SIAM J. SCI. COMPUT.
Vol. 38, No. 5, pp. S593–S619

# EFFICIENT APPROXIMATION ALGORITHMS FOR WEIGHTED *b*-MATCHING[*]

ARIF KHAN[†], ALEX POTHEN[†], MD. MOSTOFA ALI PATWARY[‡], NADATHUR RAJAGOPALAN SATISH[‡], NARAYANAN SUNDARAM[‡], FREDRIK MANNE[§], MAHANTESH HALAPPANAVAR[¶], AND PRADEEP DUBEY[‡]

**Abstract.** We describe a half-approximation algorithm, $b$-SUITOR, for computing a $b$-MATCHING of maximum weight in a graph with weights on the edges. $b$-MATCHING is a generalization of the well-known MATCHING problem in graphs, where the objective is to choose a subset of $M$ edges in the graph such that at most a specified number $b(v)$ of edges in $M$ are incident on each vertex $v$. Subject to this restriction we maximize the sum of the weights of the edges in $M$. We prove that the $b$-SUITOR algorithm computes the same $b$-MATCHING as the one obtained by the GREEDY algorithm for the problem. We implement the algorithm on serial and shared-memory parallel processors and compare its performance against a collection of approximation algorithms that have been proposed earlier. Our results show that the $b$-SUITOR algorithm outperforms the GREEDY and locally dominant edge algorithms by one to two orders of magnitude on a serial processor. The $b$-SUITOR algorithm has a high degree of concurrency, and it scales well up to 240 threads on a shared-memory multiprocessor. The $b$-SUITOR algorithm outperforms the locally dominant edge algorithm by a factor of 14 on 16 cores of an Intel Xeon multiprocessor.

**Key words.** $b$-matching, approximation algorithms, parallel algorithms

**AMS subject classifications.** 68R10, 68W10, 05C85

**DOI.** 10.1137/15M1026304

## 1. Introduction.

We describe a half-approximation algorithm, $b$-SUITOR, for computing a $b$-MATCHING of maximum weight in a graph, implement it on serial and shared-memory parallel processors, and compare its performance against approximation algorithms that have been proposed earlier. $b$-MATCHING is a generalization of the well-known MATCHING problem in graphs, where the objective is to choose a subset $M$ of edges in the graph such that at most $b(v)$ edges in $M$ are incident on each vertex $v$, and subject to this restriction we maximize the sum of the weights of the edges in $M$. (Here $b(v)$ is a nonnegative integer.)

There has been a flurry of activity on approximation algorithms for the weighted MATCHING problem in recent years, since (exact) algorithms for computing optimal matchings, while requiring polynomial time for many problems, still are too expen-

[*]Received by the editors June 16, 2015; accepted for publication (in revised form) March 11, 2016; published electronically October 27, 2016.
http://www.siam.org/journals/sisc/38-5/M102630.html

[†]Department of Computer Science, Purdue University, West Lafayette, IN 47907-2107 (khan58@purdue.edu, apothen@purdue.edu). The research of these authors was supported by the U.S. Department of Energy through grant DE-SC0010205 and the National Science Foundation through grant CCF-1552323.

[‡]Intel Labs, Santa Clara, CA 95054 (mostofa.ali.patwary@intel.com, nadathur.rajagopalan.satish@intel.com, narayanan.sundaram@intel.com, pradeep.dubey@intel.com).

[§]Department of Informatics, University of Bergen, N-5020, Bergen, Norway (Fredrik.Manne@ii.uib.no).

[¶]Pacific Northwest National Laboratory, Richland, WA 99352 (hala@pnnl.gov). This author's work was supported in part by the Applied Mathematics Program of the Office of Advance Scientific Computing Research within the Office of Science of the U.S. Department of Energy at the Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under contract DE-AC06-76RL01830.

sive for massive graphs with close to a billion edges. These approximation algorithms have nearly linear time complexity, are simple to implement, and have high concurrency, so that effective serial and shared-memory parallel algorithms are now available. Experimentally they compute nearly optimal matchings as well in terms of weight.

While a few earlier papers have described exact algorithms for $b$-Matchings, these again have high computational complexity and are difficult to implement efficiently. We do not know of an effective program that is currently available in the public domain. There has been much less work on approximation algorithms and implementations for $b$-Matching problems. $b$-Matchings have been applied to a number of problems from different domains: these include finite element mesh refinement [30], median location problems [40], spectral data clustering [19], semisupervised learning [20], etc.

Recently, Choromanski, Jebara, and Tang [3] used $b$-Matching to solve a data privacy problem called Adaptive Anonymity. Given a database of $n$ instances each with $d$ attributes, the Adaptive Anonymity problem asks for the fewest elements to mask so that each instance $i$ will be confused with $k_i - 1$ other instances to ensure privacy. The problem is NP-hard, and a heuristic solution is obtained by grouping each instance with the specified number (or more) of other instances that are most similar to it in the attributes. The grouping step is done by creating a complete graph with the instances as the vertices and the similarity score between two instances as the edge weight between the two vertices. Then a $b$-Matching of maximum weight is computed, where $b(i) = k_i - 1$. The authors of [3] used a perfect $b$-Matching for the grouping step in the context of an iterative algorithm for the Adaptive Anonymity problem, but this is not guaranteed to converge because a perfect $b$-Matching might not exist for a specified set of $b(i)$ values. In recent work, Choromanski et al. [4] have used the $b$-Suitor algorithm (described here) to compute an approximate $b$-Matching, provide an approximation bound of $2\beta$ for the anonymity problem if there are no privacy violations, and solve the problem an order of magnitude faster (Here $\beta$ is the maximum desired level of privacy.). Moreover, a specific variant of $b$-Suitor, the *delayed partial* scheme, reduces the space complexity of Adaptive Anonymity from quadratic to linear in the number of instances. An Adaptive Anonymity problem with a million instances and 500 features has been solved by this approach on a Xeon processor with 20 cores in about 10 hours. This approach has increased the size of Adaptive Anonymity problems solved by three orders of magnitude from earlier algorithms. These authors have also formulated the Adaptive Anonymity problem in terms of the related concept of $b$-Edge covers and obtained a $3\beta/2$-approximation algorithm for the problem.

Our contributions in this paper are as follows:

1. We propose the $b$-Suitor algorithm, a new half-approximation algorithm for $b$-Matching, based on the recent Suitor algorithm for Matching. This latter algorithm is considered to be among the best performing algorithms based on matching weight and run time.
2. We prove that the $b$-Suitor algorithm computes the same $b$-Matching as the one obtained by the well-known Greedy algorithm for the problem.
3. We have implemented the $b$-Suitor algorithm on serial processors and shared-memory multiprocessors and explored six variants of the algorithm to improve its performance.
4. We evaluate the performance of these variants on a collection of test problems and compare the weight and run times with seven other approximation and heuristic algorithms for the problem.

5. We show that the $b$-Suitor algorithm is highly concurrent and show that it scales well up to 240 threads on shared-memory multiprocessors.

This paper is organized as follows. Section 2 describes concepts and definitions needed to discuss Matchings and $b$-Matchings. We then discuss earlier work on exact and approximation algorithms for these problems. In section 3 we describe the serial $b$-Suitor algorithm, prove it correct, and then develop a parallel algorithm. We describe variants of the algorithm that could improve its performance. Section 4 reports on the processor architectures, test problems, the weight of the approximate matchings, and factors that influence performance such as the number of edges traversed, cache misses, and finally the run times. We compare the run times of the $b$-Suitor algorithm with other approximation algorithms for both serial and parallel computations. We discuss how the run time performance is improved by optimizations that exploit architectural features of the multiprocessors. We also investigate scalability of the $b$-Suitor algorithm on a processor with 16 threads and a coprocessor with 240 threads. Section 6 includes our concluding remarks.

## 2. Background.

We consider an undirected, simple graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. We denote $n \equiv |V|$, and $m \equiv |E|$. Given a function $b$ that maps each vertex to a nonnegative integer, a $b$-Matching is a set of edges $M$ such that *at most* $b(v)$ edges in $M$ are incident on each vertex $v \in V$. (This corresponds to the concept of a simple $b$-Matching in Schrijver [39].) An edge in $M$ is matched, and an edge not in $M$ is unmatched. Similarly, an endpoint of an edge in $M$ is a matched vertex, and other vertices are unmatched. We can maximize several metrics, e.g., the cardinality of a $b$-Matching. If $M$ has exactly $b(v)$ edges incident on each vertex $v$, then the $b$-Matching is perfect. An important special case is when the $b(v)$ values are the same for every vertex, say, equal to $b$. In this case, a perfect $b$-Matching $M$ is also called a $b$-factor. For future use, we define $\beta = \max_{v \in V} b(v)$ and $B = \sum_{v \in V} b(v)$. We also denote by $\delta(v)$ the degree of a vertex $v$ and by $\Delta$ the maximum degree of a vertex in a graph $G$.

Now consider the case when there are nonnegative weights on the edges, given by a function $W : E \mapsto R_{\geq 0}$. The weight of a $b$-Matching is the sum of the weights of the matched edges. We can maximize the weight of a $b$-Matching, and it is not necessarily a $b$-Matching of maximum cardinality.

The commonly studied case with $b = 1$ corresponds to a Matching in a graph, where the matched edges are now independent, i.e., the endpoints of matched edges are vertex-disjoint from each other. We will use results from Matching theory and algorithms to develop results for the $b$-Matching case.

An exact algorithm for a maximum weight $b$-Matching was first devised by Edmonds [10] and was implemented as a bidirected flow problem in the code Blossom I. Pulleyblank [37] later gave a pseudopolynomial time algorithm with complexity $O(mnB)$. The $b$-Matching problem can be reduced to 1-matching [12, 26] but the reduction increases the problem size and is impractical as a computational approach for large graphs. Anstee [1] proposed a three-stage algorithm where the $b$-Matching problem is solved by transforming it into a Hitchcock transportation problem, rounding the solution to integer values, and finally invoking Pulleyblank's algorithm. Derigs and Metz [7] and Miller and Pekny [29] improved the Anstee algorithm further. Padberg and Rao [33] developed another algorithm using the branch and cut approach and Grötschel and Holland [14] solved the problem using the cutting plane technique. A survey of exact algorithms for $b$-Matchings was provided by [30]. More recently, Huang and Jebara [17] proposed an exact $b$-Matching algorithm based on belief propagation. The algorithm assumes that the solution is unique, and otherwise it does not guarantee convergence.

**2.1. Approximation algorithms for** MATCHING**.** The approximation algorithms that we develop for $b$-MATCHING have their counterparts for MATCHING, so we review the algorithms for the latter problem now. After this subsection, we will review the work that has been done for approximation algorithms for $b$-MATCHING. The GREEDY algorithm iteratively matches edges in nonincreasing order of weights, deleting edges incident on the endpoints of a matched edge, and this is a half-approximation algorithm for edge-weighted matching [2]. Preis [36] designed a half-approximation algorithm that repeatedly finds and matches a *locally dominant* (LD) edge, and showed that this can be implemented in time linear in the number of edges. (An edge is LD if it is at least as heavy as all other edges incident on its endpoints.) Drake and Hougardy [8] obtained a linear time half-approximation algorithm that grows paths in the graph consisting of heavy edges incident on each vertex, decomposes each path into two matchings, and chooses the heavier matching to include in the approximation. This is called the path-growing algorithm (PGA). Dynamic programming can be employed on each path to obtain a heaviest matching from it, and this practically improves the weight of the computed matching. This variant is called the PGA$'$ algorithm. Maue and Sanders [27] describe a global paths algorithm (GPA) that sorts the edges in nonincreasing order of weights, and then grows paths from the edges in this order. All these algorithms typically find matchings of weight greater than 95% of the optimal matching, and the GPA algorithm usually finds the heaviest weight matching in practice. Due to the high quality and speed of the half-approximation algorithms, algorithms with better approximation ratios are not usually competitive for weighted matching.

Considering parallel algorithms, several variants of the LD matching have been proposed. The algorithm of Fagginger Auer and Bisseling [11] targets parallel efficiency on GPU architectures by relaxing the guarantee of half approximation. Vertices are randomly colored blue or red, following which blue colored vertices propose to red colored vertices. In the next step, red colored vertices respond to proposals, if any, and pick the best proposal. Edges along matching proposals between blue and red colored vertices get added to the matched set, and the corresponding vertices are marked black. Vertices without potential mates get marked as dead. The algorithm iterates until there are no more eligible (blue) vertices to match. In recent work, Naim et al. provide GPU implementations of the SUITOR algorithm that guarantees half approximation and exploits the hierarchical parallelism of modern Nvidia GPU architectures [32].

Riedy et al. [38] have implemented an LD algorithm for community detection. It iterates through unmatched vertices to identify LD edges, adding them to the matched set, and iterating until a maximal matching has been computed. Since this is a variant of the Manne and Bisseling algorithm, it obtains half approximation for the weight of the matched edges. This implementation targets the massively multithreaded Cray XMT platform that supports fine-grained synchronization as well as multithreaded computations using OpenMP. Halappanavar et al. [15] have designed a novel dataflow algorithm to implement the LD edge algorithm that exploits the hardware features of the Cray XMT. On measures of the run times on serial and parallel processors for the 1-Matching problem, the SUITOR algorithm has been demonstrated to perform better than other approximation algorithms [24]. In this work, we consider $b$-MATCHINGS rather than MATCHINGS and, in doing so, extend the ideas from the SUITOR algorithm with techniques such as partial sorting, delayed updates, and the order in which proposals are extended.

We discuss approximation algorithms with better approximation ratios than half. Randomized algorithms that have an approximation ratio of $2/3-\epsilon$ for small positive values of $\epsilon$ have been designed. These algorithms have been found to be an order of magnitude slower than the half-approximation algorithms [27]. A $(1-\epsilon)$ approximation algorithm based on the scaling approach has been designed recently by Duan and Pettie [9]. This algorithm is based on the scaling approach for weighted matching, runs in $O(m\log n^3\epsilon^{-2})$ time, and has not been implemented in practice. This paper and Hougardy [16] provide comprehensive surveys of the work on approximate matching algorithms.

We now describe a new half-approximation algorithm for matching called the Suitor algorithm that was recently proposed by Manne and Halappanavar [24]. This algorithm is currently the best performing algorithm in terms of the two metrics of the run time and weight of the matching. The $b$-Suitor algorithm proposed in this paper is derived from this algorithm. The Suitor algorithm may be considered as an improvement over the LD algorithm.

In the LD algorithm, each vertex sets a pointer to the neighbor it wishes to match with. Vertices consider neighbors to match with in decreasing order to weights. When two vertices point to each other, the edge is locally dominating and is added to the matching. Edges adjacent to LD edges are deleted, and the algorithm iteratively searches for LD edges, adds them to the matching, and updates the graph.

In the Suitor algorithm, each vertex $u$ proposes to match with its heaviest neighbor $v$ that currently does not have a better offer than the weight of the edge $(u,v)$. When two vertices propose to each other, they are matched, although they could get unmatched in a future step. The algorithm keeps track of the best current offer (the weight of the edge proposed to be matched) of each vertex. A vertex $u$ extends a proposal to a neighbor $v$ only if the weight of the edge $(u,v)$ is heavier than the current best offer that $v$ has. This reduces the number of candidate edges that need to be searched for matching relative to the LD algorithm. If vertex $u$ finds that a neighbor $v$ that it could propose to has a current offer from another vertex $x$ that is less than the weight of the edge $(u,v)$, then it annuls the proposal from the vertex $x$, proposes to $v$, and updates the current best offer of the vertex $v$ to the weight of $(u,v)$. Now the vertex $x$ needs to propose to its next heaviest neighbor $y$ that already does not have an offer better than the weight of the edge $(x,y)$. It can be shown that the Suitor algorithm computes the same matching as the one obtained by the Greedy and the LD matchings, provided ties are broken consistently.

Manne and Halappanavar have described shared-memory parallel implementations of the Suitor algorithm. Earlier, Manne and Bisseling [23] had developed a distributed-memory parallel algorithm based on the LD edge idea, and this was followed by Halappanavar, et al. [15], who developed shared-memory parallel algorithms for several machines.

A heuristic algorithm called Heavy Edge Matching (HEM), which matches the heaviest edge incident on each vertex in an arbitrary order of vertices, has been used to coarsen graphs in the multilevel graph partitioning algorithm [21]. This algorithm provides no guarantees on the approximation ratio of the weighted matching that it computes, but it is faster relative to the approximation algorithms considered here [24]. We will discuss it further in section 4 on results.

**2.2. Approximation algorithms for $b$-Matching.** Relatively little work has been done on approximate $b$-Matching. Mestre [28] showed that a $b$-Matching is a relaxation of a matroid called a $k$-extendible system with $k = 2$ and hence that the Greedy algorithm gives a $1/k = 1/2$-approximation for a maximum weighted $b$-Matching. He generalized the PGA of Drake and Hougardy [8] to obtain an $O(\beta m)$ time half-approximation algorithm. He also generalized a randomized algorithm for Matching to obtain a $(2/3-\epsilon)$-approximation algorithm with expected running time $O(\beta m \log \frac{1}{\epsilon})$ [28]. We will compare the performance of the serial $b$-Suitor algorithm with the PGA and PGA$'$ algorithms later in this paper. Since the PGA algorithm is inherently sequential, it is not a good candidate for parallelization. Morales, Gionis, and Sozio et al. [6] have adapted the Greedy algorithm and an integer linear program based algorithm to the MapReduce environment to compute $b$-Matchings in bipartite graphs. There have been several attempts at developing fast $b$-Matching algorithms using linear programming [22, 25], but where experimental studies have been performed, these methods are orders of magnitude slower than the approximation algorithms considered in this paper. Georgiadis and Papatriantafilou [13] have developed a distributed algorithm based on adding locally dominating edges to the $b$-Matching. We also implement the LD algorithm and compare it with the $b$-Suitor algorithm in section 4.

## 3. New $b$-Matching algorithm.

We describe here a new parallel approximation algorithm for maximum edge weighted $b$-Matching called $b$-Suitor.

**3.1. Sequential $b$-Suitor algorithm.** For each vertex $u$, we maintain a priority queue $S(u)$ that contains at most $b(u)$ elements from its adjacency list $N(u)$. The intent of this priority queue is to maintain a list of neighbors of $u$ that have proposed to $u$ and hence are *suitors* of $u$. The priority queue enables us to update the lowest weight of a suitor of $u$, in $\log b(u)$ time. If $u$ has fewer than $b(u)$ suitors, then we define this lowest weight to be zero. The operation $S(u).insert(v)$ adds the vertex $v$ to the priority queue of $u$ with the weight $W(u, v)$. If $S(u)$ has $b(u)$ vertices, then the vertex with the lowest weight in the priority queue is discarded on insertion of $v$. This lowest matched vertex is stored in $S(u).last$; if the priority queue contained fewer than $b(u)$ vertices, then a value of $NULL$ is returned for $S(u).last$.

In what follows, we will need to break ties consistently when the weights of two vertices are equal. Without loss of generality, we will say that $W(u) > W(v)$ if the weights are equal but vertex $u$ is numbered lower than $v$.

It is also conceptually helpful to consider an array $T(u)$ which contains the vertices that $u$ has proposed to. We could consider these as speculative matches. Again, there are at most $b(u)$ neighbors of $u$ in the set $T(u)$, and so this is a subset of $N(u)$. The operation $T(u).insert(v)$ inserts a vertex $v$ into the array $T(u)$, and $T(u).remove(v)$ removes the vertex $v$ from $T(u)$. Throughout the algorithm, we maintain the property that $v \in S(u)$ if and only if $u \in T(v)$. When the algorithm terminates, we satisfy the property that $v \in S(u)$ if and only if $u \in S(v)$, and then $(u, v)$ is an edge in the $b$-Matching.

Consider the situation when we attempt to find the $i$th neighbor for a vertex $u$ to propose to match to. At this stage $u$ has made $i - 1$ outstanding proposals to vertices in the set $T_{i-1}(u)$, the index showing the number of proposals made by $u$. We must have $i \leq b(u)$, for $u$ can have at most $b(u)$ outstanding proposals. If a vertex $u$ has fewer than $b(u)$ outstanding proposals, then we say that it is *unsaturated*; if it has $b(u)$ outstanding proposals, then it is *saturated*. The $b$-Suitor algorithm finds a partner for $u$, $p_i(u)$, according to the following equation:

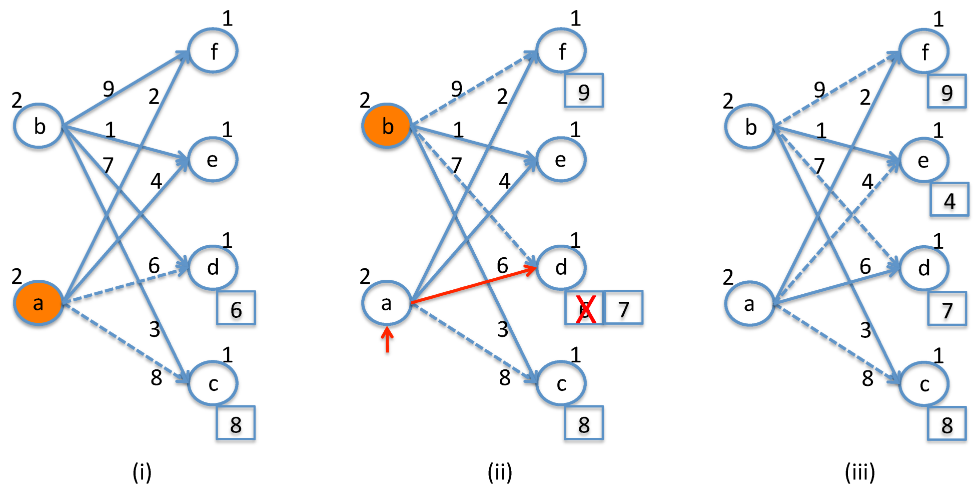
$$p_i(u) = \underset{v \in N(u) \setminus T_{i-1}(u)}{\arg\max} \{W(u,v) \mid W(u,v) > W(v, S(v).last)\}. \tag{3.1}$$

In words, the $i$th vertex that $u$ proposes to is a neighbor $v$ that it has not proposed to yet, such that the weight of the edge $(u,v)$ is maximum among such neighbors and is also greater than the lowest weight offer $v$ has currently. We will call such a vertex $v$ an *eligible partner* for $u$ at this stage in the algorithm. Note that the vertex $p_i(u)$ belongs to $T_i(u)$ but not to $T_{i-1}(u)$.

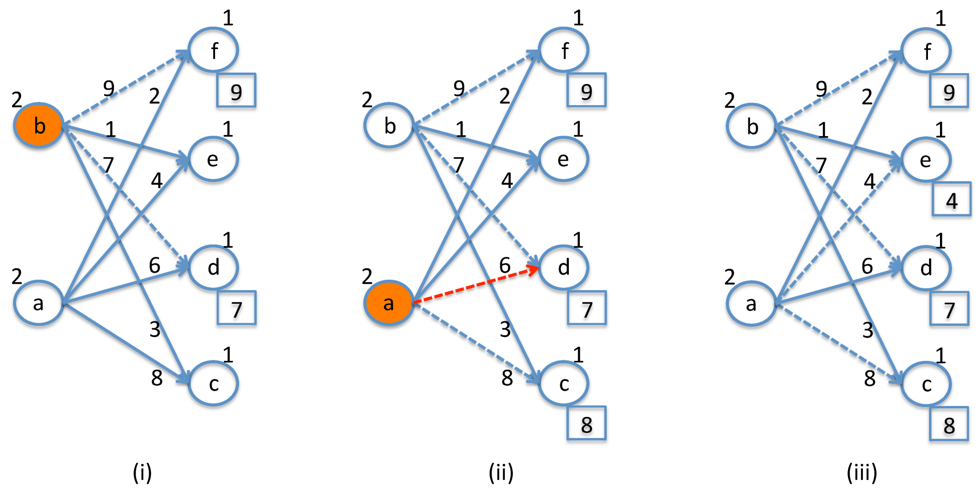
We present the pseudocode of the sequential $b$-SUITOR in Algorithm 1. We describe a recursive version of the algorithm since it is easier to understand, although the versions we have implemented for both serial and parallel algorithms use iteration rather than recursion. The algorithm processes all of the vertices, and for each vertex $u$, it seeks to match $b(u)$ neighbors. In each iteration a vertex $u$ proposes to a heaviest neighbor $v$ it has not proposed to yet if the weight $W(u,v)$ is heavier than the weight offered by the last ($b(v)$th) suitor of $v$. If it fails to find a partner, then we break out of the loop. If it succeeds in finding a partner $x$, then the algorithm calls the function MakeSuitor to make $u$ the suitor of $x$. This function updates the priority queue $S(u)$ and the array $T(u)$. When $u$ becomes the suitor of $x$, if it annuls the proposal of the previous suitor of $x$, a vertex $y$, then the algorithm looks for an eligible partner $z$ for $y$ and calls MakeSuitor recursively to make $y$ a suitor of $z$.

We illustrate a sequence of operations of the $b$-SUITOR algorithm in Figure 1. The figure shows a bipartite graph with weights on its edges and $b(v)$ values on the vertices. Thus vertices $a$ and $b$ both have $b(v) = 2$ and other vertices have $b(v) = 1$. In subfigure (a), step (i), the algorithm starts processing a vertex $a$, it finds the heaviest edge $W(a,c) = 8$, and $a$ proposes to vertex $c$. Vertex $c$ stores the weight of the offer it has in a local *min priority heap* $S$. Then $a$ finds its next heaviest unprocessed edge $W(a,d) = 6$ and also proposes to $d$. At this point $a$ has found $b(a) = 2$ partners, so the algorithm processes the next vertex $b$. In step (ii), $b$ proposes to its heaviest neighbor $f$ with weight $W(b,f) = 9$; its next heaviest neighbor is $d$ with $W(b,d) = 7$. Note that $b(d) = 1$, i.e., vertex $d$ can have at most one suitor, and it already has a suitor in $a$. Vertex $b$ checks the lowest offer vertex $d$ currently has, which is from vertex $a$, equal to 6. But vertex $b$ can make a higher offer to $d$, since $W(b,d) > W(a,d)$. Hence the algorithm makes $b$ the suitor of $d$, updates the lowest offer vertex $d$ has, from 6 to 7, and then removes $a$ from being the suitor of $d$. This removal of $a$ is important because $a$ now has one fewer partners. Eventually, the algorithm will process $a$ again, finds the edge $(a,e)$, and a becomes the suitor of $e$ in step (iii). When the vertices $c$ and $e$ propose to $a$, and vertices $d$ and $f$ propose to $b$, then (3.1) is satisfied by all the vertices, and we have a half-approximation matching.

To illustrate that the order of processing determines the work in the $b$-SUITOR algorithm, we show a different processing order in Figure 1(b). In step (i), the algorithm first processes vertex $b$, which proposes to $f$ and $d$. Next in step (ii), it processes vertex $a$, which proposes to $c$ as before. The next heaviest unprocessed neighbor of $a$ is $d$, but vertex $d$ already has an offer from $b$ of weight 7. The $b$-SUITOR algorithm checks the offer that vertex $d$ has so far, which is 7 from $b$, and correctly deduces that $a$ is not an eligible partner of $d$. So the algorithm moves on to the next heaviest edge incident on $a$, the edge $(a,e)$, and matches the edge in step (iii), thus saving redundant computations. (The LD algorithm would have vertex $a$ proposing to vertex $d$, which would eventually be rejected by $d$, and this in turn would initiate the search for a new partner for $a$.)



(a) The first execution.



(b) A second execution.

Fig. 1. *An example to illustrate the $b$-Suitor algorithm.*

Of course, we do not know the order in which the proposals in the $b$-Suitor algorithm would be made in general; this order would influence the work in the algorithm but not its correctness, since (3.1) would be satisfied by all vertices at the end of the algorithm. We will consider this aspect of the algorithm later.

**3.2. Proof of correctness.** Now we prove that the $b$-Suitor algorithm computes the same matching as the Greedy algorithm and hence that it is a half-approximation algorithm.

Lemma 3.1. *Equation* (3.1) *is satisfied by all the proposals made by the vertices in the $b$-Suitor algorithm.*

**Algorithm 1.** Sequential algorithm for approximate $b$-MATCHING. **Input:** A graph $G = (V, E, w)$ and a vector $b$. **Output:** A half-approximation edge weighted $b$-MATCHING $M$.

```
procedure $b$-SUITOR($(G, b)$)
    for all $u \in V$ do
        for $i = 1$ to $b(u)$ do
            $x = \underset{v \in N(u) \setminus T(u)}{\arg\max} \{W(u, v) : W(u, v) > W(v, S(v).last)\}$
            if $x = NULL$ then
                break
            else
                MakeSuitor($u, x$)
```

**Algorithm 2.** Recursive function MakeSuitor.

```
function MAKESUITOR($(u, x)$)
    $y = S(x).last$
    $S(x).insert(u)$
    $T(u).insert(x)$
    if $y \neq NULL$ then
        $T(y).remove(x)$
        $z = \underset{v \in N(y) \setminus T(y)}{\arg\max} \{W(y, v) : W(y, v) > W(v, S(v).last)\}$
        if $z \neq NULL$ then
            MakeSuitor($y, z$)
```

*Proof.* The proof is by induction on the number of proposals in the algorithm. Initially there are no proposals and the equation is trivially satisfied. Note that the variable $x$ in the $b$-SUITOR algorithm corresponds to an eligible partner for each vertex such that (3.1) is satisfied. Assume that the invariant is true for the first $k \geq 0$ proposals. Let the $(k + 1)$st proposal be made by a vertex $u$ to find its $(i + 1)$st partner, $p_{i+1}(u)$. There are three cases to consider:

1. $p_{i+1}(u) = NULL$, i.e., there is no neighbor of $u$ satisfying (3.1). Then the invariant is trivially satisfied since $u$ does not extend a new proposal.
2. If $p_{i+1}(u) = x$ and $x$ has fewer than $b(x)$ suitors, the invariant is satisfied because the node $x$ offers a better weight than the NULL vertex, whose weight is zero.
3. If $p_{i+1}(u) = x$ and $x$ has $b(x)$ suitors, i.e., $W(u, x) > W(x, v)$, where $v = S(x).last$, we maintain the invariant when $u$ proposes to $x$, and annuls the proposal from $x$ to $v$, since the weight offered by $u$ to $x$ is greater than the offer of $v$.

But $v$ now has one fewer partners, and the algorithm seeks a new partner for $v$. Again, the function MakeSuitor searches for an eligible partner for the annulled vertex $v$ satisfying (3.1). If it fails to find a partners, then the invariant holds since no new proposal has been made. If it finds a partner, then again the invariant holds since an eligible partner is chosen to satisfy the invariant.

However, the annulment of proposals could cascade, i.e., a vertex that gets annulled could cause another to be annulled in turn, and so on. However, since (3.1) looks for an eligible partner with a higher weight than the current lowest offer the partner has, the cascading cannot cycle.

Since a vertex $x$ annuls the proposal of another vertex $y$ which proposed earlier to $u$, and there are fewer than $n$ such vertices at any stage of the algorithm, the cascading will terminate after at most $n-1$ steps. At that point the invariant holds for all proposed vertices. This completes the proof. ∎

LEMMA 3.2. *At the termination of the $b$-SUITOR algorithm, $u \in S(v) \leftrightarrow v \in S(u)$.*

*Proof.* For one direction, assume $u \in S(v)$ and $v \notin S(u)$. If $u$ has fewer than $b(u)$ partners and $v$ has fewer than $b(v)$ partners, then $v$ would propose to $u$ and become a suitor of $u$. Hence assume that $|S(u)| = b(u)$. Then for all $t \in S(u) : W(u,t) > W(u,v)$, and $u$ has $b(u)$ partners to satisfy (3.1) and so does not propose to $v$. Thus $u \notin S(v)$, which is a contradiction.

The other direction follows from the symmetry of $u$ and $v$. ∎

Hence when the $b$-SUITOR algorithm terminates, all proposals correspond to matched edges.

LEMMA 3.3. *If (3.1) is satisfied for all $u \in V$, then $p_i(.)$ defines the same matching as the GREEDY algorithm, provided ties in weights are broken consistently in the two algorithms.*

*Proof.* The proof is by induction on the sequence of the matched edges chosen by the GREEDY algorithm. The base case is when the matching is empty, and in this case, the lemma is trivially true. Assume that both the GREEDY and $b$-SUITOR algorithms agree on the first $k$ edges matched by the former. In order to match these $k$ edges, the GREEDY algorithm has examined a subset of edges $E' \subset E$, matching some of them and rejecting the others. An edge $(u,v)$ is rejected by GREEDY when either of its endpoints is saturated, i.e., it already has $b(u)$ or $b(v)$ matched neighbors. Note that the edge with the least weight in $E'$ is at least as heavy as the heaviest edge in $F = E \setminus E'$. Therefore, GREEDY will choose the $(k+1)$st matched edge from $F$.

Assume that the GREEDY algorithm rejects $t \geq 0$ heaviest edges in $F$ until it finds an edge $(u,v)$ whose endpoints have fewer than $b(u)$ and $b(v)$ matched neighbors, respectively. This edge is now chosen as the $(k+1)$st matched edge. We show that $b$-SUITOR will also not match these $t$ edges and pick $(u,v)$ as a matched edge. Note that all of these $t$ edges have at least one of their incident vertices as saturated. In order for $b$-SUITOR to match any of these edges, it has to unmatch at least one edge from the $k$ matched edges. But all the $k$ matched edges are heavier than any of these $t$ edges, and unmatching an edge in $E'$ to match one of the $t$ rejected edges would violate (3.1).

It remains to show that the edge $(u,v)$ will not be unmatched by the $b$-SUITOR algorithm later and will be included in the final matching. This is clear because the GREEDY algorithm chose $(u,v)$ from a globally sorted set of edges, which means $W(u,v)$ is an LD edge in both $u$'s and $v$'s neighborhoods once earlier matched edges are excluded. ∎

The two lemmas lead to the following result.

THEOREM 3.4. *When ties in the weights are broken consistently, the $b$-SUITOR algorithm matches the same edges as the GREEDY algorithm, and hence it computes a half-approximation matching.*

The running time of the algorithm is $O(\Sigma_{u \in V} \delta(u)^2 \log \beta) = O(m\Delta \log \beta)$. This follows since a node $u$ might have to traverse its neighbor list at most $\delta(u)$ times to find a new partner for each of its $b(u)$ matched edges, and each time we find a candidate, updating the heap costs $O(\log \beta)$. For small $\beta \in \{2,3\}$, we use an array instead of a heap to avoid the heap updating cost.

If we completely sort the adjacency list of each vertex in decreasing order of weights, it needs to be traversed only once in the $b$-SUITOR algorithm. For, when a vertex $x$ is passed over in an adjacency list of a vertex $v$, $x$ has a better offer than the weight $v$ can offer, and as the algorithm proceeds, the weight of the lowest offer that $x$ has can only increase. Hence $v$ does not need to extend a proposal to the vertex $x$ in the future. But sorting adds an additional cost of $O(m \log \Delta)$, so that the time complexity of this variant of the algorithm is $O(m \log \Delta + m \log \beta) = O(m \log \beta\Delta)$. By partially sorting the adjacency list, and choosing the part sizes to sort carefully, we can reduce the time complexity even further, and this is discussed in section 3.4.

### 3.3. The parallel $b$-SUITOR algorithm.

In this subsection we describe a shared-memory parallel $b$-SUITOR algorithm. It uses iteration rather than recursion; it queues vertices whose proposals have been rejected for later processing, unlike the recursive algorithm that processes them immediately. It is to be noted that $b$-SUITOR finds the solution irrespective of the order in which the vertices are processed, which means the solution is stable no matter how the operating system schedules the threads. It uses locks for synchronizing multiple threads to ensure sequential consistency.

The parallel algorithm is described in Algorithm 3. The algorithm maintains a queue of unsaturated vertices $Q$, which it tries to find partners for during the current iteration of the **while** loop, and also a queue $Q'$ of vertices whose proposals are annulled in this iteration to be processed again in the next iteration. The algorithm then attempts to find a partner for each vertex $u$ in $Q$ in parallel. It tries to find $b(u)$ proposals for $u$ to make while the adjacency list $N(u)$ has not been exhaustively searched thus far in the course of the algorithm.

**Algorithm 3.** Multithreaded shared-memory algorithm for approximate $b$-MATCHING. **Input:** A graph $G = (V, E, w)$ and a vector $b$. **Output:** A half-approximation edge weighted $b$-MATCHING $M$.

```
procedure PARALLEL_b-SUITOR($G, b$)
    $Q = V$; $Q' = \emptyset$;
    while $Q \neq \emptyset$ do
        for all vertices $u \in Q$ in parallel do
            $i = 1$;
            while $i <= b(u)$ and $N(u) \neq exhausted$ do
                Let $p \in N(u)$ be an eligible partner of $u$;
                if $p \neq NULL$ then
                    Lock $p$;
                    if $p$ is still eligible then
                        $i = i + 1$;
                        Make $u$ a Suitor of $p$;
                        if $u$ annuls the proposal of a vertex $v$ then
                            Add $v$ to $Q'$; Update $db(v)$;
                    Unlock $p$;
                else
                    $N(u) = exhausted$;
        Update $Q$ using $Q'$; Update $b$ using $db$;
```

Consider the situation when a vertex $u$ has $i - 1 < b(u)$ vertices outstanding proposals. The vertex $u$ can propose to a vertex $p$ in $N(u)$ if it is a heaviest neighbor in the set $N(u) \setminus T_{i-1}(u)$ and if the weight of the edge $(u, p)$ is greater than the lowest offer that $p$ has. In this case, we say that $p$ is an eligible partner for $u$. (Thus $p$ would accept the proposal of $u$ rather than its current lowest offer.)

If the algorithm finds a partner $p$ for $u$, then the thread processing the vertex $u$ attempts to acquire the lock for the priority queue $S(p)$ so that other vertices do not concurrently become suitors of $p$. This attempt might take some time to succeed since another thread might have the lock for $p$. Once the thread processing $u$ succeeds in acquiring the lock, then it needs to check again if $p$ continues to be an eligible partner, since by this time another thread might have found another suitor for $p$, and its lowest offer might have changed. If $p$ is still an eligible partner for $u$, then we increment the count of the number of proposals made by $u$ and make $u$ a suitor of $p$. If in this process we dislodge the last suitor $x$ of $p$, then we add $x$ to the queue of vertices $Q'$ to be processed in the next iteration. Finally the thread unlocks the vertex $p$.

Now we can consider what happens when we fail to find an eligible partner $p$ for a vertex $u$. This means that we have exhaustively searched all neighbors of $u$ in $N(u)$, and none of these vertices offers a weight greater than the lowest offer $u$ has, $S(u).last$. After we have considered every vertex $u \in Q$ to be processed, we can update data structures for the next iteration. We update $Q$ to be the set of vertices in $Q'$ and the vector $b$ to reflect the number of additional partners we need to find for each vertex $u$ using $db(u)$, the number of times $u$'s proposal was annulled by a neighbor.

Since the set of edges that satisfy (3.1) in the entire graph is unique (with our tie breaking scheme for weights), the order in which we consider the edges does not matter for correctness by Lemma 3.1. However, this order will influence the work done by the algorithm, and by processing the adjacency list of vertices in decreasing order of weights, we expect to reduce this work.

### 3.4. Variants of the *b*-Suitor algorithm.

We consider three orthogonal variations to make the $b$-Suitor algorithm more efficient. The first concerns the sorting of the adjacency lists of the vertices, the second considers when a vertex whose proposal is annulled should extend a new proposal, and the third involves the order in which vertices should extend proposals.

#### 3.4.1. Neighbor sorting.

Since we need to find only $b(v) \leq \delta(v)$ mates for each vertex $v$, sorting the entire adjacency list is often unnecessary. Hence we consider partially sorting the adjacency lists so that for each vertex $v$ we list $p(v) \geq b(v)$ of the heaviest neighbors in decreasing order of weights. We try to match edges belonging to this subset at first. The value $p(v)$ is a key parameter that determines both the work to be done in partial sorting as well the probability that we can find the matching using edges solely from this subset. Given an adjacency list $A(v) = \text{adj}(v)$ and a subset size $p(v)$, we find the heaviest $p(v)$ neighbors from $A(v)$ and sort only this subset. The *pth* largest neighbor is found using an algorithm similar to the partition function in Quicksort. The adjacency list is partitioned into two subsets with respect to the pivot, and the subset with heavier edges than the pivot is sorted.

What should an algorithm do when a vertex $v$ exhausts its partially sorted subset of neighbors without finding $b(v)$ partners? We consider two schemes: (i) falling back to the unsorted mode for the rest of the neighbor list and (ii) computing the next heaviest batch of $p(v)$ neighbors by partial sorting. In the serial $b$-Suitor algorithm, we use the batching scheme. In the parallel version of $b$-Suitor, falling back to the unsorted scheme may be useful in some cases, more specifically in combination with the *eager update* case to be discussed next.

For the algorithm that employs partial sorting with batching, the total running time is $O(mc+ncp\log p+m\log\beta)\approx O(m(c+\log\beta))$, where $p$ is the maximum subset size over all vertices, and $c$ is the maximum number of batches (subsets) required for any node. Here the first term comes from the selection of the pivots in each adjacency list, the second term from the partial sorting, and the final term from updating the heap as edges are added or deleted from it. The number of batches $c$ satisfies $1\le c\le\max_{v\in V}\delta(v)/p(v)$. If we choose $p(v)$ carefully, we could avoid having to sort more than a few batches, and hence the number of batches could be bounded by a small constant. In practice, with good choices that will be discussed in section 4, we observe that the average number of batches per node is indeed bounded by a constant. Hence the time complexity of the partially sorted $b$-Suitor algorithm becomes $O(m\log\beta)$.

**3.4.2. Delayed versus eager processing.** We consider two strategies for how the algorithm treats a vertex $x$ whose proposal is annulled. The algorithm can either immediately have $x$ extend another proposal (an *eager update*) or put it a separate queue for later processing (a *delayed update*). The recursive $b$-Suitor algorithm described earlier is the *eager update* variant. The downside of this scheme is that as soon as the algorithm makes the vertex $v$ as the *current* vertex, we could lose memory locality. Also the vertex $v$'s next offer to a neighbor will be lower than its rejected offer, and it could be rejected again. To make matters worse, the annulment operations could cascade. Another issue with this scheme for $b$-Matching is that a vertex $v$ can have more than one proposal annulled at the same time. In a partially sorted scheme, two threads may initiate partial sorting on $v$'s neighbor list, which requires synchronization, causing further overhead. Our experiments show that falling back to unsorted mode mentioned above performs better with the *eager update*.

If a vertex $x$ has $k$ proposals annulled, the algorithm needs to find $k$ new partners for $x$. In the *delayed* scheme the dislodged vertex $x$ is stored in a queue $Q'$ once, and we count the number of times it gets dislodged. After the current iteration is done, the algorithm can start processing all the deficient vertices $x$ from the queue $Q'$, and it processes the vertex $x$ to find multiple partners.

Considering all these enhancements, we have six variations of our $b$-Suitor algorithm. We name these schemes as follows: (i) eager unsorted (ST EU), (ii) eager sorted (ST ES), (iii) eager partially sorted (ST EP), (iv) delayed unsorted (ST DU), (v) delayed sorted (ST DS), and (vi) delayed partially sorted (ST DP).

**3.4.3. Order in which vertices are processed.** We can also investigate how the order in which vertices make proposals influences the work in the serial $b$-Suitor algorithm.

A sophisticated approach is to partition the edges into heavy and light edges based on their weight and to consider only the heavy edges incident on the vertices as candidates to be matched. Hence initially each vertex only makes as many proposals as the number of heavy edges incident on it. If this initial phase succeeds in finding $b(v)$ matches for a vertex $v$, then we are done with that vertex. If it does not, then in a second phase, $v$ proposes to the neighbors of $v$ that are incident on it through the light edges to make up the deficit if possible. Giving higher priority to heavier edges to be involved in proposals could decrease the number of annulled proposals in the algorithm and potentially search fewer edges leading to faster run times. Clearly in this case, finding a good value for the pivot element to split each adjacency list into heavy and light edges is important. Recall that $B=\sum_{v\in V}b(v)$. We have empirically chosen the weight of the $(kB)$th element as this pivot value, where $k$ is a small integer typically between 1 and 5.

A simpler approach is to sort each vertex by the heaviest edge incident on it and then to process the vertices in the algorithm in this order. This scheme has the advantage of simplicity, and low overhead cost, since it needs only to sort the vertices by a key value.

## 4. Experiments and results.

We conducted our experiments on an Intel Xeon[1] E5-2670 processor based system (part of the Purdue University Community Cluster, called *Conte*[2] ). The system consists of two processors, each with 8 cores running at 2.6 GHz (16 cores in total) with 32 KB of L1, 256 KB of L2, 20 MB of unified L3 cache, and 64 GB of memory. The system is also equipped with a 60-core Intel Xeon Phi coprocessor running at 1.1 GHz with 32 KB L1 data cache per core and 512 KB unified L2 cache. The operating system is Red Hat Enterprise Linux 6. All the codes were developed using C++ and compiled using the Intel C++ Composer XE 2013 compiler[3] (version 1.1.163) using the -O3 flag.

Our test problems consist of both real-world and synthetic graphs. Synthetic datasets were generated using the Graph500 RMAT data generator [31]. We generate three different synthetic datasets varying the RMAT parameters (similar to previous work [24]). These are (i) *rmat_b* with parameter set (0.55, 0.15, 0.15, 0.15), (ii) *rmat_g* with parameter set (0.45, 0.15, 0.15, 0.25), and (iii) *rmat_er* with parameter set (0.25, 0.25, 0.25, 0.25). These graphs have varying degree distributions representing different application areas. We also consider a random geometric graph (*geo*_14) [35] that has recently attracted attention in the study of neural networks, astrophysics, etc.

Additionally we consider ten real-world datasets taken from the University of Florida Matrix collection [5] covering mutually exclusive application areas such as medical science, structural engineering, and sensor data.

Table 1 shows the sizes of our test problems, which are divided into two subsets: the first nine problems with tens of millions of edges or greater, and the last five with low edge counts. The larger problems are the five largest symmetric problems from the Florida Matrix collection. The other five out of these ten real-world problems have fewer than a million edges, and we use these to compare the matching quality of the approximation algorithms relative to the exact matching. The reason for using smaller datasets for quality comparison is that we do not have a good implementation for exact non-perfect $b$-MATCHING. The exact $b$-MATCHING algorithm that we compare $b$-SUITOR to uses integer linear programming and is programmed in MATLAB. This implementation can run only on the smaller datasets due to memory limitations, and hence we selected the largest symmetric problems that this implementation could solve.

The space complexity of the best performing delayed partial $b$-SUITOR algorithm is $2m + 5n + n\beta$. Our largest problem *rmat_er* requires roughly 572 MB of memory when $\beta = 10$. For each vertex, $b$-SUITOR accesses its adjacency list and the heap

[1]Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

[2]https://www.rcac.purdue.edu/compute/conte/.

[3]Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel micro-architecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

TABLE 1
*The structural properties of our test problems for* $b$-MATCHING.

| Problem | Vertices | Edges | Avg. Deg. | Max. Deg. |
|---|---|---|---|---|
| Fault_639 | 638,802 | 13,987,881 | 11 | 317 |
| mouse_gene | 45,101 | 14,461,095 | 160 | 8,031 |
| bone010 | 986,703 | 35,339,811 | 18 | 80 |
| dielFil.V3real | 1,102,824 | 44,101,598 | 20 | 269 |
| kron.logn21 | 2,097,152 | 91,040,932 | 22 | 213,904 |
| geo_14 | 16,384 | 54,689,168 | 3274 | 10,365 |
| rmat_b | 1,048,576 | 123,599,502 | 58 | 63,605 |
| rmat_g | 1,048,576 | 133,056,675 | 64 | 7,998 |
| rmat_er | 1,048,576 | 134,201,240 | 64 | 337 |
| astro-ph | 16,706 | 121,251 | 8 | 360 |
| Reuters911 | 13,332 | 148,038 | 12 | 2265 |
| cond-mat-2005 | 40,421 | 175,693 | 5 | 278 |
| gas_sensor | 66,917 | 885,141 | 14 | 33 |
| turon_m | 189,924 | 912,345 | 5 | 11 |

associated with its neighbors. We use the compressed row storage format for storing the adjacency list and the heap size for each node is at most $\beta$. In shared memory, each thread processes one vertex at a time and there are 16 such threads. So the working memory requirement in parallel execution of the algorithm is roughly $16 * (\delta_{avg} + \delta_{avg} * \beta)$, where $\delta_{avg}$ is the average degree of a vertex. For example, with $\beta = 10$, the working memory for *rmat_er* is roughly 500 KB, which fits in the L3 cache of the Xeon and barely so in the L2 cache of the Phi.

### 4.1. Serial performance.

We analyze the serial performance of 12 algorithms: GREEDY, PGA, PGA′, three variants of LD, and six variations of $b$-SUITOR. The LD algorithms differ in terms of whether the neighbor list of each vertex is unsorted, sorted or partially sorted; we call these *LD U*, *LD S*, and *LD P*, respectively.

We have considered two sets of $b(v)$ values for the experiments. First we fix $b(v) = \min\{b, \delta(v)\}$, where $b \in \{1, 3, 5, 10, 15\}$ for all the vertices. The reason for using constant values of $b(v)$ is to study how the algorithms perform as the value of $b(v)$ is varied. The second set of $b(v)$ values is motivated by the privacy application in [3]. Here we randomly assign values of $b(v)$ for each vertex $v$, choosing $b(v)$ uniformly between 1 and $(\delta(v))^{1/2}$. We generate three sets of $b(v)$ values for each problem and select the set which leads to the median weight for its $b$-MATCHING. We use these randomized $b(v)$ values for all the experiments. The average randomized $b(v)$ values for all the problems are between 3 and 9, except for the relatively dense (geo_14) problem, where the average value of $b(v)$ is 40. In the experiments, we explicitly list the $b$ values unless we use randomized values for them.

#### 4.1.1. Matching weight.

We have compared the weight of the heuristic HEM algorithm and the PGA with the algorithms that we consider in this section. We found that the quality of the weighted $b$-MATCHINGS obtained by the former algorithms was dependent on the order in which the vertices were processed. We repeated the computations for the original ordering and three random orderings of the vertices. For HEM, the coefficient of variation was 1.3, whereas for the PG algorithm it was 0.8. The GREEDY, LD, and $b$-SUITOR algorithms all find the identical matching irrespective of vertex ordering, and the weight is not affected by the order in which the vertices are processed. For this reason, we do not consider the HEM algorithm any further.

TABLE 2
*Total edge weight of b-*MATCHING *produced by b-*SUITOR *compared to the optimal solution.*

| Problem | Quality in % ($b = 1$) | Quality in % ($b = 3$) | Quality in % ($b = 5$) | Quality in % ($b$ random) |
|---|---|---|---|---|
| astro-ph | 97.89 | 98.39 | 98.75 | 99.09 |
| Reuters911 | 97.11 | 97.73 | 98.06 | 98.36 |
| cond-mat-2005 | 97.45 | 98.48 | 99.07 | 99.19 |
| gas_sensor | 98.06 | 99.17 | 99.44 | 99.38 |
| turon_m | 99.67 | 99.90 | 100.00 | 99.94 |
| Geo mean | 98.03 | 98.73 | 99.06 | 99.19 |

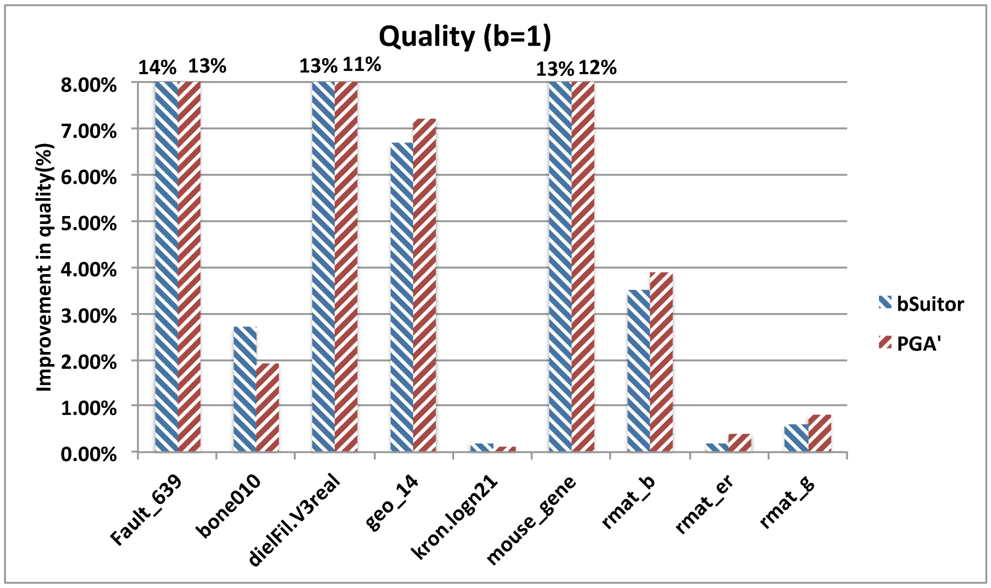
In Table 2 we compare the total edge weight computed by $b$-SUITOR with the edge weight of an optimal $b$-MATCHING for different choices of $b(v)$. We see that although $b$-SUITOR guarantees only half approximation in the worst case, it finds more than 97% of the optimal weight for the five smaller problems where we could run the exact algorithm. The matching weight increases with higher $b(v)$ values and greater number of edges.

We compare the weights of the matchings obtained by different approximate $b$-MATCHING algorithms in Figure 2. Since the LD, GREEDY, and $b$-SUITOR algorithms find exactly the same set of matched edges when ties are broken consistently, we need to compare the quality of the matching only among $b$-SUITOR and PGA$'$ relative to PGA. We report the percent improvement of matching quality for $b(v) = \{1, 10\}$. We observe that $b$-SUITOR and PGA$'$ give better quality matchings, up to 14% relative to PGA, for $b(v) = 1$. But with the higher $b(v) = 10$, the improvements are lower, up to 9%; also the differences between $b$-SUITOR and PGA$'$ are smaller. This is an important insight for the $b$-MATCHING problem overall relative to the MATCHING problem. Given a fixed $b$ value, the expected number of matched edges is $(n \times b)/2$, meaning that the set of matched edges is larger with higher $b(v)$. As the solution set gets larger, it is less likely that a $b$-MATCHING algorithm will miss out on a good edge. Therefore, the difference in matching quality will tend to be smaller among the $b$-MATCHING algorithms than for MATCHING algorithms; this difference should also decrease with increasing $b(v)$ values.

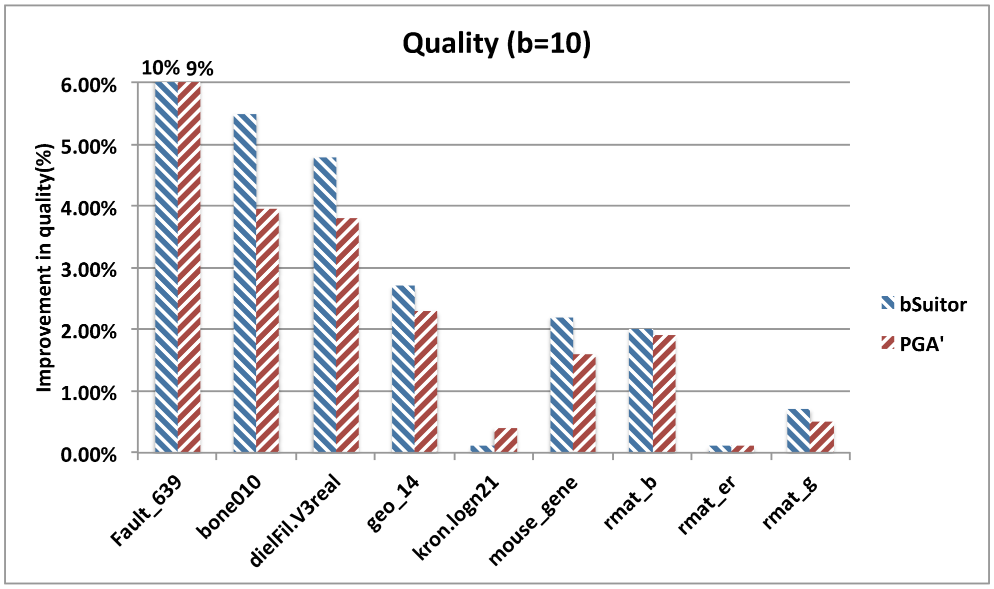
**4.1.2. Run times of the algorithms.** There are two factors that influence the run times of these algorithms: (i) the number of edge traversals required and (ii) the memory access times, primarily determined by the number of cache misses. Unfortunately the edges searched in sorting or partially sorting the adjacency lists are not included in the first term, since we use a system sort and accessing these numbers is not straightforward. We consider these two issues one by one.

Recall that the partially sorted variants of the LD and $b$-SUITOR algorithms work with a subset of the adjacency list of each vertex. We chose the size of this sorted subset $p(v)$ empirically to reduce the number of edge traversals. For 1-matching, the best value of $p(v) = 9 * b(v)$ for all problems except **kron.logn21**, for which the multiplier is 11, and **geo_14**, for which it is 14. For 10-matching, the best value of the multiplier was 3 for the first group of problems, 5 for **kron.logn21**, and 9 for **geo_14**. For each problem, these values are nearly identical for the variants of the $b$-SUITOR algorithm and the variants of the LD algorithm, since all of them compute the same $b$-MATCHING (identical to the GREEDY algorithm; cf. Theorem 3.4). With these choices for $p(v)$, for more than 90% of the vertices, one batch sufficed to find their mates in the $b$-MATCHING.

The number of edge traversals is the total number of times the edges of a graph are searched to compute the matching. We compare the number of edge traversals



(a) Quality (b=1)



(b) Quality (b=10)

FIG. 2. *Improvement in weight (%) of matchings from $b$-*SUITOR *and* PGA′ *algorithms w.r.t.* PGA.

for 1-Matching in Table 3 and for 10-Matching in Table 4.

Since we present results in the same table format in this section, we discuss here this manner of describing the performance of an algorithm. We show only the unsorted and partially sorted variants of the LD and $b$-SUITOR algorithms because the run times for the fully sorted variations are not competitive due to the costs of complete sorting. (The adjacency list of every vertex is sorted, not the edge lists.) Our performance

TABLE 3
*Number of edge traversals (in millions) for* 1-MATCHING.

| Problem | LD U | LD P | PGA′ | ST EU | ST EP | ST DU | ST DP |
|---|---|---|---|---|---|---|---|
| Fault_639 | 84 | **2** | 30 | 46 | 14 | 41 | 3 |
| bone010 | 2,950 | 4 | 71 | 147 | 9 | 148 | **4** |
| dielFil.V3real | 624 | 12 | 94 | 146 | 10 | 153 | **12** |
| geo_14 | 334 | 7 | 109 | 232 | 23 | 178 | **6** |
| kron.logn21 | 390 | 23 | 183 | 231 | 49 | 223 | **18** |
| mouse_gene | 283 | 25 | 29 | 58 | 37 | 53 | **1** |
| rmat_b | 531 | 11 | 248 | 330 | 20 | 314 | **10** |
| rmat_er | 577 | 14 | 268 | 400 | 22 | 369 | **13** |
| rmat_g | 571 | 13 | 266 | 380 | 22 | 355 | **12** |
| Avg Rel Perf | 67.89 | 1.51 | 15.93 | 25.67 | 2.96 | 23.87 | **1.00** |

metrics are the number of edge traversals, the number of cache misses, and run times; for all of these, lower values are better. The entries in the table are absolute values of the metrics and the winners are highlighted in bold font. The last row of a table is the average relative performance which shows the relative performance of the algorithms with respect to the delayed partial variant of the $b$-SUITOR algorithm (ST DP). This value is calculated for a specific algorithm by dividing the metric for that algorithm by the ST DP metric for each problem, and then computing the geometric mean of these normalized quantities. Thus the average relative performance of ST DP is always 1 and that of a better (worse) performing algorithm is less (greater) than 1.

In Table 3, we observe that for MATCHING (1-MATCHING), the delayed update and partially sorted adjacency list (DP) variant of the $b$-SUITOR algorithm requires the fewest edge traversals for all except one problem. If we divide the number of edges traversed by this algorithm by the total number of edges for each problem, the mean is 14%. This clearly shows that the subsets of the adjacency lists we sort suffice to find the MATCHING using the DP $b$-SUITOR algorithm. In the eager update and partially sorted version of the $b$-SUITOR algorithm, once we exhaust the first batch in each adjacency list, we switch to unsorted mode, and this leads to increased number of edges traversed (by a factor of two or more over DP). The reasons for this switch are explained in section 3.4. The LD algorithm also benefits from partial sorting, and this is the second best performer with respect to this metric. The unsorted variants of the algorithms require 15 to 60 times more edge traversals than the DP variant of $b$-SUITOR.

We see the same effect, in an even more pronounced manner, in Table 4 for 10-MATCHING. Note that the multiplier here is smaller (3 for most problems) because the value of $b(v)$ is higher. Again ST DP is the best performer, and the partial sort reduces the number of edge traversals from their unsorted counterparts by two to three orders of magnitude. The DP $b$-SUITOR algorithm traverses fewer edges than the second best performer, the partially sorted variant of the LD algorithm, by a factor greater than six.

Now we turn to cache misses. In Figure 3, we show the number of $L1$, $L2$, and $L3$ cache misses (in logarithmic scale) relative to the ST DP scheme of $b$-SUITOR for the *rmat_er* problem. We observe that PGA′ incurs the highest number of cache misses, $2^8\times$ with respect to the ST DP algorithm, since it is depth-first-search based. We also observe that the unsorted versions LD U, ST EU, and ST DU have higher numbers of cache misses than their corresponding (partially) sorted versions, because sorting allows the algorithms to scan the adjacency lists only once in order to find the heaviest remaining edges.

TABLE 4
*Number of edge traversals (in millions) for* 10-MATCHING.

| Problem | LD U | LD P | PGA′ | ST EU | ST EP | ST DU | ST DP |
|---|---|---|---|---|---|---|---|
| Fault_639 | 586 | 15 | 115 | 135 | 9 | 128 | **4** |
| bone010 | 11,576 | 165 | 287 | 416 | 14 | 396 | **7** |
| dielFil.V3real | 4,772 | 71 | 187 | 620 | 100 | 621 | **14** |
| geo_14 | 20,554 | 8 | 3,622 | 6,590 | 769 | 5,976 | **6** |
| kron.logn21 | 62,326 | 662 | 183 | 6,033 | 116 | 5,896 | **20** |
| mouse_gene | 16,954 | 33 | 116 | 1,011 | 291 | 944 | **3** |
| rmat_b | 66,072 | 358 | 3,410 | 6,434 | 107 | 6,227 | **18** |
| rmat_er | 7,569 | 43 | 3,760 | 2,938 | 133 | 2,781 | **22** |
| rmat_g | 18,805 | 99 | 799 | 3,713 | 109 | 3,545 | **21** |
| Avg Rel Perf | 1133.78 | 6.70 | 50.98 | 153.11 | 8.62 | 146.05 | **1.00** |

We also see that ST DS and ST DP incur the smallest number of cache misses with ST DS slightly less than that of ST DP. We observe a similar pattern for all other problems.

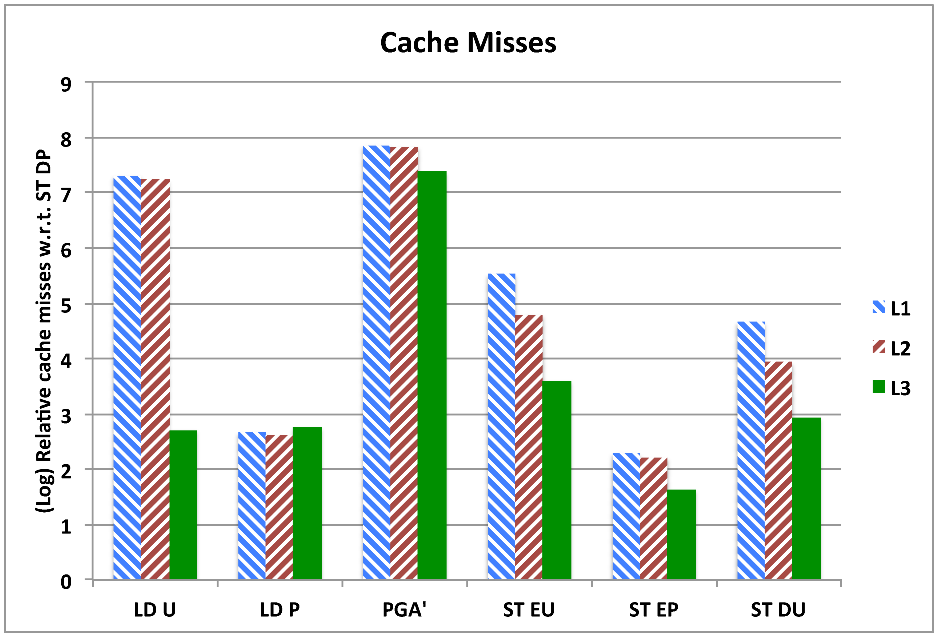


FIG. 3. *Relative number of cache misses for various algorithms with respect to the delayed partial variant of the* $b$-SUITOR *algorithm for the* rmat_er *problem.*

The previous two results confirm our claim that $b$-SUITOR is more efficient than the other algorithms both in terms of the number of edge traversals and in the cache hits. These efficiencies translate into serial run time performance on the Intel Xeon processor, which we compare in Tables 5 and 6.

In Table 5, we report the run time results of seven algorithmic variants for $b$-MATCHING with $b$ = 10 on nine of the larger problems. Due to the number of columns we can include in the table, we have not shown results for the PGA′ algorithm since it was a poor performer. We have listed the problems in increasing order of the number of edges (in millions) traversed by the DP SUITOR algorithm (in the last column) and show that it correlates quite well with the run times of the DP SUITOR

TABLE 5
*Serial run times on Intel Xeon processor for b-MATCHING with $b = 10$.*

| Problem | Greedy | LD U | LD P | ST EU | ST EP | ST DU | ST DP | EdgeTrv (ST DP) |
|---|---|---|---|---|---|---|---|---|
| mouse_gene | 47.95 | 308.63 | 15.99 | 5.61 | 5.29 | 5.90 | 4.19 | 3 |
| Fault_639 | 51.01 | 19.38 | 12.90 | **4.35** | 6.95 | 4.63 | 5.86 | 4 |
| geo_14 | 102.75 | 221.32 | 5.01 | 9.02 | 2.07 | 6.69 | **1.98** | 6 |
| bone010 | 121.31 | 343.97 | 85.08 | 10.43 | 9.66 | 12.92 | **8.48** | 7 |
| dielFil.V3real | 173.41 | 72.82 | 37.89 | 17.84 | 14.85 | 12.62 | **9.53** | 14 |
| rmat_b | 526.51 | 2768.24 | 68.71 | 37.20 | 31.10 | 32.10 | **24.83** | 18 |
| kron.logn21 | 368.89 | 2881.20 | 76.32 | 27.93 | 23.98 | 25.21 | **19.36** | 20 |
| rmat_g | 585.87 | 829.32 | 50.55 | 42.62 | 40.07 | 42.39 | **31.11** | 21 |
| rmat_er | 589.90 | 830.23 | 54.16 | 45.28 | 44.28 | 45.70 | **35.01** | 22 |
| Avg Rel Perf | 17.72 | 35.45 | 3.06 | 1.51 | 1.24 | 1.41 | **1.00** | |

TABLE 6
*Serial run times on Intel Xeon processor for b-MATCHING with random $b(v)$ values.*

| Problem | Greedy | LD U | LD P | PGA′ | ST EU | ST EP | ST DU | ST DP |
|---|---|---|---|---|---|---|---|---|
| Fault_639 | 14.83 | 6.37 | 4.29 | 2.62 | 1.00 | 1.01 | **1.00** | 1.15 |
| bone010 | 73.41 | 81.50 | 22.47 | 4.59 | 2.47 | 1.97 | 2.30 | **1.72** |
| dielFil.V3real | 62.74 | 30.59 | 15.48 | 11.92 | 3.11 | 2.98 | 3.45 | **2.71** |
| geo_14 | 130.40 | 462.50 | 9.87 | 218.51 | 18.51 | 4.89 | 16.51 | **4.50** |
| kron.logn21 | 84.32 | 972.16 | 67.27 | 283.63 | 23.08 | 5.61 | 20.92 | **5.53** |
| mouse_gene | 10.71 | 92.23 | 12.79 | 21.57 | 3.45 | 2.57 | 3.07 | **2.18** |
| rmat_b | 93.62 | 880.18 | 47.42 | 256.13 | 25.68 | 6.75 | 22.17 | **6.52** |
| rmat_er | 94.74 | 87.49 | 28.97 | 582.99 | 18.00 | 8.87 | 14.73 | **8.48** |
| rmat_g | 99.17 | 225.96 | 42.75 | 66.60 | 19.37 | 8.24 | 15.95 | **7.95** |
| Avg Rel Perf | 15.62 | 35.50 | 5.35 | 13.50 | 2.10 | 1.05 | 1.91 | **1.00** |

algorithm. There are only two exceptions to greater run times with increase in the number of edges traversed. The first of these is the geometric random graph `geo_14`, where the run time is lower than expected. It has an extremely high average vertex degree, which makes it easy to find $b$-MATCHING. The second is the `kron.logn21` problem, which has edges all of equal weight.

Since the cache misses for the `rmat_er` problem are shown in Figure 3, for this problem we can try to correlate the run times of the seven algorithms with the observed cache misses for them. However, as the reader can verify, these metrics are poorly correlated.

In Table 6, we show the serial run times for eight algorithms when random values of $b(v)$ are used. We observe that all four schemes of $b$-SUITOR are considerably faster than the other algorithms, $b$-SUITOR with the partial sorting schemes ST EP and ST DP being roughly twice as fast as their unsorted variants. The LD algorithm with partially sorted adjacency lists follows the $b$-SUITOR algorithms with their performances within a factor of 5 of the latter. The PGA and the GREEDY algorithms are slower than ST DP by a factor of 13 to 15. Finally, as expected from the previous results, LD U is the slowest by a factor of 35 over the best variant of $b$-SUITOR. Note that the run times for a problem and algorithm pair do not correlate well with the run times of the same pair with $b = 10$. This shows how the values of $b(v)$ influence the run times.

Finally, we study the effect of the order in which the vertices are processed for matching in $b$-SUITOR algorithms. We present the impact of the heaviest edge and pivot ordering on the number of proposals extended in the course of the algorithm in Figure 4. We find that both these ordering schemes reduce the number of proposals

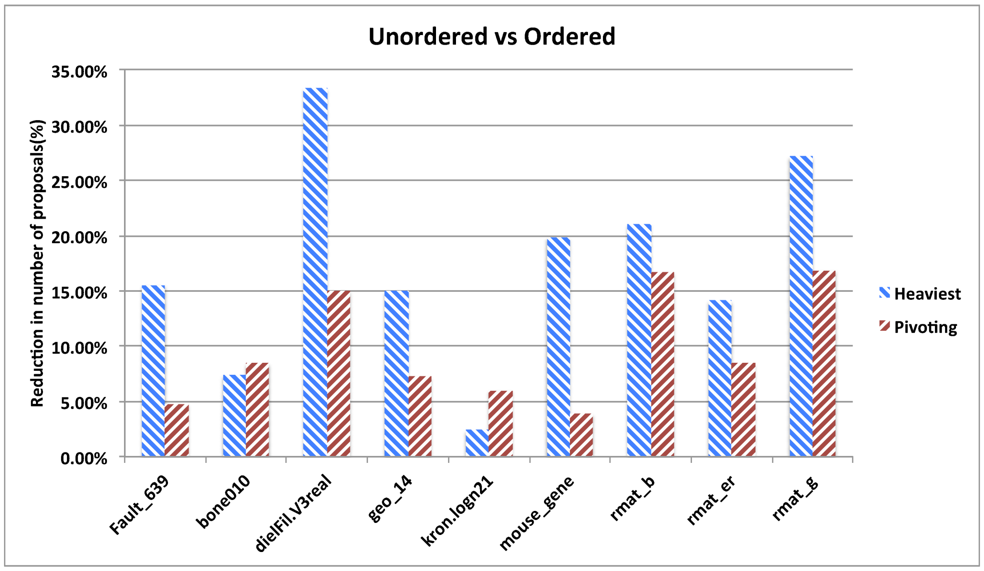


FIG. 4. *Change in the number of proposals with respect to unordered processing of vertices.*

and report the results as a percentage reduction over the variant with no reordering (i.e., the original vertex ordering in the problem is used). The heaviest edge reordering reduces the number of proposals by up to 32%, whereas the pivot ordering reduces this number up to 17% (Figure 4). Since fewer proposals translates to less work, one would expect the run time to improve for $b$-SUITOR. But the ordering itself adds an overhead cost, i.e., the additional cost of sorting the vertices by this key. If the reduction of run time due to fewer edge traversals cannot compensate for the reordering overhead, then the run time will increase. We observe that for smaller graphs where the amount of work is low, the reduced work load cannot compensate for the reordering overhead, and so the reordering led to increased run times (up to 6%). On the other hand, the run time reduces for larger graphs but by at most 12%. We conjecture that the reordering of the vertices will be more important for distributed-memory parallel algorithms, where reducing the number of proposals could lead to fewer proposals having to be communicated across the interconnection network.

The impact of various schemes on the MATCHING and the $b$-MATCHING problems is summarized in Table 7. For MATCHING, the full/partial sorting scheme reduces the number of edge traversals, but since the work per vertex in MATCHING is small, the impact on run times is also small due to the sorting overhead. Similarly, delayed updating reduces the cache misses significantly for MATCHING, but the queue management overhead for delayed updates reduces the impact on run times. However, both these variations have greater impact on run times for $b$-MATCHING. The ordering of vertices impacts the performance of both MATCHING and $b$-MATCHING algorithms in terms of the numbers of proposals, but due to the overhead cost of computing the vertex ordering, we do not see greatly reduced runtimes for our multithreaded algorithms.

**5. Parallel performance.** Among the algorithms we have considered so far, LD and $b$-SUITOR are the only algorithms with sufficient concurrency that makes

TABLE 7
*Impact of variant algorithms on the performance of* MATCHING *and* $b$-MATCHING.

| Variation | 1-MATCHING | $b$-MATCHING |
|---|---|---|
| Sorting | Minimal impact | Reduces edge traversals |
| Updating | Minimal impact | Reduces cache misses |
| Ordering | Fewer proposals | Fewer proposals |

TABLE 8
*Parallel run times (sec) with* 16 *cores on Intel Xeon processor.*

| Problem | LD U | LD P | ST EU | ST EP | ST DU | ST DP |
|---|---|---|---|---|---|---|
| Fault_639 | 1.22 | 1.35 | **0.13** | 0.14 | 0.15 | 0.21 |
| bone010 | 13.14 | 12.11 | 0.25 | **0.19** | 0.27 | 0.29 |
| dielFil.V3real | 4.54 | 4.74 | 0.43 | 0.51 | 0.42 | 0.37 |
| geo_14 | 35.77 | 0.68 | 1.92 | 0.88 | 1.58 | **0.34** |
| kron.logn21 | 82.36 | 56.68 | 1.93 | 0.59 | 1.80 | **0.55** |
| mouse_gene | 10.47 | 1.85 | 0.31 | 0.20 | 0.29 | **0.19** |
| rmat_b | 69.47 | 31.24 | 2.03 | 0.62 | 1.83 | **0.61** |
| rmat_er | 6.72 | 4.04 | 1.51 | 0.79 | 1.26 | **0.72** |
| rmat_g | 17.10 | 9.46 | 1.61 | 0.76 | 1.35 | **0.73** |
| Avg Rel Perf | 34.66 | 14.40 | 1.89 | 1.09 | 1.78 | **1.00** |

them suitable for parallelization. We parallelized all variants of these two algorithms but the completely sorted variants are not competitive in terms of run times, and thus we report results for the unsorted and partially sorted variants. Table 8 shows the parallel performance of these algorithms on 16 cores of the Intel Xeon processor. Note that the $b$-SUITOR and LD algorithms separate quite nicely into two groups, with the former performing better by large factors. Note that the partial sorting of the adjacency lists of the vertices can be performed concurrently on multiple threads, and thus the relative overhead they add to the algorithms is less significant. Hence among the $b$-SUITOR algorithms, ST DP is the fastest closely followed by ST EP, and among the LD algorithms, LD P is the fastest. Each variant of $b$-SUITOR is clearly faster than the variants of the LD algorithms by factors as large as 35 on the average.

Next, we compare the speed-up achieved by the best performing $b$-SUITOR and LD algorithms on 16 cores of the Intel Xeon in Figure 5. Due to the significant differences in performance, we plot the $y$-axis in log scale. That is, we calculate the speed-up and then take the logarithm (to base two) of that speed up. The baseline serial algorithm here is the LD algorithm with partial sorting of adjacency lists. This is why we see speed-ups larger than $\log_2(16) = 4$ for the $b$-SUITOR DP variant algorithm. We observe that the LD partial sorted variant does not scale well in general and it achieves only 8× speed up except for the problem *geo*_14, which is a denser problem.

**5.1. Optimizations of the parallel *b*-SUITOR algorithm.** We optimized our implementation of the $b$-SUITOR algorithm to take advantage of the features of modern microprocessor architectures on the Intel Xeon. Our optimizations include aligned memory allocation to increase cache line usage [18] and software prefetch instructions (using `_mm_prefetch` intrinsics) to ensure that data resides in nearby caches when it is accessed. We use dynamic scheduling to improve load balancing among the threads and the chunk size of 256 gave the best performance.

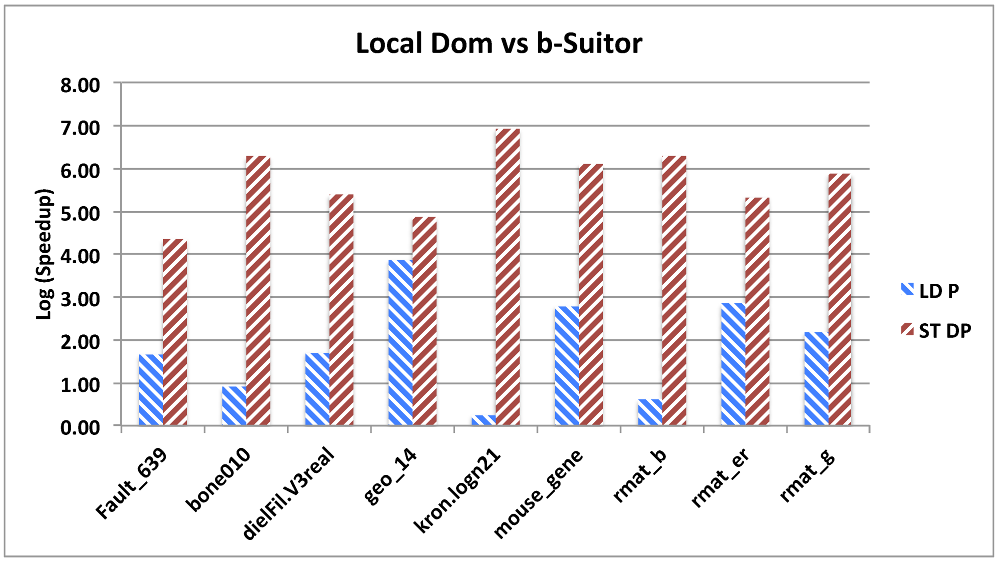


FIG. 5. *Comparing (log) speed up on* 16 *cores of Intel Xeon processor between the best performing LD and* $b$-SUITOR *variants.*

In $b$-SUITOR, threads access various shared data structures. Threads need to get exclusive access to vertex data when they update the heap; we implement spinlocks using lightweight `__sync_fetch_and_add` atomics instead of traditional locks to reduce overheads. This locking scheme is specific to the Intel architecture but any atomic instruction should work. Further, we observe that each thread inserts vertices into the queue $Q$ of vertices whose proposals are annulled, requiring frequent locking of the data structure. To reduce this overhead, each thread $t$ maintains a small private queue $Q_t$ that is sized to be cache-resident and only writes to it. Whenever $Q_t$ is full, the thread $t$ locks $Q$ once and moves all vertices from $Q_t$ to $Q$, thus reducing the total number of locks. We experimented with different sizes for the thread-private queues, and a size of 512 performed the best for the Intel Xeon, and a size of 256 was best for the Intel Xeon Phi.

We demonstrate the impact of these optimization techniques in Figure 6. We take the naive $b$-SUITOR implementation as the baseline for the comparison. The figure shows the impact of our optimizations normalized to the baseline implementation, set to one. We observe that dynamic scheduling improves the performance by up to 21% (with geometric mean of 8%). Incorporating private queues per thread gave an additional 10% (geometric mean 4.7%) performance improvement. Spin locks and heap optimizations contribute an additional 28% (geometric mean 13.8%) and 30% (geometric mean 10%). When combined with prefetching and other techniques, the overall performance improvement of the optimized code is a factor of 2.1 (geometric mean 1.9).

**5.2. The Intel Xeon Phi processor.** We optimized our codes for the Xeon Phi as on the Xeon processor and additionally used several intrinsics to take advantage of the SIMD feature on the Phi. These include operations to initialize values such as the locks and $b$-values.

We have seen that the *delayed partial* $b$-SUITOR is the best performing algorithm

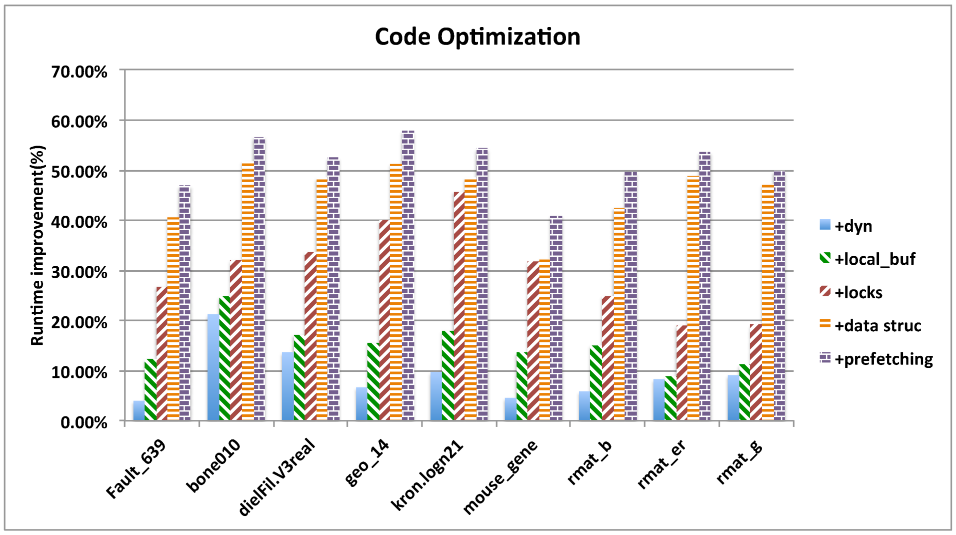


FIG. 6. *Improvements in performance from architectural optimizations on the Intel Xeon.*

both in the serial and the shared memory context. So now we focus on the parallel performance of the ST DP algorithm when the values of $b(v)$ vary, on the Intel Xeon Phi. In Figure 7, we observe that scalability increases with the increase of $b(v)$ because there is more work per node (as well as per thread). We see that ST DP scales up linearly to 54× on the Intel Xeon Phi with 60 cores. (For the Xeon, the figure is 14× for 16 cores.)

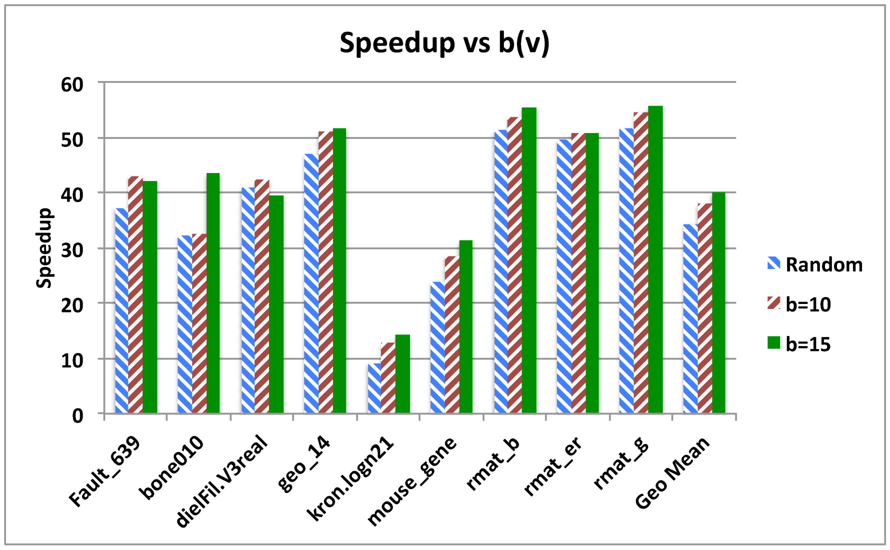


FIG. 7. *Strong scaling of* $b$-SUITOR *(delayed partial scheme) on* 60 *cores of the Intel Xeon Phi coprocessor.*

We also compare the parallel run times on 16 cores of the Intel Xeon with 60 cores of the Intel Xeon Phi in Figure 8. Xeon Phi has 4× as many cores as the Xeon, but each Phi core is roughly 2.5× slower than the Xeon. With this configuration, the Xeon Phi is up to 2.3× faster than Xeon, and this is fairly representative of the performance observed on other algorithms [34].

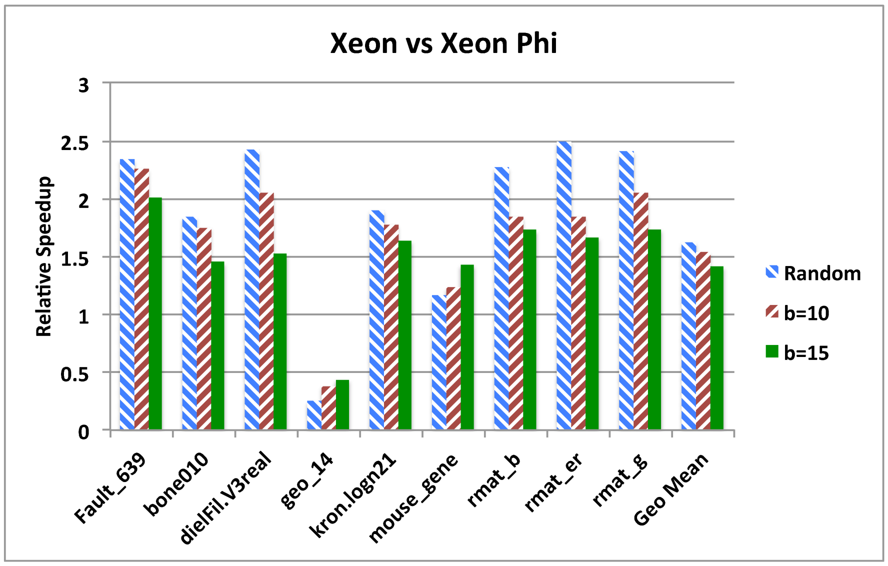


FIG. 8. *Performance improvement on Intel Xeon Phi over Intel Xeon processor.*

**6. Conclusions.** We have designed and implemented several variants of a new half-approximation algorithm for computing maximum weighted $b$-MATCHINGS in graphs, called $b$-SUITOR. The algorithm computes the same matching as that would be obtained by the GREEDY as well the as the LD algorithm. We show that $b$-SUITOR computes matchings with weights that are comparable to or better by a few percent than the PGA that uses dynamic programming to improve the weight. The $b$-SUITOR algorithm traverses fewer edges and has more cache hits than the LD algorithm and has more concurrency relative to the latter algorithm. Hence the $b$-SUITOR algorithm outperforms it by a factor of 14 (geometric mean) with respect to run times on 16 cores of an Intel Xeon multiprocessor. Strong scaling is also observed on the Intel Xeon Phi coprocessor, and speed-ups of 50 are seen on 60 (slower) cores relative to the time on one core (a factor of three greater clock frequency) of a Xeon processor. We believe that this approach could yield effective distributed-memory $b$-MATCHING algorithms as well. In related work, we have used the $b$-SUITOR algorithm and a $b$-EDGE COVER algorithm to solve a problem in data privacy [4].

## REFERENCES

[1] R. P. ANSTEE, *A polynomial algorithm for b-matching: An alternative approach*, Inform. Process. Lett., 24 (1987), pp. 153–157.

[2] D. AVIS, *A survey of heuristics for the weighted matching problem*, Networks, 13 (1983), pp. 475–493, http://dx.doi.org/10.1002/net.3230130404.

[3] K. M. CHOROMANSKI, T. JEBARA, AND K. TANG, *Adaptive anonymity via b-matching*, in Advances in Neural Information Processing Systems, MIT Press, Cambridge, MA, 2013, pp. 3192–3200.

[4] K. M. CHOROMANSKI, A. KHAN, A. POTHEN, AND T. JEBARA, *Large Scale Robust Adaptive Anonymity via b-Edge Cover and Parallel Computation*, Preprint, 2015.

[5] T. A. DAVIS AND Y. HU, *The University of Florida Sparse Matrix Collection*, ACM Trans. Math. Software, 38 (2011), pp. 1:1–1:25.

[6] G. De Francisci Morales, A. Gionis, and M. Sozio, *Social content matching in Mapreduce*, Proc. VLDB Endowment, 4 (2011), pp. 460–469.

[7] U. Derigs and A. Metz, *On the use of optimal fractional matchings for solving the (integer) matching problem*, Computing, 36 (1986), pp. 263–270.

[8] D. E. Drake and S. Hougardy, *A simple approximation algorithm for the weighted matching problem*, Inform. Process. Lett., 85 (2003), pp. 211–213.

[9] R. Duan and S. Pettie, *Linear-time approximation for maximum weight matching*, J. ACM, 61 (2014), pp. 1–23, http://dx.doi.org/10.1145/2529989.

[10] J. Edmonds, *Maximum matching and a polyhedron with* 0,1*-vertices*, J. Res. National Bureau of Standards B, 69 (1965), pp. 125–130.

[11] B. Fagginger Auer and R. Bisseling, *Graph coarsening and clustering on the GPU*, Contemp. Math., 588 (2013).

[12] H. N. Gabow, *An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems*, in Proceedings of the 15th Annual ACM Symposium on the Theory of Computing, 1983, pp. 448–456.

[13] G. Georgiadis and M. Papatriantafilou, *Overlays with preferences: Distributed, adaptive approximation algorithms for matching with preference lists*, Algorithms, 6 (2013), pp. 824–856.

[14] M. Grötschel and O. Holland, *Solving matching problems with linear programming*, Math. Program., 33 (1985), pp. 243–259.

[15] M. Halappanavar, J. Feo, O. Villa, F. Dobrian, and A. Pothen, *Approximate weighted matching on emerging many-core and multithreaded architectures*, Internat. J. High Performance Comput. Appl., 26 (2012), pp. 413–430.

[16] S. Hougardy, *Linear time approximation algorithms for degree constrained subgraph problems*, in Research Trends in Combinatorial Optimization, W. J. Cook, L. Lovasz, and J. Vygen, eds., Springer-Verlag, Berlin, 2009, pp. 185–200.

[17] B. C. Huang and T. Jebara, *Fast b-matching via sufficient selection belief propagation*, in Proceedings of International Conference on Artificial Intelligence and Statistics, 2011, pp. 361–369.

[18] *Memory Management for Optimal Performance on Intel Xeon Phi Coprocessor*. https://software.intel.com/en-us/articles/memory-management-for-optimal-performance-on-intel-xeon-phi-coprocessor-alignment-and (2015).

[19] T. Jebara and V. Shchogolev, *b-matching for spectral clustering*, in European Conference on Machine Learning, Springer, New York, 2006, pp. 679–686.

[20] T. Jebara, J. Wang, and S.-F. Chang, *Graph construction and b-matching for semi-supervised learning*, in Proceedings of the 26th Annual International Conference on Machine Learning, ACM, 2009, pp. 441–448.

[21] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392, http://dx.doi.org/10.1137/S1064827595287997.

[22] C. Koufogiannakis and N. E. Young, *Distributed algorithms for covering, packing and maximum weighted matching*, Distributed Comput., 24 (2011), pp. 45–63.

[23] F. Manne and R. H. Bisseling, *A parallel approximation algorithm for the weighted maximum matching problem*, in Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics, 2007, pp. 708–717.

[24] F. Manne and M. Halappanavar, *New effective multithreaded matching algorithms*, in IEEE 28th International Symposium on Parallel and Distributed Processing Symposium, 2014, pp. 519–528, http://dx.doi.org/10.1109/IPDPS.2014.61.

[25] F. M. Manshadi, B. Awerbuch, R. Gemulla, R. Khandekar, J. Mestre, and M. Sozio, *A distributed algorithm for large-scale generalized matching*, Proc. VLDB Endowment, 6 (2013), pp. 613–624.

[26] A. B. Marsh III, *Matching Algorithms*, Ph.D. thesis, John Hopkins University, Baltimore, MD, 1979.

[27] J. Maue and P. Sanders, *Engineering algorithms for approximate weighted matching*, in Experimental Algorithms, Springer, New York, 2007, pp. 242–255.

[28] J. Mestre, *Greedy in approximation algorithms*, in Algorithms–ESA 2006, Springer, New York, 2006, pp. 528–539.

[29] D. L. Miller and J. F. Pekny, *A staged primal-dual algorithm for perfect b-matching with edge capacities*, ORSA J. Comput., 7 (1995), pp. 298–320.

[30] M. Müller-Hannemann and A. Schwartz, *Implementing weighted b-matching algorithms: Insights from a computational study*, J. Exp. Algorithmics, 5 (2000).

[31] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, *Introducing the Graph* 500, Cray User's Group, 2010.

[32] M. Naim, F. Manne, M. Halappanavar, A. Tumeo, and J. Langguth, *Optimizing approximate weighted matching on Nvidia Kepler* K40, in Proceedings of the 22nd Annual IEEE International Conference on High Performance Computing (HiPC), Bengaluru, India, 2015.

[33] M. Padberg and M. R. Rao, *Odd minimum cut-sets and $b$-matchings*, Math. Oper. Res., 7 (1982), pp. 67–80.

[34] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey, *Pardicle: Parallel Approximate Density-based Clustering*, in Proceedings of Supercomputing (SC14), 2014, pp. 560–571.

[35] M. Penrose, *Random Geometric Graphs*, vol. 5, Oxford University Press, Oxford, UK, 2003.

[36] R. Preis, *Linear time* 1/2*-approximation algorithm for maximum weighted matching in general graphs*, in Proceedings of the of the 16th Annual Conference on Theoretical Aspects of Computer Science, STACS 99, Springer, New York, 1998, pp. 259–269.

[37] W. R. Pulleyblank, *Faces of Matching Polyhedra*, Ph.D. thesis, Faculty of Mathematics, University of Waterloo, 1973.

[38] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, *Parallel community detection for massive graphs*, in Parallel Processing and Applied Mathematics, Springer, New York, 2012, pp. 286–296.

[39] A. Schrijver, *Combinatorial Optimization—Polyhedra and Efficiency. Volume A: Paths, Flows, Matchings*, Springer, New York, 2003.

[40] A. Tamir and J. S. B. Mitchell, *A maximum $b$-matching problem arising from median location models with applications to the roommates problem*, Math. Program., 80 (1995), pp. 171–194.