpkt240 which required 15 scaling iterations. Even with a single iteration, the quality of TWOSIDEDMATCH is more than 0.86 for all matrices. Only for two among them, the quality is between 0.863 and 0.866. The results are similar for ONESIDEDMATCH. However, even with 10 scaling iterations, ONESIDEDMATCH cannot achieve a quality of 0.80 on any of the matrices. We conclude that ONESIDEDMATCH is faster, TWOSIDEDMATCH has a better quality guarantee, and both demonstrate good speedups.

V. CONCLUSION

We proposed two heuristics for the bipartite maximum cardinality matching problem. The first one, ONESIDEDMATCH, is shown to have an approximation guarantee no smaller than 1 − 1/e ≈ 0.632. The second heuristic, TWOSIDEDMATCH, is conjectured to have an approximation guarantee no smaller than 0.866. Both algorithms use well-known methods to scale the sparse matrix associated with the given bipartite graph to a doubly stochastic form whose entries are used as the probability density functions to randomly select a subset of the edges of the input graph. ONESIDEDMATCH selects exactly n edges to construct a subgraph in which a matching of the guaranteed cardinality is identified with virtually no overhead, both in sequential and parallel execution. TWOSIDEDMATCH selects around 2n edges and then runs the Karp-Sipser (KS) heuristic as an exact algorithm on the selected subgraph to obtain a matching of conjectured cardinality. The subgraphs are analyzed to develop a specialized KS algorithm for efficient parallelization. All theoretical investigations are first performed assuming bipartite graphs with perfect matchings. Then, theoretical arguments and experimental evidence are provided to extend the results to cover other cases and validate the applicability and practicality of the proposed heuristics in general settings. Parallel performance is analyzed on a shared
memory parallel computer with up to 16 threads and speedups beyond 10 fold are demonstrated.

We are investigating variants of the proposed heuristics for finding approximate matchings in undirected graphs. The algorithms and results extend naturally, but more work needs to be done for theoretical explanations.

ACKNOWLEDGEMENTS

Fanny Dufossé is supported by the foundation STAE, Toulouse, France.

REFERENCES


