Computing Maximum Cardinality Matchings in Parallel on Bipartite Graphs via Tree-Grafting

Ariful Azad, Aydın Buluç, and Alex Pothen

Abstract—It is difficult to obtain high performance when computing matchings on parallel processors because matching algorithms explicitly or implicitly search for paths in the graph, and when these paths become long, there is little concurrency. In spite of this limitation, we present a new algorithm and its shared-memory parallelization that achieves good performance and scalability in computing maximum cardinality matchings in bipartite graphs. Our algorithm searches for augmenting paths via specialized breadth-first searches (BFS) from multiple source vertices, hence creating more parallelism than single source algorithms. Algorithms that employ multiple-source searches cannot discard a search tree once no augmenting path is discovered from the tree, unlike algorithms that rely on single-source searches. We describe a novel tree-grafting method that eliminates most of the redundant edge traversals resulting from this property of multiple-source searches. We also employ the recent direction-optimizing BFS algorithm as a subroutine to discover augmenting paths faster. Our algorithm compares favorably with the current best algorithms in terms of the number of edges traversed, the average augmenting path length, and the number of iterations. We provide a proof of correctness for our algorithm. Our NUMA-aware implementation is scalable to 80 threads of an Intel multiprocessor and to 240 threads on an Intel Knights Corner coprocessor. On average, our parallel algorithm runs an order of magnitude faster than the fastest algorithms available. The performance improvement is more significant on graphs with small matching number.

Index Terms—Cardinality matching, bipartite graph, tree grafting, parallel algorithms.

1 Introduction

We design and implement a parallel algorithm for computing maximum cardinality matchings in bipartite graphs on shared memory parallel processors. Approximation algorithms are employed to create parallelism in matching problems, but a matching of maximum cardinality is needed in several applications. One application in scientific computing is to permute a matrix to its block triangular form (BTF) via the Dulmage-Mendelsohn decomposition of bipartite graphs [1]. Once the BTF is obtained, in circuit simulations, sparse linear systems of equations can be solved faster [2], and data structures for sparse orthogonal factors for least-squares problems can be correctly predicted [3].

Matching algorithms that achieve high performance on modern multiprocessors are challenging to design and implement because they either rely on searching explicitly for long paths or implicitly transmit information along long paths in a graph. In earlier work, effective parallel matching algorithms have been designed and implemented on shared memory multiprocessors, especially multi-threaded machines [4], [5]. These algorithms achieve parallelism by using multi-source graph searches, i.e., by searching with many threads from multiple (unmatched) vertices for augmenting paths (paths in the graph that alternate between matched and unmatched edges with unmatched vertices as endpoints), and using these augmenting paths to increase the cardinality of the matching.

A. Azad and A. Buluç are with the Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720. E-mail: azad@lbl.gov, abuluc@lbl.gov.
A. Pothen is with the department of Computer Science, Purdue University, West Lafayette, IN 47907. E-mail: apothen@purdue.edu

Algorithms based on multi-source graph searches (i.e., multi-source or MS algorithms) have a significant weakness relative to algorithms based on single-source graph searches (i.e., single-source or SS algorithms). When an SS algorithm fails to match a vertex u, it will not match u at any future step, so it can remove u (and other vertices in its search tree) from further consideration. Setubal [6] and Duff *et al.* [7] used this property to implement SS algorithms, although they did not provide a formal proof. A restricted version of this property of SS algorithms that applies only to the starting vertices of graph searches was proved in Theorem 10.5 by Papadimitriou and Steiglitz [8]. In this paper, we prove a stronger version of this property that applies to *all the vertices visited* during an unsuccessful search. However, the search trees in MS algorithms are constrained to be vertex-disjoint to allow concurrent augmentations along multiple augmenting paths [9], [7]. Thus, even if the algorithm fails to find an augmenting path from an unmatched vertex u at some step, there could be an augmenting path from u at a future step since some vertices that could be included in the search tree rooted at u at this step might have been included in some other search tree. Hence, MS algorithms cannot discard search trees from which we do not discover augmenting paths and have to reconstruct them many times. This property limits the scaling of MS algorithms, especially on graphs where the size of the maximum matching is small compared to the number of vertices.

We address this limitation of MS algorithms by reusing the trees constructed in one phase in the next phase. We graft a part of a search tree that yields an augmenting path onto another search tree from which we have not found an augmenting path. If the search succeeds in finding an augmenting path in the grafted tree, we reuse parts of this

tree in subsequent grafting operations. Otherwise, we keep the tree intact with the hope of discovering an augmenting path in future. In both cases, we have avoided the work of constructing this search tree from scratch, at the cost of the work associated with the grafting operation. In addition to tree grafting, we have integrated the direction optimization idea [10] to the specialized breadth-first-searches (BFS) of our algorithm. We demonstrate that the new serial algorithm computes maximum cardinality matchings an order of magnitude faster than the current best-performing algorithms on several classes of graphs. Even faster performance is obtained by the parallel grafting algorithm on multithreaded shared memory multiprocessors.

Our main contributions in this paper are as follows:

- We theoretically and experimentally characterize the properties of existing SS and MS algorithms for maximum cardinality matching. With a theorem, we prove the advantage of SS algorithms on certain graphs and motivate the changes in MS algorithms needed to make them faster on those graphs.
- We present a novel tree-grafting method that eliminates most of the redundant edge traversals of MS matching algorithms, and prove the correctness of our algorithm.
- We employ the recently developed direction-optimized BFS [10] to speed up augmenting path discoveries.
- We provide a NUMA-aware multithreaded implementation that attains up to 15x speedup on a two-socket node with 24 cores. The algorithm yields better search rates than its competitors, and is less sensitive to performance variability of multithreaded platforms.
- On average, our algorithm runs 7x faster than current best parallel algorithm on a 40-core Intel multiprocessor. On graphs where the maximum matching leaves a number of vertices unmatched, our algorithm runs 10x and 27x faster than two state-of-the-art implementations.
- We provide an efficient implementation of our algorithm on Intel's Many Integrated Core (MIC) architecture. The newly developed algorithm attains up to 36x speedup on an Intel Knights Corner coprocessor using 60 cores, which demonstrates the utility of our algorithm on future manycore architectures.

2 Existing Algorithms for Maximum Matching in Bipartite Graphs

2.1 Background and Notations

Given a graph $G=(V, E)$ on the set of vertices V and edges E, a *matching* M is a subset of edges such that at most one edge in M is incident on each vertex in V. The number of edges in M is called the cardinality $|M|$ of the matching. A matching M is *maximal* if there is no other matching M' that properly contains M. M is a *maximum* cardinality matching if $|M| \geq |M'|$ for every matching M'. M is a *perfect* matching if every vertex of V is matched. The cardinality of the maximum matching is the *matching number* of the graph. In this paper, we report the matching number as a fraction of the number of vertices. We denote $|V|$ by n and $|E|$ by m.

This paper focuses on matchings in a bipartite graph, $G=(X \cup Y, E)$, where the vertex set V is partitioned into two disjoint sets such that every edge connects a vertex in X to a vertex in Y. Given a matching M in a bipartite graph G, an edge is matched if it belongs to M, and unmatched otherwise. Similarly, a vertex is matched if it is an endpoint of a matched edge, and unmatched otherwise. If x in X is matched to y in Y, we call x is the *mate* of y and write $x=mate[y]$ and $y=mate[x]$. An *M-alternating path* in G with respect to a matching M is a path whose edges are alternately matched and unmatched. An *M-augmenting path* is an M-alternating path which begins and ends with unmatched vertices. By exchanging the matched and unmatched edges on an M-augmenting path P, we can increase the cardinality of the matching M by one (this is equivalent to the symmetric difference of M and P, $M \oplus P = (M \setminus P) \cup (P \setminus M)$). Given a set of vertex disjoint M-augmenting paths $\mathbb{P}$, $M' = M \oplus \mathbb{P}$ is a matching with cardinality $|M|+|\mathbb{P}|$.

2.2 Classes of cardinality matching algorithms

Maximal matching algorithms compute a matching with cardinality at least half of the maximum matching. For many input graphs, maximal matching algorithms find all or a large fraction of the maximum cardinality matching [11]. Since maximal matching algorithms can be implemented in $O(m)$ time, which is much faster than maximum matching algorithms, the former algorithms are often used to initialize the latter.

Maximum matching algorithms are broadly classified into three groups: (1) augmenting-path based, (2) push-relabel based [5], [12], and (3) auction based [13]. This paper is primarily focused on the augmenting-path based algorithms. An augmenting-path based matching algorithm runs in several *phases*, each of which searches for augmenting paths in the graph with respect to the current matching M and augments M by the augmenting paths. The algorithm finds a maximum matching M when there is no M-augmenting path in the graph [14]. Augmenting path based algorithms primarily differ from one another based on the search strategies used to find augmenting paths. The search can be performed form one unmatched vertex (SS algorithms) or from all unmatched vertices simultaneously (MS algorithms). The search can be performed by using the BFS, depth-first search (DFS), or a combination of both BFS and DFS (the Hopcroft-Karp algorithm [9]). Table 1 summarizes the major classes of cardinality matching algorithms relevant to the discussion in this paper. For more details, we refer the reader to a book on matching algorithms [15].

In this paper, without loss of generality, we search for augmenting paths from unmatched X vertices, one vertex set of a bipartite graph. The graph searches for augmenting paths have a structure different from the usual graph searches: the only vertex reachable from a matched vertex y in Y is its unique *mate*. The search for all neighbors continues from the mate, which is again a vertex in X. We call the search trees constructed in a phase of the algorithms as *alternating search trees*. An alternating search tree T is

Algorithm 1 Matching algorithms based on single-source augmenting path searches. **Input:** A bipartite graph $G(X \cup Y, E)$, a matching M. **Output:** A maximum cardinality matching M.

1: **procedure** SS-MATCH($G(X \cup Y, E)$, M)
2: **for** each $y \in Y$ **do** $visited[y] \leftarrow 0$
3: **for** each unmatched vertex $x_0 \in X$ **do**
4: $P \leftarrow$SS-SEARCH($G, x_0, visited, M$) ▷ search for an augmenting path from x_0 using previously unvisited vertices. $visited[y]$ is set to 1 for every traversed vertex y in Y.
5: $Y_s \leftarrow Y$ vertices traversed in the latest search
6: **if** $P \neq \phi$ **then** ▷ An augmenting path is found
7: $M \leftarrow M \oplus P$ ▷ Increase matching by one
8: **for** each $y \in Y_s$ **do** $visited[y] \leftarrow 0$

Algorithm 2 Matching algorithm based on multi-source augmenting path searches. Input and Output same as Algorithm 1.

1: **procedure** MS-MATCH($G(X \cup Y, E)$, M)
2: **repeat**
3: **for** each $y \in Y$ **do** $visited[y] \leftarrow 0$
4: $X_0 \leftarrow$ all unmatched X vertices
5: $\mathbb{P} \leftarrow$MS-SEARCH($G, X_0, visited, M$) ▷ search for a set of vertex disjoint augmenting paths from X_0. $visited[y]$ is set to 1 for each traversed vertex y in Y.
6: $M \leftarrow M \oplus \mathbb{P}$ ▷ Increase matching by $|\mathbb{P}|$
7: **until** $\mathbb{P} = \phi$ ▷ Continue if an augmenting path is found

rooted at an unmatched vertex x, and all paths from x to other vertices in the tree are alternating paths. We denote a tree rooted at x by $T(x)$.

2.3 Single- and multi-source algorithms

The SS-MATCH (Algorithm 1) and MS-MATCH (Algorithm 2) functions describe the general structures of the SS and MS augmenting path based algorithms. They both take a bipartite graph $G(X \cup Y, E)$ and an initial matching M as input, and return a maximum cardinality matching by updating M. In each phase, SS-MATCH searches for an augmenting path from an unmatched vertex x_0 in X using the SS-SEARCH function. SS-SEARCH constructs an alternating tree $T(x_0)$ by using Y vertices whose *visited* flags are 0 and sets the *visited* flag to 1 for every Y vertex included in the current search tree. SS-SEARCH stops exploring the graph as soon as an augmenting path P is found, which is then used to augment the matching. By contrast, MS-MATCH traverses the graph from X_0, the set of all unmatched X vertices, using the MS-SEARCH function. MS-SEARCH constructs an alternating forest and returns a set of vertex disjoint augmenting paths $\mathbb{P}$, which is used to augment the matching.

There is a crucial difference between the SS and MS algorithms when we fail to augment a matching from an unmatched vertex x_0. For the SS algorithm, vertices and edges in the alternating search tree $T(x_0)$ cannot be part of any future augmenting paths. We formally prove this property of SS algorithms in Theorem 1 and Corollary 1 below, which are the extensions of Theorem 10.5 and its Corollary described by Papadimitriou and Steiglitz [8]. As a consequence, SS algorithms could remove all vertices and edges in a tree $T(x_0)$ that fails to discover any augmenting path from further consideration [6], [7]. Such a failed tree $T(x_0)$ can be easily removed from future search space by not clearing the *visited* flags of the vertices in $T(x_0)$. For example, line 8 of Algorithm 1 clears the visited flags only when an augmenting path is discovered in the current phase. By contrast, an MS algorithm cannot discard vertices from a tree $T(x_0)$ without an augmenting path, because the same tree could yield an augmenting path in future phases of the algorithm. Consequently, Algorithm 2 clears the *visited* flags of Y vertices in every phase of the algorithm.

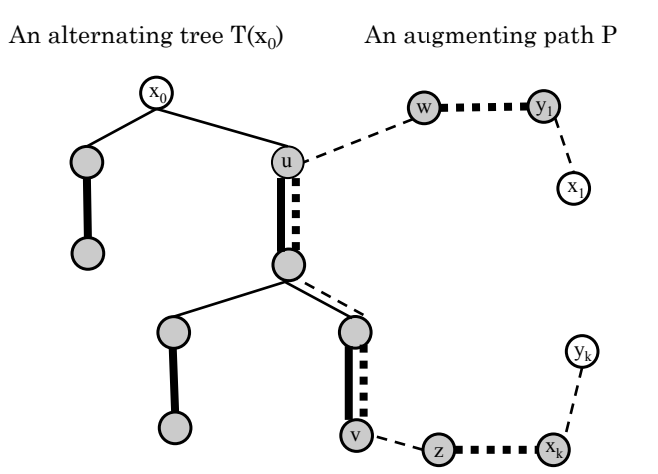

Fig. 1. Edges shown with solid lines represent an alternating tree $T(x_0)$ that does not have an augmenting path. Edges shown with broken lines represent an augmenting path $P = (x_1, y_1, ..., x_k, y_k)$. Matched and unmatched vertices are shown in filled and empty circles, respectively. Thin lines represent unmatched edges and thick lines represent matched edges. This Figure represents a plausible situation that $T(x_0)$ and P do not have an edge in common.

Theorem 1. *Suppose that in a bipartite graph $G(X \cup Y, E)$ there is no augmenting path starting from an unmatched vertex x_0 in X with respect to a matching M. Let $T(x_0)$ be an M-alternating tree rooted at x_0 identified by the single source algorithm (Algorithm 1) and P be an M-augmenting path starting from another unmatched vertex x_1 in X. Then $T(x_0)$ and P are vertex disjoint.*

Proof. Let $P = (x_1, y_1, ..., x_k, y_k)$ where x_i and y_j are vertices in X and Y, respectively. Suppose that P and $T(x_0)$ are not vertex disjoint. Let u be the first vertex on P that is also in $T(x_0)$, and v be the last vertex on P belonging to $T(x_0)$ such that all vertices between u and v are in $T(x_0)$ (see Fig. 1).

First, consider that u is an X vertex and w in Y is the vertex preceding u on the path P where w is not in $T(x_0)$. Such a vertex w always exists because the first vertex x_1 in P does not belong to $T(x_0)$. Since $T(x_0)$ yields no augmenting path, the SS algorithm explores all neighbors of X vertices in the tree. Therefore, all neighbors of u must be in $T(x_0)$, which contradicts the fact that a neighbor w of u is outside of $T(x_0)$.

Next, consider that u is a Y vertex and z is the vertex following v on the path P where z is not in $T(x_0)$. (This is the situation illustrated in Fig. 1.) Such a vertex z always exists because the last vertex y_k in P is unmatched and therefore, does not belong to $T(x_0)$. Since the alternating subpath between u and v is of odd length, v must be an X vertex. Therefore, all neighbors of v must be in $T(x_0)$,

TABLE 1
Summary of cardinality matching algorithms. Newly developed MS-BFS-Graft algorithm is shown in bold. In fine grained parallelism, each thread processes a vertex, whereas in coarse grained parallelism, each thread processes a DFS tree or a large portion of a search tree. Fat cores are latency-optimized cores (such as the CPU cores today) and thin cores are throughput optimized cores (such as Xeon Phi or GPU cores today). In the last column, references to the original algorithms are shown first, followed by their serial and parallel implementations.

Algorithm Class	Search Strategy	Acronym	Serial Complexity	Parallelization Strategy	Architecture Preference	References
Single-Source	Alternating DFS	SS-DFS	$O(nm)$	Not parallel	One fat core	[7]
	Alternating BFS	SS-BFS	$O(nm)$	Fine-grained	Many thin cores	[7]
Multi-Source	Vertex-disjoint alternating DFSs (Pothen-Fan algorithm)	PF	$O(nm)$	Coarse-grained	Few fat cores	[1], [7], [4]
	Vertex-disjoint alternating BFSs	MS-BFS	$O(nm)$	Fine-grained	Many thin cores	[4]
	Vertex-disjoint BFSs with tree grafting	**MS-BFS-Graft**	$O(nm)$	Fine-grained	Many thin cores	[16]
	Both alternating BFS and DFS (Hopcroft-Karp algorithm)	HK	$O(\sqrt{n}m)$	Both fine- and coarse-grained	Both fat and thin cores	[9], [7], [4]
Push-Relabel	Label guided FIFO search	PR	$O(nm)$ [a]	Both fine- and coarse-grained	Both fat and thin cores	[12], [17], [5]
Auction	Concurrent bidding	Auction	$O(\sqrt{n}m \log n)$	Both fine- and coarse-grained	Both fat and thin cores	[13], [18], [19]
Maximal matching	Greedy maximal matching	GM	$O(m)$	Fine-grained	Many thin cores	[11]
	Karp-Sipser algorithm	KS	$O(m)$	Fine-grained	Many thin cores	[20], [11], [4]

a. A variant of the push-relabel algorithm called the minimum distance discharge algorithm attains the asymptotic complexity of $O(\sqrt{n}m)$ for the maximum cardinality matching problem [21]. This algorithm processes *push* and *relabel* operations by non-decreasing order of vertex labels. However, Kaya *et al.* [17] found that the first-in-first-out (FIFO) ordering that has $O(nm)$ complexity performs the best on most practical problems.

which contradicts the fact that a neighbor z of v is outside of $T(x_0)$. Hence, $T(x_0)$ and P are vertex disjoint. □

Corollary 1. *If at some phase of a single source algorithm there is no augmenting path in a tree $T(x_0)$ rooted at x_0, then there will never be an augmenting path passing through any vertex in $T(x_0)$.*

Proof. Let M_0 be the matching before constructing the tree $T(x_0)$, and M_k be the matching at the beginning of the kth phase after building $T(x_0)$. Since $T(x_0)$ fails to discover any augmenting path, M_0 remains unchanged after building $T(x_0)$ (i.e., $M_1 = M_0$). Now we proceed by induction on M_k to prove the corollary.

If we discover an M_1-augmenting path P in the first phase after building $T(x_0)$, then by Theorem 1, P and $T(x_0)$ are vertex disjoint. Suppose that all augmenting paths discovered in the first k phases after building $T(x_0)$ are vertex disjoint with $T(x_0)$. That is, $T(x_0)$ remains unchanged (i.e., matched and unmatched edges in $T(x_0)$ remain matched and unmatched, respectively) in all of these possible k augmentations and therefore, every alternating path in $T(x_0)$ is M_{k+1}-alternating path in the $(k+1)$th phase. Then by Theorem 1, any augmenting path discovered in the $(k+1)$th phase is also vertex disjoint with $T(x_0)$. Therefore, by induction, there will never be an augmenting path passing through any vertex in $T(x_0)$. □

2.4 DFS- and BFS-based Algorithms

SS and MS algorithms search for augmenting paths using *alternating* DFS, BFS, or a combination of both. Alternating DFS and BFS generate search trees whose roots are always unmatched vertices and each path from the root to every other vertex in a search tree is an alternating path. Since matching algorithms always search for alternating paths, we often drop the term "alternating" in this paper. Next, we briefly describe three algorithms with theoretical and practical importance [4], [7], [22].

The *multi-source BFS (MS-BFS) algorithm* runs a level-synchronous BFS from all unmatched vertices and builds an alternating forest. At each level, the MS-BFS algorithm explores the unvisited neighbors of the current frontier (the set of vertices in the current level) and the mates of the neighbors. A tree stops growing when it finds an augmenting path, while other trees continue growing by advancing the frontier in a level-synchronous way. When the frontier becomes empty, we augment the current matching by the augmenting paths discovered in this phase and proceed to the next phase. In the worst case, the MS-BFS algorithm might need n phases to find the maximum matching, hence the $O(mn)$ bound.

The *Pothen-Fan (PF) algorithm* [1] is a multi-source DFS-based algorithm that uses DFS with *lookahead* to find a maximal set of vertex-disjoint augmenting paths. The idea of the lookahead mechanism in DFS is to search for an unmatched vertex in the adjacency list of a vertex x being searched before proceeding to continue the DFS from one of x's children. If the lookahead discovers an unmatched vertex, then we obtain an augmenting path and can terminate the DFS. From one iteration to the next, the direction in which the adjacency list is searched for unmatched vertices can be switched from the beginning of the list to the end of the list, and vice versa. Duff *et al.* [7] call this *fairness*, and found that this enhancement leads to faster execution times of the PF algorithm. The complexity of the algorithm is $O(mn)$.

The Hopcroft-Karp (HK) algorithm [9] finds a *maximal* set of *shortest* vertex-disjoint augmenting paths and augments along each path simultaneously. At each phase, the

HK algorithm employs BFS from all unmatched vertices and constructs an alternating *layered graph* by traversing the graph level by level. The BFS stops at the first level where an unmatched vertex is discovered, hence exposing only the shortest augmenting paths. Next, the DFS is used to find a maximal set of vertex-disjoint augmenting paths within this layered graph. By using the two-step searches, the number of augmentation phases can be bounded by $O(\sqrt{n})$, resulting in faster asymptotic time complexity $O(m\sqrt{n})$ [9].

2.5 Characteristics of existing algorithms relevant to parallel performance

In this subsection, we investigate three properties of augmenting path based matching algorithms, which significantly impact their parallel performance. These properties are: (a) the number of traversed edges, (b) the number of phases, and (c) average length of augmenting paths. Since a matching algorithm spends most of its time on graph searches (e.g., see Fig. 11), the first property determines its serial execution time. The number of phases is a lower bound on the number of synchronization steps needed by an algorithm because any parallel matching algorithm needs to synchronize between consecutive phases. Finally, the average length of augmenting paths determines the heights of search trees. Hence, longer augmenting paths could lead to load imbalance on higher concurrency. Furthermore, augmenting a matching by long paths where each augmentation is often performed sequentially might increase the time to augment the matching. Hence, the second and third properties influence the performance of parallel algorithms. Here, we compare these three properties of five sequential algorithms, all initialized via the Karp-Sipser algorithm: (1) single-source DFS (SS-DFS), (2) single-source BFS (SS-BFS), (3) the PF algorithm (with fairness), (4) multi-source BFS (MS-BFS), and (5) the HK algorithm. The MS-BFS implementation is taken from Azad *et al.* [4] and the rest are taken from Duff *et al.* [7]. We selected three graphs (`kkt-power`, `cit-patents`, and `wikipedia`), one from each class of graphs described in Table 3.

Number of traversed edges (Fig. 2(a)): PF and MS-BFS algorithms traverse fewer edges than HK algorithm in spite of the latter's superior asymptotic time complexity [4], [7]. For `kkt-power` (with a perfect matching), MS algorithms perform significantly better than SS algorithms with PF traversing the fewest edges. However, for graphs with smaller matching numbers (`cit-patents`, and `wikipedia`), SS-BFS traverses the fewest number of edges.

Number of phases (Fig. 2(b)): MS algorithms could identify multiple augmenting paths in a single phase; hence they need fewer number of phases than the SS algorithms. When computing matchings on graphs with small diameters, the PF algorithm might require more phases than MS-BFS because the former usually finds longer augmenting paths instead of several short augmenting paths. Despite the superior bound of $O(\sqrt{n})$ on the number of phases, the HK algorithm requires more phases than MS-BFS since the former discovers only the shortest augmenting paths.

Augmenting path lengths (Fig. 2(c)): BFS-based algorithms find shorter augmenting paths than DFS-based algorithms. The HK algorithm discovers the shortest augmenting paths. MS algorithms usually discover shorter augmenting paths than the SS algorithms, and the difference is more dramatic for the DFS based algorithms.

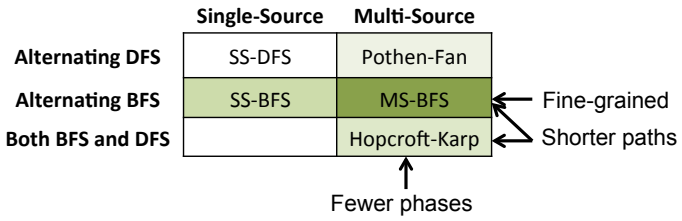

Fig. 3. Different algorithms expose different features favorable to parallelization. Darker shades are used for algorithms with more features.

Theoretical vs. practical performance of matching algorithms: The Hopcroft-Karp algorithm has the best asymptotic complexity of $O(\sqrt{n}m)$ where $O(\sqrt{n})$ is the bound on the number of phases, and each phase requires time $O(m)$. Other maximum matching algorithms described in Table 1 have higher asymptotic complexity of $O(mn)$ because in the worst case, they might require $O(n)$ phases. However, empirical results suggest that the worst case running time seems to be an over-pessimistic estimation of the actual running time in practice [4], [7]. On Erdős-Rényi random graphs, Motwani [23] and Bast *et al.* [24] proved that with high probability every non-maximum matching has an augmenting path of length $O(\log n)$. In the HK algorithm, the length of the shortest augmenting path strictly increases from one phase to the next and thus a bound $O(\log n)$ on the maximal length of shortest augmenting paths implies a bound on the number of phases. Empirical results on most practical (non-random) graphs also indicate the existence of short augmenting paths in any phase of cardinality matching. For example, Fig. 2 (c) demonstrates that the average length of augmenting paths can be less than ten for different classes of graphs with several millions of vertices, which directly influences the number of phases (Fig. 2 (b)) and the overall runtime of an algorithm.

Practical considerations in parallel algorithms: Based on our preliminary experiments, Fig. 3 summarizes different favorable features present in different algorithms. MS algorithms are more scalable because of increased concurrency and decreased synchronization between consecutive phases. In contrast to the PF algorithm that employs coarse grained parallelism [4], the BFS-based algorithm can employ fine-grained parallelism with each thread processing one vertex in a level of a BFS tree. Based on these considerations, MS-BFS promises to be the most scalable algorithm on highly parallel environments.

However, for some graphs, especially those with small matching numbers (e.g., `cit-patents` and `wikipedia` in Fig. 2), SS-BFS traverses fewer edges than MS-BFS. As discussed earlier, single source algorithms can reduce the search space in later phases of the algorithm by removing search trees yielding no augmenting paths. For example, `cit-patents` has 75,982 unmatched vertices with respect to the maximum matching (see Table 3). For `cit-patents`, an SS algorithm could discard 75,982 trees at different phases of the algorithm, whereas the MS-BFS algorithm could build these trees many times. Therefore, when run sequentially, SS-BFS is expected to run faster than MS-

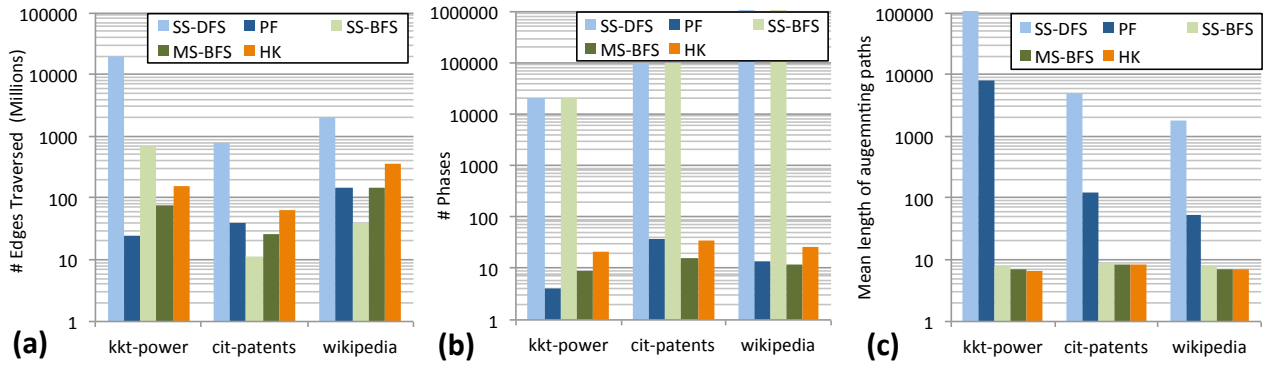

Fig. 2. (a) The number of traversed edges, (b) the number of phases, and (c) the average length of augmenting paths for five maximum matching algorithms.

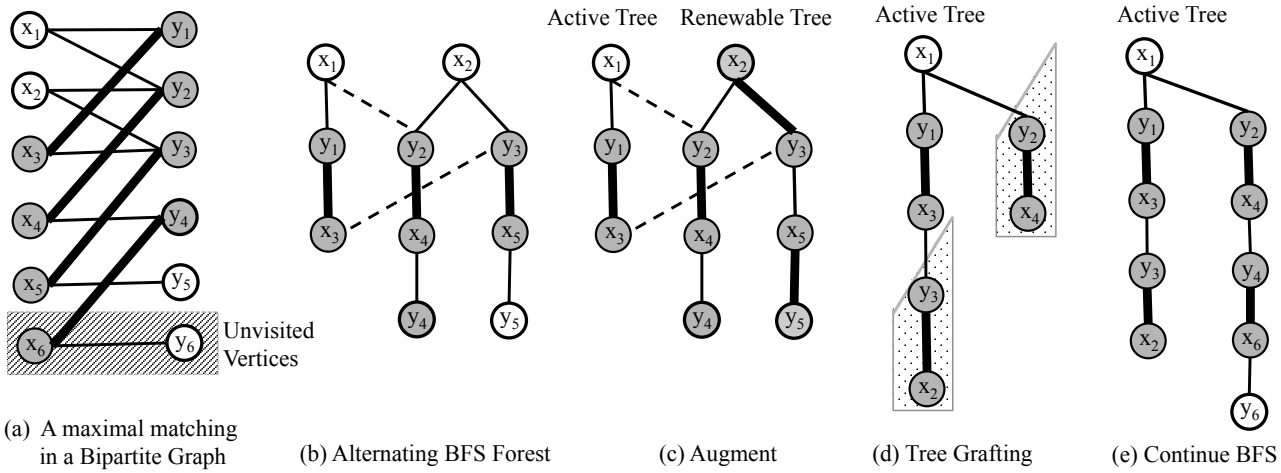

Fig. 4. (a) A maximal matching in a bipartite graph. Matched and unmatched vertices are shown in filled and empty circles, respectively. Thin lines represent unmatched edges and thick lines represent matched edges. (b) A BFS forest with two trees $T(x_1)$ and $T(x_2)$ created by the MS-BFS algorithm. The edges (x_1, y_2) and (x_3, y_3) (shown with broken lines) are scanned but not included in $T(x_1)$ to keep the trees vertex-disjoint. Unvisited vertices shown in Subfig. (a) did not take part in the current BFS traversal. (c) The current matching is augmented by the augmenting path (x_2, y_3, x_5, y_5). $T(x_1)$ remains active since no augmenting path is found in it, while $T(x_2)$ becomes a renewable tree. (d) Vertices y_2 and y_3 along with their mates are grafted onto $T(x_1)$. The vertices x_2 and x_4 form the new frontier. (e) BFS proceeds from the new frontier and finds an augmenting path in $T(x_1)$.

BFS on a graph with small matching number. We address this limitation of the MS-BFS algorithm with a tree-grafting mechanism that reduces the repetition of work across multiple phases in MS algorithms. This newly developed algorithm called MS-BFS-Graft has the advantage of both SS and MS algorithms and demonstrates better serial and parallel performance than other existing algorithms.

3 MS-BFS ALGORITHM WITH TREE GRAFTING

3.1 Intuition behind the algorithm

Consider a maximal matching in a bipartite graph shown in Fig. 4(a). The MS-BFS algorithm traverses the graph from unmatched X vertices x_1 and x_2 and creates two vertex-disjoint alternating trees $T(x_1)$ and $T(x_2)$. The trees are shown in Fig. 4(b) where the edges (x_1, y_2) and (x_3, y_3) (shown with broken lines) are scanned but not included in $T(x_1)$ to maintain the vertex-disjointedness property. In Fig. 4(b), $T(x_1)$ stops growing because its last frontier $\{x_3\}$ does not have any unvisited neighbors. On the other hand, $T(x_2)$ stops growing as soon as an augmenting path (x_2, y_3, x_5, y_5) is found. Next, we augment the current matching with (x_2, y_3, x_5, y_5) as shown in Fig. 4(c), which finishes the current phase. After augmentation, existing MS algorithms (e.g., the PF algorithm) destroy both $T(x_1)$ and $T(x_2)$ and start the next phase from the remaining unmatched vertex x_1. Notice that the whole tree $T(x_1)$ must be grown again along with the edges (x_1, y_2) and (x_3, y_3) before we can explore the rest of the graph for an augmenting path. An alternative approach is to keep $T(x_1)$ intact, *graft* relevant edges ((x_1, y_2) and (x_3, y_3)) onto $T(x_1)$, and then continue the next phase with the grafted tree $T(x_1)$. We call this process "tree grafting".

In this context, we call $T(x_1)$ (a tree where no augmenting path is found) an *active tree* and $T(x_2)$ (a tree where an augmenting path is found) a *renewable tree*. Additionally, in Fig. 4(a), x_6 and y_6 are *unvisited vertices* in the current phase. At the end of a phase, we graft a Y vertex y_j from a renewable tree onto an X vertex x_i in an active tree if (x_i, y_j) is an edge in the graph. In Fig. 4(c), y_2 and y_3 from the renewable tree have edges to x_1 and x_3 in the active tree. Therefore, y_2 and y_3 along with their mates are grafted onto $T(x_1)$ as shown in Fig. 4(d). The rest of $T(x_2)$ are destroyed (by clearing parent pointers, visited flags, etc.). The next phase of the algorithm begins with the frontier $\{x_2, x_4\}$ obtained after the tree-grafting step (Fig. 4(d)) and continues growing $T(x_1)$. Fig. 4(e) shows the next phase where the algorithm discovers an augment-

Algorithm 3 The MS-BFS-Graft algorithm. **Input:** A bipartite graph $G(X \cup Y, E)$, an initial matching vector $mate$. **Output:** Updated $mate$ with a maximum cardinality matching.

```
1: for each y ∈ Y in parallel do
2:     visited[y] ← 0, root[y] ← −1, parent[y] ← −1
3: for each x ∈ X in parallel do
4:     root[x] ← −1, leaf[x] ← −1
5: F ← all unmatched X vertices                    ▷ initial frontier
6: for each x ∈ F in parallel do  root[x] ← x
7: repeat
8:     ▷ Step 1: Construct alternating BFS forest
9:     while F ≠ ∅ do
10:        if | F | < numUnvisitedY /α then
11:            F ← TopDown(G, F, ...)
12:        else
13:            R ← unvisited Y vertices
14:            F ← BottomUp(G, R, ...)
15:    ▷ Step 2: frontier F becomes empty. Augment matching.
16:    for every unmatched vertex x0 ∈ X in parallel do
17:        if an augmenting path P from x0 is found then
18:            Augment matching by P
19:    ▷ Step 3: Construct frontier for the next phase
20:    F ← Graft(G, visited, parent, root, leaf, mate)
21: until no augmenting path is found in the current phase
```

Algorithm 4 Top-down construction of the next level BFS frontier from the current frontier F.

```
1: procedure TopDown(G, F, visited, parent, root, leaf, mate)
2:     Q ← ∅                          ▷ next frontier (thread-safe queue)
3:     for x ∈ F in parallel do
4:         if x is in an active tree then      ▷ leaf[root[x]] = −1
5:             for each unvisited neighbor y of x do   ▷ atomic
6:                 Update pointers and queue (Algorithm 5)
       return Q
```

Algorithm 5 Updating pointers in BFS traversals.

```
1: parent[y] ← x, visited[y] ← 1, root[y] ← root[x]
2: if mate[y] ≠ −1 then
3:     Q ← Q ∪ {mate[y]}                              ▷ thread safe
4:     root[mate[y]] ← root[y]
5: else                                  ▷ an augmenting path is found
6:     leaf[root[x]] ← y                          ▷ end of augmenting path
```

Algorithm 6 Bottom-up construction of BFS frontier from a subset of Y vertices R.

```
1: procedure BottomUp(G, R, visited, parent, root, leaf,mate)
2:     Q ← ∅                          ▷ next frontier (thread-safe queue)
3:     for y ∈ R in parallel do
4:         for each neighbor x of y do
5:             if x is in an active tree then   ▷ leaf[root[x]] = −1
6:                 Update pointers and queue (Algorithm 5)
7:                 break                ▷ stop exploring neighbors of y
       return Q
```

ing path $(x_1, y_2, x_4, y_4, x_6, y_6)$. The tree-grafting mechanism therefore reduces the work involved in the reconstruction of alternating trees at the beginning of a phase.

3.2 The MS-BFS-Graft algorithm

We employ tree grafting and direction-optimized BFS [10] to the MS-BFS algorithm and call it the MS-BFS-Graft algorithm. A multithreaded version of this algorithm is described in Algorithm 3 that takes a bipartite graph $G(X \cup Y, E)$ and an initial matching represented by a $mate$ array of size $|X \cup Y|$ as inputs. $mate[u]$ is set to -1 for an unmatched vertex u. The MS-BFS-Graft algorithm updates the $mate$ array with the maximum cardinality matching.

Algorithm 7 Rebuild frontier for the next phase.

```
1: procedure Graft(G, visited, parent, root, leaf, mate)
2:     activeX ← {x ∈ X : root[x] ≠ −1 & leaf[root[x]] = −1}
3:     activeY ← {y ∈ Y : root[y] ≠ −1 & leaf[root[y]] = −1}
4:     renewableY ← {y ∈ Y : root[y] ≠ −1 &
5:                          leaf[root[y]] ≠ −1}
6:     for y ∈ renewableY in parallel do
7:         visited[y] ← 0, root[y] ← −1
8:     if | activeX | > | renewableY |/α then      ▷ tree grafting
9:         F ← BottomUp(G, renewableY, ...)
10:    else                                  ▷ regrow active trees
11:        F ← unmatched X vertices
12:        for y ∈ activeY in parallel do
13:            visited[y] ← 0, root[y] ← −1
14:        for x ∈ (activeX \F) in parallel do
15:            root[y] ← −1
       return F                ▷ return frontier for the next phase
```

We assume that the alternating trees are rooted at unmatched X vertices. Since the alternating forest grows two levels at a time, the BFS frontier F is always a subset of X vertices. A *visited* flag for each Y vertex ensures that it is part of a single tree. The pointer $parent[y]$ points to the parent of a vertex y in Y. A matched X vertex is visited from its $mate$, hence it does not need a *visited* flag or *parent* pointer. For every vertex v in $X \cup Y$, $root[v]$ stores the root of the tree containing v. Finally, $leaf[x]$ stores an unmatched leaf of a tree rooted at x, denoting an augmenting path from x to $leaf[x]$. The *parent*, *root* and *leaf* pointers are set to -1 for a vertex that is not part of any tree. Each iteration of the **repeat-until** block in Algorithm 3 is a *phase* of the algorithm. Each phase is further divided into three steps: (1) discovering a set of vertex-disjoint augmenting paths by multi-source BFS, (2) augmenting the current matching by the augmenting paths, and (3) rebuilding the frontier for the next phase by the tree-grafting mechanism.

Step 1 (BFS traversal): Algorithm 3 employs level-synchronous BFS to grow an alternating BFS forest until the frontier F becomes empty. We use the *direction-optimizing* BFS algorithm [10] that dynamically selects between *top-down* and *bottom-up* traversals based on the frontier size.

The top-down traversal: Algorithm 4 describes the top-down traversal that constructs the next frontier Q by exploring the neighbors of the current frontier F. If a vertex x in F is part of an active tree, then each unvisited neighbor y of x becomes a child of x. Then we update the necessary pointers by Algorithm 5. When y is matched, we include $mate[y]$ into Q. Otherwise, we discover an augmenting path from $root[x]$ to y and set $leaf[root[x]] = y$. In the latter case, $T(root[x])$ becomes a renewable tree and stops growing further.

In the multithreaded implementation of the TopDown function, threads maintain the vertex-disjointedness properties of the alternating trees via atomic updates of the *visited* array. Hence, a vertex y is processed by one thread and becomes a child of a single vertex x in F. (We check the *visited* flags before performing the atomic operations to reduce the overhead of unnecessary atomics [25]). A

vertex is inserted in Q in a thread-safe way (line 3 of Algorithm 5). To reduce memory contention among threads, we assign a small private queue to each thread so that it fits in the local cache. When a private queue is filled up, the associated thread copies the local queue to the global shared queue in a thread-safe manner. This approach is similar to the implementation of `omp_csr` reference code of Graph500 benchmark [26] and improves the scalability of our matching algorithm on multiple sockets. When a thread discovers an augmenting path, it immediately marks the corresponding tree as renewable by setting the *leaf* pointer (line 6). This could create a race condition if multiple threads discover augmenting paths in the same tree at the same time. This is a benign race condition that does not affect correctness because the last update of the *leaf* pointer overwrites previous updates by other threads and maintains a single augmenting path in a tree.

The bottom-up traversal: Algorithm 6 describes the bottom-up traversal that explores the neighborhood of a subset of Y vertices, R, that will be defined below. We use the same BOTTOMUP function for both regular BFS traversal and the tree-grafting steps. Here, R is the set of unvisited Y vertices in the former case and the set of Y vertices in the renewable trees in the latter case. If a vertex y in R has a neighbor x in an active tree, y becomes a child of x. The necessary pointers and the next frontier Q are updated by Algorithm 5, similar to the top-down traversal. We stop exploring the neighbors of a vertex y in R as soon as it is included in an active tree (**break** at line 7). In a multithreaded execution, the vertices in R are concurrently processed by different threads. Since a vertex y in R is processed by a single thread in the BOTTOMUP function, its *visited* flag can be updated without atomic operations.

Top-down or bottom-up?: When the size of the current frontier F is smaller than a fraction $(1/\alpha)$ of the number of unvisited Y vertices $numUnvisitedY$, we use the top-down BFS. Otherwise, the bottom-up BFS is used.

Step 2 (Augment the matching): Assume that x_0 is the root of a renewable tree $T(x_0)$ and $leaf[x_0]{=}y_0$, where both x_0 and y_0 are unmatched vertices. The unique augmenting path P is retrieved from $T(x_0)$ by following the *parent* and *mate* pointers from y_0 to x_0. We augment the matching by flipping the matched and unmatched edges in P. Since the augmenting paths are vertex disjoint, each path can be processed in parallel by different threads (lines 16–18 of Algorithm 3).

Step 3 (Reconstruction of the frontier): When the current frontier becomes empty, Algorithm 3 constructs the frontier for the next phase by calling the GRAFT function described in Algorithm 7. For this step, we identify three sets of vertices: (1) $activeX$ is the set of X vertices in active trees, (2) $activeY$ is the the set of Y vertices in active trees, and (3) $renewableY$ is the the set of Y vertices in renewable trees. We reset *visited* flags and *root* pointers of the $renewableY$ vertices so that they can be reused (lines 6–7 of Algorithm 7).

Algorithm 7 constructs the frontier for the next phase by using the tree-grafting mechanism (line 9) or with the set of unmatched X vertices (line 11). The former is more beneficial than the latter when the size of the renewable forest is smaller than the size of the active forest. Following the same argument of the top-down vs bottom-up traversal, we employ tree grafting when the size of $activeX$ is greater than $|\,renewableY\,|/\alpha$. The BOTTOMUP function grafts vertices from renewable trees onto active trees when the function is called with $renewableY$ as its argument. When it is not profitable to employ tree grafting, we destroy all trees and start building active trees from scratch (lines 11–15). For most graphs, we observe that tree grafting is usually not beneficial in the first few phases when a large number of augmenting paths is discovered (i.e., a large number of renewable trees).

After a new frontier is constructed, Algorithm 3 proceeds to the next phase. The algorithm terminates when no augmenting paths are found in a phase, at which point the maximum cardinality matching is attained.

Time and space complexity: In the worst case, the MS-BFS-Graft algorithm might need n phases, each phase traversing $O(m)$ edges. Hence, the complexity of the (serial) algorithm is $O(mn)$. We store the graph in compressed sparse row (CSR) format that requires $O(n+m)$ storage. Since we store edges in both directions, the actual memory requirement to store the graph is $n+2m$. Additionaly, the MS-BFS-Graft algorithm uses seven arrays of size n.

3.3 Correctness of the algorithm

Lemma 1. *There is no unvisited Y vertex adjacent to vertices in active trees at the end of any phase of the MS-BFS-Graft algorithm.*

Proof. The phases of MS-BFS-Graft algorithm come in batches: each batch starts with a phase with unmatched vertices in the frontier, followed by several phases with grafted frontiers. Since every batch of phases progresses similarly, it suffices to prove the lemma for one such batch of phases beginning with the ith phase. We will prove by induction that there is no unvisited Y vertex adjacent to active trees at the end of $(i+k)$th phase where k successive grafting phases follow the ith phase.

The ith phase builds active trees from scratch starting with all unmatched X vertices. Thus at the end of the ith phase, there is no edge between an active X vertex x_1 and an unvisited Y vertex y_1 because if such an edge were to exist, BFS search would have discovered y_1 from x_1. Assume that phases $i, (i{+}1), ..., (i{+}k)$ have applied tree grafting, and there is no unvisited Y vertex adjacent to vertices in active trees at the end of the $(i{+}k)$th phase. To conclude the induction, we have to show that there is no unvisited Y vertex adjacent to active trees at the end of the $(i{+}k{+}1)$th phase.

Let $\tilde{X}$ be the set of active X vertices and $\tilde{Y}$ be the set of unvisited Y vertices at the end of the $(i{+}k{+}1)$th phase. $\tilde{X}$ can be further divided into two subsets X_1 and X_2, where the former is a subset of active vertices in the $(i{+}k)$th phase and the latter is a subset of newly visited active vertices in the $(i{+}k{+}1)$th phase. Similarly, $\tilde{Y}$ can be divided into two subsets Y_1 and Y_2, where the former is a subset of unvisited vertices and the latter is a subset of renewable vertices (that were not grafted) in the $(i{+}k)$th phase. According to our inductive hypothesis, X_1 has no neighbor in Y_1. In the tree-grafting step in the $(i{+}k)$th phase, all renewable Y vertices adjacent to X_1 have been grafted onto active trees. Hence,

X_1 has no neighbor in Y_2 either. Finally, X_2 is the set of newly visited X vertices in the $(i+k+1)$th phase. Therefore, no unvisited vertex in $Y_1 \cup Y_2$ is adjacent to a vertex in X_2. By induction, there is no unvisited Y vertex adjacent to vertices in active trees at the end of every phase in the batch beginning with the ith phase. □

Theorem 2. *The MS-BFS-Graft algorithm finds a maximum cardinality matching in a bipartite graph $G(X \cup Y, E)$.*

Proof. Let M be the final matching returned by the MS-BFS-Graft algorithm. To obtain a contradiction, assume that M is not a maximum cardinality matching. Then by Berge's theorem [14], there is an M-augmenting path in the graph G that the MS-BFS-Graft algorithm failed to find.

Let $P = (x_0, y_1, x_1, ..., y_k, x_k, y_{k+1})$ be an M-augmenting path in G, where x_0 and y_{k+1} are unmatched vertices. x_0 is an active vertex because it being an unmatched vertex in X is the root of an active tree and y_{k+1} is an unvisited vertex because the algorithm does not discover an augmenting path in the last phase. To form a path P whose end points are active and unvisited vertices, P must include at least one edge connecting an active vertex with an unvisited vertex. Since there is no edge between active and unvisited vertices at the end of any phase according to Lemma 1, P does not exist. Hence, by Berge's theorem [14], M is a maximum cardinality matching. □

Definition 1. Let F be the initial frontier in a phase where an augmenting path P is discovered. The *suffix* of the path P is the alternating path from F to the last unmatched vertex on P. For example, in Fig. 4(e) where the initial frontier is $\{x_2, x_4\}$, the suffix of the augmenting path $(x_1, y_2, x_4, y_4, x_6, y_6)$ is (x_4, y_4, x_6, y_6).

Lemma 2. *Let M be the initial matching in the current phase. Then, the MS-BFS-Graft algorithm discovers an M-augmenting path with the shortest suffix.*

Proof. Let F be the initial frontier in the current phase. Let y_0 be the first unmatched Y vertex discovered in the ith level of BFS search, and P' be corresponding M-alternating path from F to y_0. Since BFS proceeds level by level until the discovery of y_0, no other unmatched Y vertex is discovered before the ith level. Hence, P' is the shortest M-alternating path from F to y_0. □

Even though Lemma 2 has no direct influence on the time complexity of the MS-BFS-Graft algorithm, this property is relevant in comparing MS-BFS-Graft with other maximum matching algorithms. For example, the Hopcroft-Karp algorithm obtains $O(\sqrt{n})$ bound on the number of phases using the "maximal" and "shortest length" properties of augmenting paths. The Pothen-Fan algorithm maintains the "maximal" property, but not the "shortest length" property. By contrast, MS-BFS-Graft discovers augmenting paths with the "shortest suffixes", which translates into shortest augmenting paths if tree grafting is not used. We believe that this property of MS-BFS-Graft would help the readers and other researchers to study MS-BFS-Graft along with other matching algorithms.

4 Experimental Setup

4.1 Methodology and Implementation Details

We evaluate the performance of parallel matching algorithms on two multithreaded multiprocessors, Mirasol and Edison, and on a Knights Corner (KNC) coprocessor based on the Intel Many Integrated Core (MIC) Architecture. Mirasol has 40 Intel Westmere-EX processors, a single node of Edison (a Cray XC30 supercomputer at NERSC) has 24 Intel Ivy Bridge processors and the KNC coprocessor has 60 cores. The KNC coprocessor is part of a NERSC testbed system called Babbage and we use it in "native" mode in which both the operating system and user applications run on the MIC card (i.e., Intel Xeon "host" processors were not used in computing matchings). The specifications of these systems are described in Table 2. We implemented our algorithms using C++ and OpenMP. For atomic memory access, we used compiler builtin functions, such as `__sync_fetch_and_add`.

We use the Karp-Sipser [20] algorithm to initialize all matching algorithms described in this paper, because it has been shown to be one of the best initializers when computing maximum matching on most practical problems [7] and scales well on large number of threads [4]. In our experiments, we found that $\alpha \sim 5$ works the best for most input graphs when deciding about top-down or bottom-up traversals and tree grafting in the MS-BFS-Graft algorithm. Slight performance improvement can be obtained with different α values for different input graphs, but we opt to fix α to 5 so that the algorithm does not need to discover this parameter for different inputs.

To reduce overhead of thread migration, we pinned threads to specific cores. We employed compact thread pinning (filling threads one socket after another) by setting the environment variable `GOMP_CPU_AFFINITY` on Mirasol and `KMP_AFFINITY` on Edison and KNC. Both Mirasol and Edison have non-uniform memory access (NUMA) costs since physically separate memory banks are associated with each socket. When the threads are distributed across sockets, we employed interleaved memory allocation (memory allocated evenly across sockets). Otherwise, we allocated memory on the socket on which the threads were running using the `numactl` command. **Availability:** The source code of the MS-BFS-Graft algorithm is publicly available at https://bitbucket.org/azadcse/ms-bfs-graft.

4.2 Input Graphs

Table 3 describes a representative set of bipartite graphs from the University of Florida sparse matrix collection [27] and a randomly generated RMAT graph. For the RMAT graph, we did not use the default Graph500 [26] parameter (.57, .19, .19, .05) because it creates graphs that are trivial for a matching algorithm. We group the problems into three classes based on application areas where they arise. Note that the matching number is relatively small for problems in the last group.

Let A be an $n_1 \times n_2$ matrix with $nnz(A)$ nonzero entries. We create an undirected bipartite graph $G(X \cup Y, E)$ such that every row (column) of A is represented by a vertex in X (Y), and each nonzero entry $A[i, j]$ of A is represented by two edges (x_i, y_j) and (y_j, x_i) connecting the vertices x_i

TABLE 2
Description of the systems used in the experiments.

Feature	Edison (24-core)	Mirasol (40-core)	KNC (60-core)
Architecture	Ivy Bridge	Westmere-EX	Many Integrated Core (MIC)
Intel Model	E5-2695 v2	E7-4870	Knights Corner
Clock rate	2.4 GHz	2.4 GHz	1 GHz
NUMA domains	2	4	1
Cores	24	40	60
Threads	48	80	240
DRAM size	64 GB	256 GB	8GB
Compiler	icc 14.0.2	gcc 4.4.7	icc 15.0.1
Optimization	-O2	-O2	-O2

TABLE 3
Test problems for evaluating the matching algorithms. The problems are grouped into three classes: (top) scientific computing and random instances, (middle) real world and random networks with large matching numbers, (bottom) networks with small matching numbers. Matching cardinalities are shown as a fraction of total number of non-isolated vertices. Maximal matchings are computed by the Karp-Sipser algorithm.

Class	Graph	#Vertices (M)	#Edges (M)	Maximal Card. (%)	Maximum Card. (%)	Description
Scientific Computing	hugetrace	32.00	96.00	98.68	100.00	Frames from 2D Dynamic Simulations
	delaunay_n24	33.55	201.33	98.58	100.00	Delaunay triangulations of random points
	kkt_power	4.13	29.23	99.03	100.00	Optimal power flow, nonlinear optimization
	rgg_n24_s0	33.55	530.23	99.97	99.99	Random geometric graph
Networks-LargeM	coPapersDBLP	1.08	60.98	97.93	99.95	Citation networks in DBLP
	amazon0312	0.80	3.20	93.82	99.56	Amazon product co-purchasing network
	cit-Patents	7.55	16.52	97.42	97.99	Citation network among US Patents
	RMAT	8.38	255	96.12	96.78	RMAT random graph (param: .45,.15,.15,.25)
Networks-SmallM	road_usa	47.89	115.42	93.68	94.90	USA street networks
	wb-edu	19.69	57.16	79.90	80.50	Web crawl on .edu domain
	web-Google	1.83	5.11	67.45	68.75	Webgraph from the Google prog. contest, 2002
	wikipedia	7.13	45.03	58.33	58.70	Wikipedia page links

TABLE 4
Runtimes (average of ten runs) of the MS-BFS-Graft algorithm for problems in Table 3. Two rightmost columns show the time to read input files and create the graph (both are multithreaded), respectively using 24 threads on Edison. The matching time includes the runtimes of the Karp-Sipser and MS-BFS-Graft algorithms. RMAT graphs were generated on the fly, and `rgg_n24_s0` ran out of memory on KNC.

Graph	Matching time							I/O and data structure	
	Mirasol		KNC			Edison		24 threads of Edison	
	1 thread	40 threads	1 thread	60 threads	240 threads	1 thread	24 threads	File I/O	Graph creation
hugetrace	57.861	3.734	519.465	12.318	5.761	51.605	3.933	3.767	0.504
delaunay_n24	25.703	1.578	148.562	6.312	4.012	17.910	1.341	5.368	0.979
kkt_power	3.579	0.329	24.303	1.599	0.973	3.259	0.331	1.955	0.149
rgg_n24_s0	170.411	22.429	-	-	-	153.204	11.552	11.783	0.884
coPapersDBLP	1.776	0.097	13.509	0.566	0.381	1.115	0.088	2.685	0.137
amazon0312	2.162	0.212	51.470	1.595	0.797	1.755	0.133	0.577	0.029
cit-Patents	2.942	0.204	3.902	0.382	0.254	0.379	0.057	0.151	0.023
RMAT22	89.642	4.421	2069.320	56.086	25.743	87.495	4.657	-	-
road_usa	19.920	1.184	75.327	1.836	1.151	14.621	0.955	4.052	0.350
wb-edu	4.658	0.360	51.210	1.260	0.978	2.699	0.218	4.056	0.180
web-Google	0.653	0.050	6.017	0.216	0.159	0.481	0.034	0.763	0.062
wikipedia	4.635	0.297	20.164	0.637	0.421	3.685	0.265	3.691	0.466

(denoting the ith row) and y_j (denoting the jth column). We keep edges in both directions to facilitate the top-down and bottom-up searches. Hence, $|V|=|X \cup Y|=n_1+n_2=n$ and $|E|=2 \cdot nnz(A)=m$. Two rightmost columns in Table 4 show the time to read input files and construct the graph on 24 cores on Edison. Approximately 30% of the graph construction time was spent on creating edges from the opposite direction.

5 Results

5.1 Relative performance of algorithms

Table 4 shows runtimes of the MS-BFS-Graft algorithm (including the time of the Karp-Sipser initialization) for problems in Table 3 on three shared-memory systems used in our experiments. We compare the performance of the MS-BFS-Graft algorithm shown in Table 4 with the PF (with fairness) and PR algorithms. The latter algorithms are considered the state-of-the art for the maximum cardinality matching problem [4], [5], [7]. The multithreaded implementations of PF and PR are taken from Azad *et al.* [4] and Langguth *et al.* [5], respectively. To improve performance, the PR code uses periodic global relabeling by setting all labels to exact distances [5], [17]. Following the suggestion made in prior work [5], we apply one global relabeling after every n pushes when using one thread and after every $n/8$ pushes when using 40 threads.

Fig. 5 shows the relative performance of three algorithms on Mirasol using one and 40 threads. For every input graph, we compute the average time of 10 runs of each algorithm.

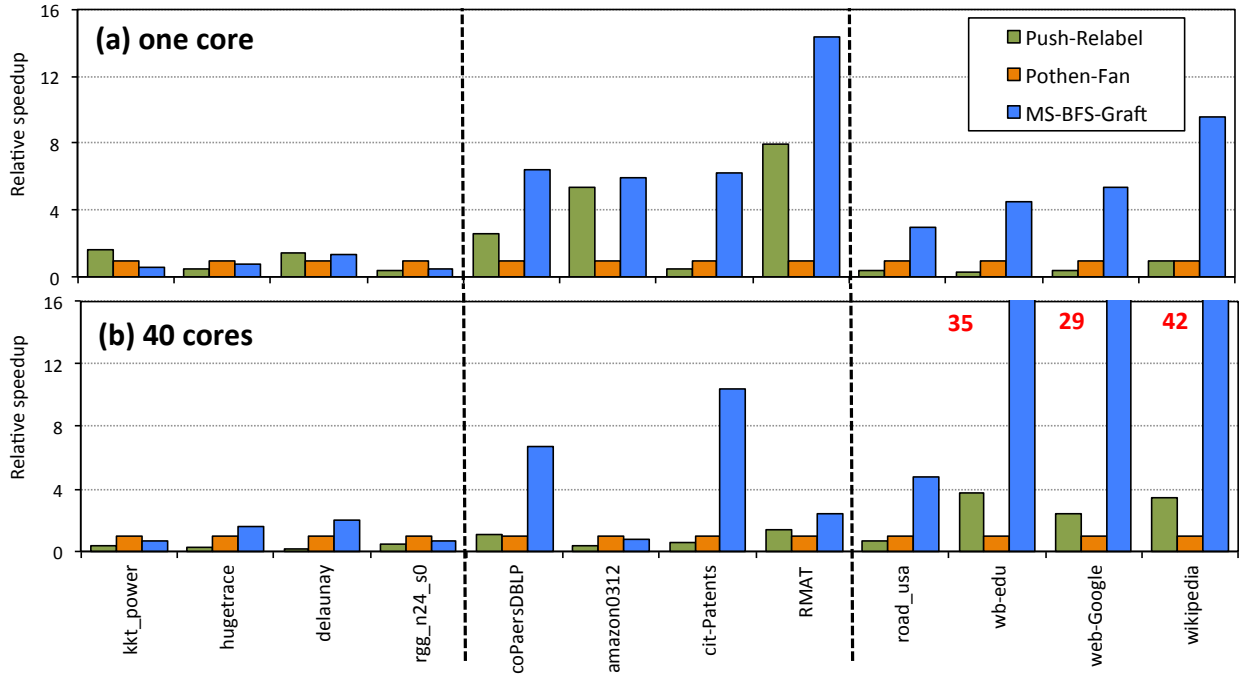

Fig. 5. Speedups of MS-BFS-Graft and PR algorithms relative to the PF algorithm on Mirasol using (a) one core and (b) 40 cores (i.e., PF has the speedup of 1). We truncate y-axis in Subfig. (b) and show large values beside the bars. Dashed vertical lines separate different classes of graphs.

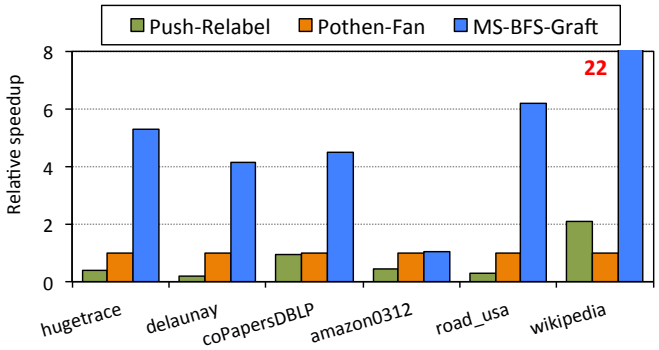

Fig. 6. Speedups of MS-BFS-Graft, PR, and PF algorithms relative to the PF algorithm on Intel Knights Corner using 120 threads (i.e., PF has the speedup of 1).

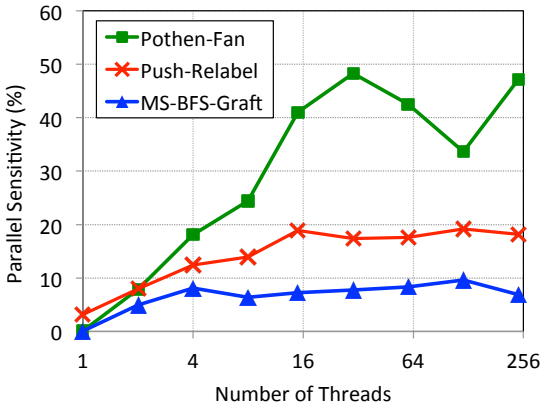

Fig. 7. Parallel sensitivity of three algorithms for `coPapersDBLP` graph on the KNC coprocessor. An algorithm with small parallel sensitivity is more desirable because of its deterministic parallel performance.

We compute the relative speedups of an algorithm by dividing its runtime by the runtime of the PF algorithm (i.e., PF has the speedup of 1). On average, over all problem instances, serial MS-BFS-Graft algorithm runs 5.7x faster than serial PR and 4.8x faster than serial PF algorithms. We did not observe any performance improvement for the first class of graphs (scientific computing and random instances) for serial runs. However, the serial MS-BFS-Graft algorithm runs 4.9x faster than PR and 8.2x faster than PF on networks with large matching numbers, and 11.3x faster than PR and 5.5x faster than PF on networks with small matching numbers. The performance improvement of the MS-BFS-Graft algorithm is more significant on 40 threads (Fig. 5 (b)). On average, our algorithm runs 7.5x faster than PR and 11.4x faster than PF on 40 threads. For different classes of problems, the average performance improvements of the MS-BFS-Graft algorithm are as follows: (a) scientific: 4.9x to PR, 1.2x to PF (2) networks-largeM: 7.1x to PR, 5.1x to PF and (3) networks-smallM: 10.4x to PR, 27.8x to PF. As in the serial case, the performance improvement is more significant for the second and third classes of graphs. Note that the PF algorithm might achieve super-linear speedups for certain networks (e.g., `amazon0312`) because the number of phases needed by the PF algorithm decreases as we increase threads for these graphs [4]. Hence, the PF algorithm becomes the fastest algorithm for `amazon0312` on 40 threads despite it being the slowest serial algorithm for this graph.

Parallel MS-BFS-Graft algorithm also shows superior performance on Intel's KNC coprocessor. Fig. 6 shows the speedups of MS-BFS-Graft, PR, and PF algorithms relative to the PF algorithm on KNC using 120 threads (2 threads per core). On this architecture, PR is usually the slowest algorithm because of its limited scalability on higher concurrency (for example see Fig. 9). The PF algorithm runs up to 5x faster than PR on 120 threads. By contrast, the

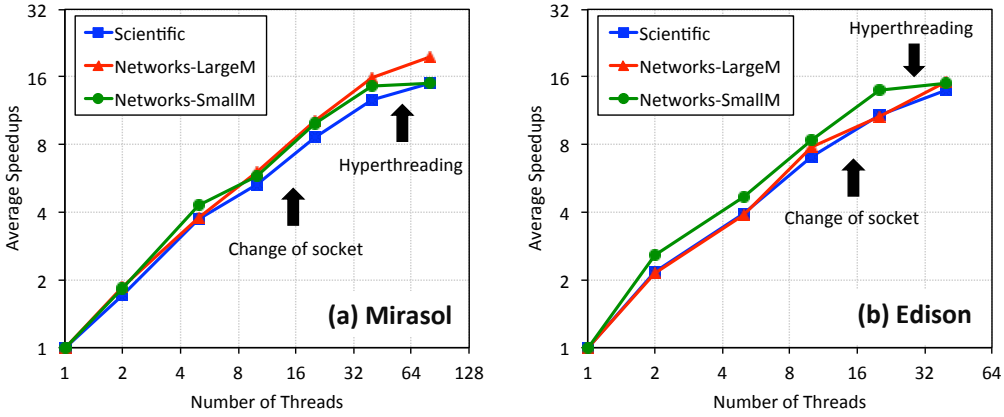

Fig. 8. Strong scaling of the MS-BFS-Graft algorithm for three classes of input graphs on (a) Mirasol and (b) Edison. For each class of graphs, we compute the speedups of individual graphs with respect to the serial MS-BFS-Graft algorithm and take the average.

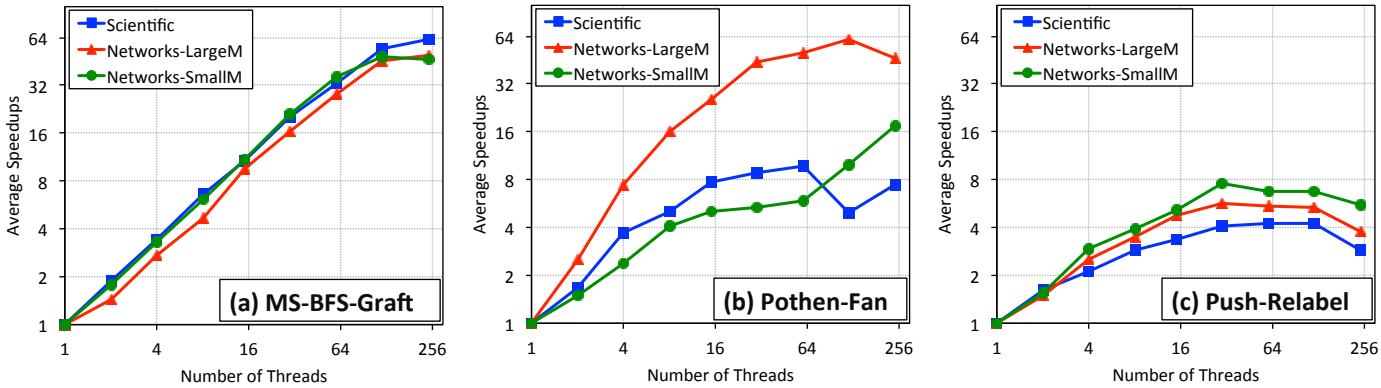

Fig. 9. Strong scaling of MS-BFS-Graft, PF and PR algorithms for three classes of input graphs on the Intel KNC coprocessor. On the right side of the vertical line, more than one thread is run on each core.

newly developed MS-BFS-Graft algorithm runs faster than both PF and PR on all problem instances with the maximum improvement of 22x over the PF algorithm.

The performance of MS-BFS-Graft on KNC is comparable to the reported runtime of an efficient GPU implementation on NVIDIA Tesla C2050 [28]. Here we only consider problems common to our test suite and the GPU paper by Deveci *et al.* [28]. For `hugetrace`, a problem with a perfect matching, MS-BFS-Graft on KNC using 240 threads runs about 30% faster than the GPU implementation. On graphs with small matching numbers, MS-BFS-Graft on KNC performs even better than GPU implementations. For example, for `wikipedia` and `wb-edu`, MS-BFS-Graft on KNC computes the maximum matching in 0.42 and 0.98 seconds, whereas the GPU implementation by Deveci *et al.* takes 1.09 and 33.82 seconds, respectively.

5.2 Variation in parallel runtimes

In a multithreaded environment, different executions of an algorithm are likely to process vertices in different orderings, which could change the runtime of an algorithm. To measure the sensitivity of matching algorithms to the multithreaded environment, we run each algorithm 20 times for each input graph and compute the variation in parallel runtimes. Following prior notation [4], we measure the parallel sensitivity (ψ) of an algorithm as the ratio of the standard deviation (σ) of runtimes from 20 runs, to the mean of runtimes (μ): $\psi = \frac{\sigma}{\mu} \times 100$. For all input graphs of Table 3, we computed ψ for MS-BFS-Graft, PF, and PR algorithms using 40 threads on Mirasol. On average, the variation in runtimes is higher for PF (17%) and PR (10%) relative to the MS-BFS-Graft algorithm (6%).

DFS-based algorithm is more sensitive (i.e., less deterministic) to the multithreaded environment because it assigns each thread to explore a DFS tree, exposing a greater potential for load imbalance. By contrast, MS-BFS-Graft divides work into smaller chunks that could be distributed evenly among threads resulting in less sensitivity in runtimes. The stable parallel performance of MS-BFS-Graft is even more valuable on modern manycore architectures with hundreds of threads. For example, Fig. 7 shows the parallel sensitivity of three algorithms for `coPapersDBLP` graph on the KNC coprocessor. As we increase the number of threads, the sensitivity of PF increases rapidly relative to PR and MS-BFS-Graft. The maximum variation in runtime is about 50% for PF, 20% for PR, and less than 10% for MS-BFS-Graft. This deterministic parallel performance of MS-BFS-Graft is expected to make it an attractive candidate on future architectures.

5.3 Scalability

MS-BFS-Graft algorithm shows good scaling on different multithreaded and MIC architectures. Fig. 8 shows the strong scaling of the MS-BFS-Graft algorithm on Mirasol

and Edison for different classes of graphs. We report the average speedup for graphs in each class with respect to the serial MS-BFS-Graft algorithm. Using all available cores (without hyperthreading), the average speedup of problems in Table 3 is 15 on Mirasol (stdev=3.5, min=9, max=21) and 12 on Edison (stdev=2, min=7, max=15). This performance is significantly better than the reported speedup (5.5x on a 32-core Intel multiprocessor) of the PR algorithm [5].

Both of the multicore multiprocessors used in our experiments are NUMA systems with multiple sockets. By using an efficient queue implementation (discussed earlier), we achieve excellent speedups on multiple sockets on both machines. On average (over all problem instances), we observe 22% performance improvement on Mirasol and 19% performance improvement on Edison when we used hyperthreading. Hence, for the best problem instance, we can achieve up to 35x speedup on Mirasol and 19x speedup on Edison when all the available threads are used with hyperthreading. Unlike PF and PR algorithms [5], the MS-BFS-Graft algorithm continues to scale up to 80 threads of Intel multiprocessors.

On modern manycore systems with a large number of slower cores, MS-BFS-Graft algorithm scales even better than other maximum matching algorithms. Fig 9 shows the strong scaling of three matching algorithms on the KNC coprocessor. On 60 cores (1 threads per core), MS-BFS-Graft achieves 28x-36x speedups for different classes of graphs. By contrast, PR achieves 4x-7x and PF achieves 6x-50x speedups on the same number of cores. As discussed before, PF might show super linear scaling initially on some networks (e.g., 7x speedups on 4 cores in Fig. 9(b)). We note that the speedup achieved by MS-BFS-Graft on 60 cores of the KNC coprocessor is comparable to the reported 15x-20x speedup achieved by traditional BFS on 31 cores of a Knights Ferry (KNF) coprocessor [29]. However, the performance of the latter BFS implementation degraded by a factor of two when four hardware threads were used in each core [29]. By contrast, our more involved matching algorithm continues to scale when several hardware threads are used in each core and achieves up to 2x speedups when we go from 60 to 240 threads on KNC.

5.4 Search rate

We report the search rate of a matching algorithm in millions of traversed edges per second (MTEPS), defined by the ratio of millions of edges traversed in all phases of the algorithm to its total runtime. The MS-BFS-Graft algorithm traverses edges at a faster rate relative to the DFS-based algorithms. Fig. 10 shows that MS-BFS-Graft searches at a rate 2-12 times faster than the PF algorithm for different input graphs on Mirasol. The improvement is larger for graphs with small matching number, e.g., our algorithm searches 12x faster for `wikipedia` and 10x faster for `web-Google`.

Note that the search rate of a matching algorithm is defined differently than traditional BFS searches used in the Graph500 benchmark [26]. The latter computes the search rate using the total number of edges in the graph even though an algorithm might traverse fewer edges (e.g., direction-optimized BFS [10]). By contrast, we use the actual number of edges traversed in the search, augmentation

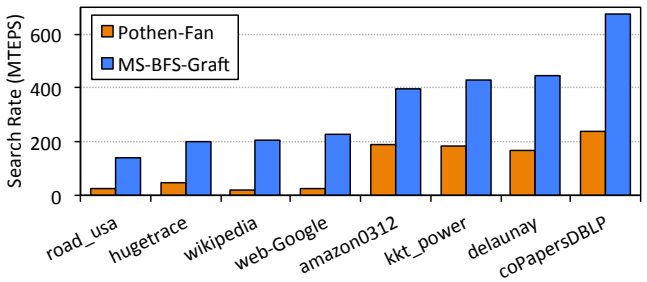

Fig. 10. Search rates of the MS-BFS-Graft and Pothen-Fan algorithms for different input graphs on Mirasol with 40 threads.

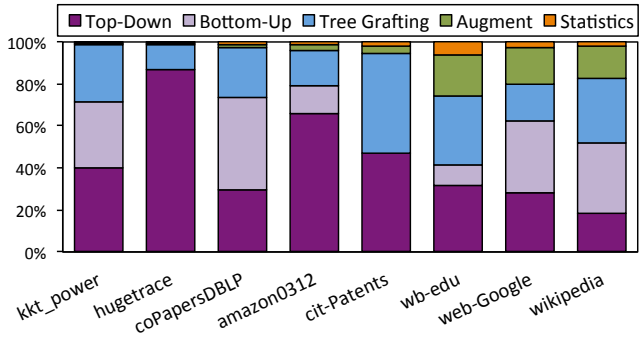

Fig. 11. Breakdown of time spent on different steps of the MS-BFS-Graft algorithm for different graphs on Mirasol with 40 threads.

and grafting steps of matching algorithms. We can not use Graph500-style search rates because matching algorithms use alternating BFS, and each phase of MS-BFS-Graft searches different subgraphs of the input. Therefore, the search rates reported here are not comparable to those reported for direction-optimized BFS [10].

5.5 Breakdown of runtime

Fig. 11 shows the breakdown of runtime of the MS-BFS-Graft algorithm on Mirasol with 40 threads. Here, the "Top-Down" and "Bottom-Up" steps comprise the BFS traversal (Step 1 of Algorithm 3), "Augment" step increases the cardinality of the matching (Step 2 of Algorithm 3), "Tree-Grafting" step constructs frontier for the next phase (Step 3 of Algorithm 3), and "Statistics" denotes the time to collect statistics needed for tree grafting (lines 2-4 of Algorithm 7). For all graphs in Table 3, at least 40% of the time is spent on the BFS traversal. However, graphs with large matching number (e.g., `hugetrace`, `kkt_power`) spend a larger proportion of total runtime on BFS traversal, whereas graphs with small matching number (e.g., `wb-edu`, `wikipedia`) spend more time on the augmentation and tree-grafting steps.

The distributions of total runtime spent on different steps of the MS-BFS-Graft algorithm indeed capture the unique properties of the respective graphs. For example, consider two graphs (a) `wikipedia` and (b) `hugetrace` with small and large matching numbers, respectively. Fig. 12 shows the fraction of active, renewable, and unvisited vertices at different phases of MS-BFS-Graft algorithm run on these two graphs. On `wikipedia`, renewable vertices are always fewer than active vertices because the algorithm maintains a large number of active trees where no augmenting path is found (due to the small matching number of `wikipedia`).

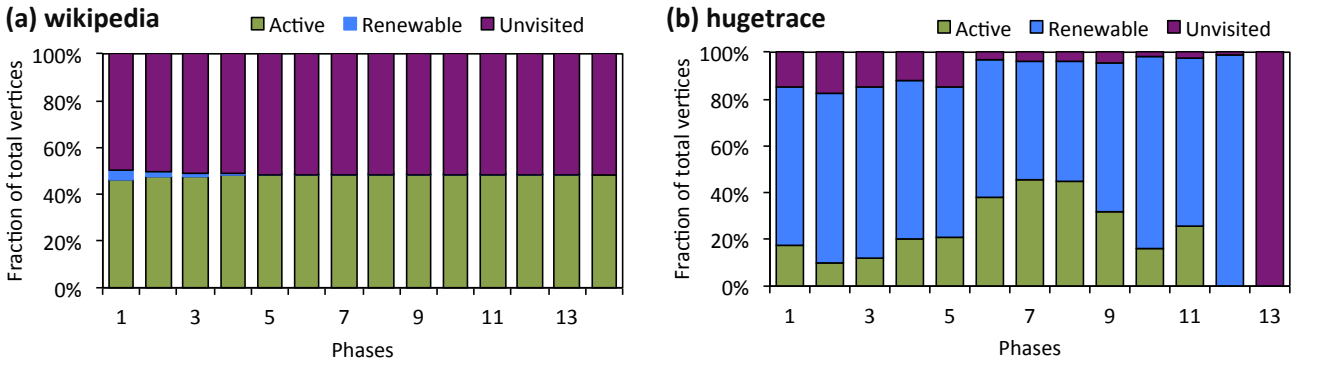

Fig. 12. The fraction of active, renewable and unvisited vertices at the end of each phase of the MS-BFS-Graft algorithm for graphs: (a) `wikipedia` and (b) `hugetrace` on Mirasol with 40 threads.

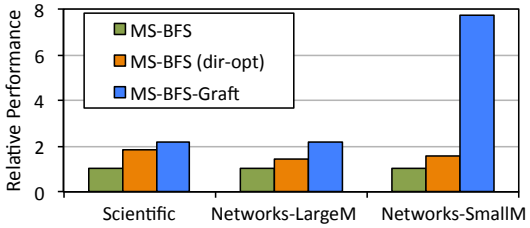

Fig. 13. Performance improvement of the parallel MS-BFS algorithm by direction-optimized BFS and tree grafting for different classes of graphs on Mirasol with 40 threads (normalized to MS-BFS).

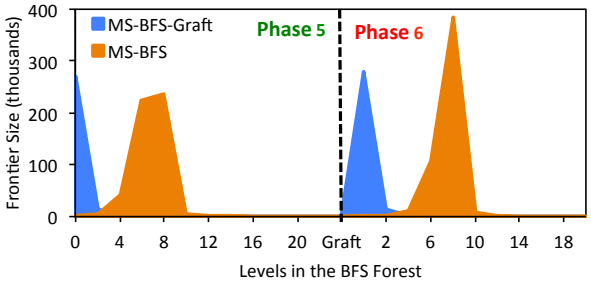

Fig. 14. The size of frontiers in the 5th and 6th phases (separated by a vertical line) of the MS-BFS and MS-BFS-Graft algorithms for `coPapersDBLP` on Mirasol with 40 threads. Tree grafting is employed between two phases.

Since tree grafting only explores the neighborhood of renewable vertices, it is much faster than rebuilding all active trees for this graph. By contrast, `hugetrace` maintains significantly more renewable vertices than active vertices, which makes the tree-grafting mechanism more expensive than reconstructing the active trees (due to the large matching number of `hugetrace`). Hence, for `hugetrace` the algorithm spends more time on BFS traversal than tree grafting as was shown in Fig. 11.

5.6 Performance contributions

Fig. 13 shows the effects of direction-optimizing BFS and tree grafting on the performance of parallel MS-BFS. On average, direction optimization speeds the MS-BFS algorithm up by 1.6x, and tree grafting speeds it up by another 3x. Graphs with relatively small matching number benefit most from tree grafting, with 7.8x.

Fig. 14 shows the size of BFS frontiers at two phases of the MS-BFS and MS-BFS-Graft algorithms on `coPapersDBLP`. At the beginning of each phase, tree grafting generates a large frontier that gradually shrinks as the algorithm progresses level by level. By contrast, without grafting, a phase starts with a small frontier (the unmatched vertices) that grows to the highest size before shrinking. Hence, tree grafting reduces the height of BFS forests, decreasing the synchronization points of the parallel algorithm. Furthermore, tree grafting decreases the number of vertices in an alternating forest (the area under the curve), thus reducing the work in graph traversals at the expense of additional work in the tree-grafting step.

6 Conclusions

We presented a novel multi-source (MS) cardinality matching algorithm that can reuse the trees constructed in earlier phases. This method, called *tree grafting*, eliminates redundant augmenting path reconstructions, which is a major impediment of MS algorithms for achieving high performance on several classes of graphs, especially those with small matching number. By combining tree grafting, direction-optimizing BFS searches, and an efficient parallel implementation, we compute maximum cardinality matchings an order of magnitude faster than the current best performing algorithms on graphs with small matching numbers. The newly developed MS-BFS-Graft algorithm scales up to 80 threads of an Intel multiprocessor and up to 240 threads on an Intel Knights Corner coprocessor, yields better search rates than its competitors, and is less sensitive to performance variability of multithreaded platforms. This insensitivity may be valuable for future systems with nonuniform performance characteristics due to various reasons such as frequent error correction and near-threshold voltage scaling [30]. The MS-BFS-Graft algorithm employs level synchronous BFSs for which efficient distributed algorithms exist [31]. In the future, we plan to develop a distributed memory MS-BFS-Graft algorithm, which could be used in static pivoting for solving large sparse systems of linear equations [32], and computing the block triangular form.

7 Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under contract number No. DE-AC02-05CH11231 and by NSF grants CCF 1218196 and 1552323, and DOE grant DE-FG02-13ER26135.

REFERENCES

[1] A. Pothen and C.-J. Fan, "Computing the block triangular form of a sparse matrix," *ACM Trans. Math. Softw.*, vol. 16, pp. 303–324, 1990.

[2] T. A. Davis and E. P. Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*, vol. 37, no. 3, p. 36, 2010.

[3] A. Pothen, "Predicting the structure of sparse orthogonal factors," *Linear algebra and its applications*, vol. 194, pp. 183–203, 1993.

[4] A. Azad, M. Halappanavar, S. Rajamanickam, E. G. Boman, A. Khan, and A. Pothen, "Multithreaded algorithms for maximum matching in bipartite graphs," in *IPDPS*. IEEE, 2012, pp. 860–872.

[5] J. Langguth, A. Azad, M. Halappanavar, and F. Manne, "On parallel push–relabel based algorithms for bipartite maximum matching," *Parallel Computing*, 2014.

[6] J. C. Setubal, "Sequential and parallel experimental results with bipartite matching algorithms," *Univ. of Campinas, Tech. Rep. IC-96-09*, 1996.

[7] I. S. Duff, K. Kaya, and B. Uçar, "Design, implementation, and analysis of maximum transversal algorithms," *ACM Trans. Math. Softw.*, vol. 38, no. 2, pp. 13:1– 13:31, 2011.

[8] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Courier Dover Publications, 1998.

[9] J. Hopcroft and R. Karp, "A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM J. Comput.*, vol. 2, pp. 225–231, 1973.

[10] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3, pp. 137–148, 2013.

[11] J. Langguth, F. Manne, and P. Sanders, "Heuristic initialization for bipartite matching problems," *Journal of Experimental Algorithmics (JEA)*, vol. 15, pp. 1–3, 2010.

[12] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *Journal of the ACM*, vol. 35, no. 4, pp. 921–940, 1988.

[13] D. P. Bertsekas, "The auction algorithm: A distributed relaxation method for the assignment problem," *Annals of operations research*, vol. 14, no. 1, pp. 105–123, 1988.

[14] C. Berge, "Two theorems in graph theory," *Proceeding of National Academy of Science*, pp. 842–844, 1957.

[15] R. Burkard, M. Dell'Amico, and S. Martello, *Assignment Problems*. Society for Industrial and Applied Mathematics, 2009.

[16] A. Azad, A. Buluç, and A. Pothen, "A parallel tree grafting algorithm for maximum cardinality matching in bipartite graphs," in *IPDPS*. IEEE, 2015.

[17] K. Kaya, J. Langguth, F. Manne, and B. Uçar, "Push-relabel based algorithms for the maximum transversal problem," *Computers & Operations Research*, vol. 40, no. 5, pp. 1266–1275, 2013.

[18] D. P. Bertsekas and D. A. Castañon, "Parallel synchronous and asynchronous implementations of the auction algorithm," *Parallel Computing*, vol. 17, no. 6, pp. 707–732, 1991.

[19] M. Brady, K. K. Jung, H. Nguyen, R. Raghavan, and R. Subramonian, "The assignment problem on parallel architectures," *Network Flows and Matching. DIMACS*, pp. 469–517, 1993.

[20] R. M. Karp and M. Sipser, "Maximum matching in sparse random graphs," in *FOCS'81*. IEEE, 1981, pp. 364–375.

[21] A. V. Goldberg and R. Kennedy, "Global price updates help," *SIAM Journal on Discrete Mathematics*, vol. 10, no. 4, pp. 551–572, 1997.

[22] A. Azad and A. Pothen, "Multithreaded algorithms for matching in graphs with application to data analysis in flow cytometry," in *IPDPSW*. IEEE, 2012, pp. 2494–2497.

[23] R. Motwani, "Average-case analysis of algorithms for matchings and related problems," *Journal of the ACM (JACM)*, vol. 41, no. 6, pp. 1329–1356, 1994.

[24] H. Bast, K. Mehlhorn, G. Schafer, and H. Tamaki, "Matching algorithms are fast in sparse random graphs," *Theory of Computing Systems*, vol. 39, no. 1, pp. 3–14, 2006.

[25] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *SC'10*, 2010.

[26] "Graph500 benchmark," www.graph500.org.

[27] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, p. 1, 2011.

[28] M. Deveci, K. Kaya, B. Uçar, and Ü. V. Çatalyürek, "Gpu accelerated maximum cardinality matching algorithms for bipartite graphs," in *Euro-Par 2013 Parallel Processing*. Springer, 2013, pp. 850–861.

[29] E. Saule and U. V. Catalyurek, "An early evaluation of the scalability of graph algorithms on the intel mic architecture," in *IPDPSW*. IEEE, 2012, pp. 1629–1639.

[30] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the "new normal" for computer artichture," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 16–26, 2013.

[31] S. Beamer, A. Buluç, K. Asanović, and D. Patterson, "Distributed memory breadth-first search revisited: Enabling bottom-up search," in *IPDPSW*. IEEE Computer Society, 2013.

[32] X. S. Li and J. W. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Trans. Math. Softw.*, vol. 29, no. 2, pp. 110–140, 2003.

Ariful Azad is a postdoctoral fellow in the Computational Research Division at Lawrence Berkeley National Laboratory. His research interests are in parallel computing, high-performance graph analysis, and bioinformatics. Ariful received his PhD in Computer Science from Purdue University in 2014 and BSc in Computer Science and Engineering from Bangladesh University of Engineering and Technology in 2006. Ariful has received an IBM PhD fellowship, a Purdue fellowship, a fellowship incentive grant, an outstanding poster award.

Aydın Buluç is a Research Scientist at the Lawrence Berkeley National Laboratory. His research interests include parallel computing, combinatorial scientific computing, high-performance graph analysis, sparse matrix computations, and computational genomics. Previously, he was a Luis W. Alvarez Postdoctoral Fellow at LBNL and a visiting scientist at the Simons Institute for the Theory of Computing. He received his Ph.D. in Computer Science from the University of California, Santa Barbara in 2010 and his B.S. in Computer Science and Engineering from Sabanci University, Turkey in 2005. Dr. Buluç received the DOE Early Career Award in 2013 and the IEEE TCSC Award for Excellence for Early Career Researchers in 2015. He is also a founding associate editor of the ACM Transactions on Parallel Computing.

Alex Pothen is a professor of computer science at Purdue University. He led a pioneering research project in combinatorial scientific computing, as the Director of the CSCAPES Institute funded by the U.S. Department of Energy, to work on combinatorial algorithms to enable computational science and engineering on extreme-scale computers. Alex's research interests span combinatorial scientific computing, parallel computing, computational science and engineering, and bioinformatics. Alex received an undergraduate degree from the Indian Institute of Technology, Delhi, and a PhD from Cornell in applied mathematics. He is an editor of the Journal of the ACM, the SIAM Review, SIAM Books and other publications. Alex has received a National Science Talent Scholarship, the Director's Silver Medal and a Distinguished Alumnus award from IIT Delhi, a Cornell fellowship, an IBM University Research award, two distinguished teaching awards, and several best paper prizes. He has advised more than twenty PhD students and postdoctoral scientists.