A Parallel Tree Grafting Algorithm for Maximum Cardinality Matching in Bipartite Graphs

Ariful Azad[1], Aydın Buluç[1], and Alex Pothen[2],

E-mail: azad@lbl.gov, abuluc@lbl.gov, and apothen@purdue.edu

[1]Lawrence Berkeley National Laboratory, [2]Purdue University

Abstract—In computing matchings in graphs on parallel processors, it is challenging to achieve high performance because these algorithms rely on searching for paths in the graph, and when these paths become long, there is little concurrency. We present a new algorithm and its shared-memory parallelization for 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. Unfortunately, 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 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. 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 low matching number.

I. 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 multi-source graph searches, i.e., by searching with many threads from multiple

(unmatched) vertices for augmenting paths that can increase the cardinality of the matching.

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 [6], so it can remove u (and other vertices in its search tree) from further consideration. However, the search trees in MS algorithms are constrained to be vertex-disjoint to allow concurrent augmentations along multiple augmenting paths [7], [8]. 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 failing to 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 grafted 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 [9] 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 multi-threaded shared memory multiprocessors.

Our main contributions in this paper are as follows:

- 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 [9] 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.

II. Existing Algorithms for Maximum Matching in Bipartite Graphs

A. 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 in V. 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 size of the matching M by one (this is equivalent to the symmetric difference of M and P, $M \oplus P=(M \backslash P) \cup (P \backslash M)$). Given a set of vertex disjoint M-augmenting paths $\mathbb{P}$, $M'=M \oplus \mathbb{P}$ is a matching with cardinality $|M|+|\mathbb{P}|$.

B. Classes of cardinality matching algorithms

Maximal matching algorithms compute a matching with cardinality at least half of the maximum matching. For many input graphs, the maximal matching algorithms finds all or a large fraction of the maximum cardinality matching [10]. 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. We use the Karp-Sipser [11] algorithm to initialize all matching algorithms described in this paper, because it is one of the best initializer algorithms for cardinality matching [8].

Maximum matching algorithms are broadly classified into two groups: (1) augmenting-path based and (2) push-relabel based [12], [5]. 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 when no augmenting path is discovered in a phase [13]. 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 [7]). For more details, we refer the reader to a book on matching algorithms [14].

In this paper, without loss of generality, we search for augmenting paths from unmatched X vertices, one vertex part 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 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 $T(x)$.

C. 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. Hence, we can remove x_0 and all other vertices and edges in $T(x_0)$ from further consideration [6], [8], [15]. If a search tree $T(x_0)$ fails to produce an augmenting path, the SS-MATCH function does not clear the *visited* flags of the vertices in $T(x_0)$, thus hiding them from future

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, 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, M)
2:     **for** each $y \in Y$ **do** $visited[y] \leftarrow 0$
3:     **repeat**
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 using unvisited vertices. $visited[y]$ is set to 1 for each traversed vertex y in Y.
6:         $Y_s \leftarrow Y$ vertices traversed in the latest search
7:         **for** each $y \in Y_s$ **do** $visited[y] \leftarrow 0$
8:         $M \leftarrow M \oplus \mathbb{P}$ ▷ Increase matching by $|\mathbb{P}|$
9:     **until** $\mathbb{P} = \phi$ ▷ Continue if an augmenting path is found
---

searches. Otherwise, we clear the *visited* flags of Y-vertices in $T(x_0)$, making them available for future searches (line 8 of Algorithm 1). By contrast, an MS algorithm cannot discard vertices from a tree $T(x_0)$ without an augmenting path in a phase of the algorithm, because the same tree could yield an augmenting path in future phases. Consequently, Algorithm 2 always clears the *visited* flags of the Y-vertices in the current forest and makes them available for future searches (line 7).

D. DFS- and BFS-based Algorithms

SS and MS algorithms search for augmenting paths using DFS, BFS, or a combination of both. We briefly describe three algorithms with theoretical and practical importance [8], [4].

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.* [8] 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 [7] 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 could be bounded by $O(\sqrt{n})$, resulting in faster asymptotic time complexity ($m\sqrt{n}$) [7].

E. Characteristics of existing algorithms

We investigate three properties of matching algorithms: (a) the number of traversed edges, (b) the number of phases, and (c) average length of augmenting paths. Since matching algorithms spends most of its time on graph searches (Fig. 6), the first property determines the their serial execution time. Parallel matching algorithms need to synchronize between consecutive phases. Due to the complexity of storing explicit paths, most parallel algorithms represent an augmenting path P by a chain of parent pointers and augment matching by P sequentially (vertex-disjoint augmenting paths are processed in parallel). Hence, the second and third properties influence the performance of parallel algorithms. We compare 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.* [8]. We selected three graphs (`kkt-power`, `cit-patents`, and `wikipedia`), one from each class of graphs described in Table IV-A.

Number of traversed edges (Fig. 1(a)): PF and MS-BFS algorithms traverse fewer edges than HK algorithm in spite of the latter's superior asymptotic time complexity [8], [4]. 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 lower matching numbers (`cit-patents`, and `wikipedia`), SS-BFS traverses the fewest number of edges.

Number of phases (Fig. 1(b)): MS algorithms could identify multiple augmenting paths in a single phase; hence they need fewer number of phases than the SS algorithms. For scale free and web graphs, the PF algorithm might require more phases than MS-BFS because the former usually finds longer augmenting paths instead of several short augmenting
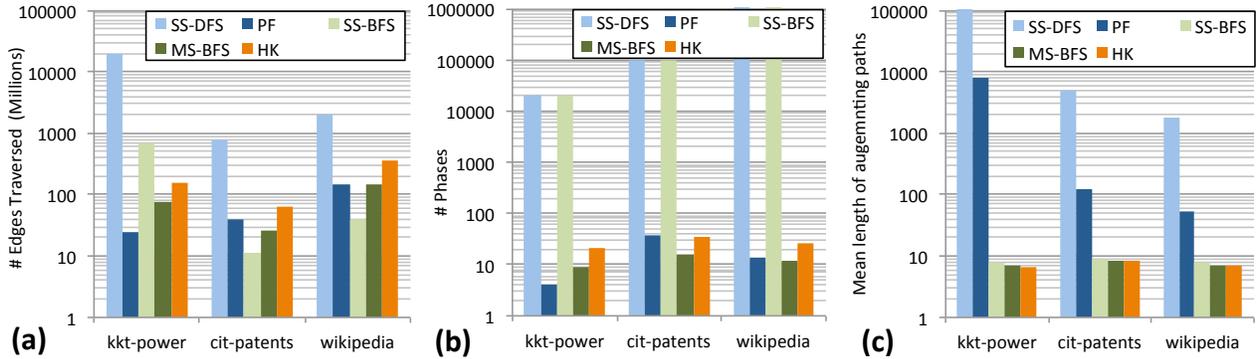
Fig. 1. (a) The number of traversed edges, (b) the number of phases, and (c) the average length of augmenting paths for five maximum matching algorithms.

paths. Despite the superior bound ($\sqrt{n}$) 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. 1(c)): BFS-based algorithms finds shorter augmenting paths than DFS-based algorithms. The HK algorithm guarantees to discover 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 algorithm.

Practical considerations: For the three classes of graphs that we consider, PF, SS-BFS, and MS-BFS algorithms traverse fewer edges than others and are expected to run faster when run sequentially. The SS-BFS algorithm can remove a search tree when no augmenting path is found in it (see the discussion in Section II-C). Hence, SS-BFS traverses fewer edges when a graph has a small matching number. In the next section, we describe a tree-grafting mechanism that reduces the repetition of work across multiple phases in MS algorithms. MS algorithms are more scalable because of increased concurrency and decreased synchronization between consecutive phases. In contrast to the PF algorithm that employs course grained parallelism [4], the MS-BFS algorithm can employ fine-grained parallelism with each thread processing one vertex in a level of a BFS tree. Hence, we employ tree-grafting to enhance MS-BFS and show that its parallel implementation outperforms the competitors for most practical graphs.

III. MS-BFS ALGORITHM WITH TREE GRAFTING

A. Intuition behind the algorithm

Consider a maximal matching in a bipartite graph shown in Fig. 2(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. 2(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. 2(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. 2(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 $(x_3, y_3))$ into $T(x_1)$, 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, vertices that are not explored in the current phase (x_6 and y_6 in Fig. 2(a)) are called *inactive vertices*. 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. 2(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 matched vertices are grafted into $T(x_1)$ as shown in Fig. 2(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 (see Fig. 2(d)) and continues growing $T(x_1)$. Fig. 2(e) shows the next phase where the algorithm discovers an augmenting 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 every phase.

B. The MS-BFS-Graft algorithm

We employ tree grafting and direction-optimized BFS [9] 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.

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,
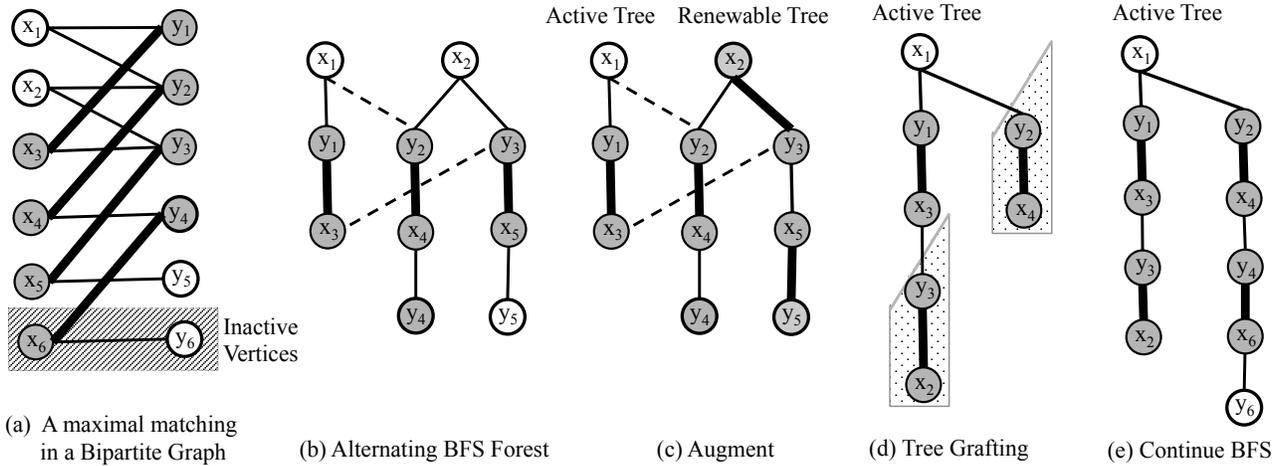
Fig. 2. (a) A maximal matching in a bipartite graph. Matched and unmatched vertices are shown in filled and empty circles, respectively. Thin lines represent unmatched edges and thick lines represent matched edges. (b) A BFS forest with two trees $T(x_1)$ and $T(x_2)$ created by the MS-BFS algorithm. 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. Inactive 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 into $T(x_1)$. The vertices x_2 and x_4 form the new frontier. (e) BFS proceeds from the new frontier and finds an augmenting path in $T(x_1)$.

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 $y \in Y$ **in parallel do**
2:      $visited[y] \leftarrow 0, \; root[y] \leftarrow -1, \; parent[y] \leftarrow -1$
3: **for** each $x \in X$ **in parallel do**
4:      $root[x] \leftarrow -1, \; leaf[x] \leftarrow -1$
5: $F \leftarrow$ all unmatched X vertices   ▷ initial frontier
6: **repeat**
7:      ▷ **Step 1:** Construct alternating BFS forest
8:      **while** $F \neq \emptyset$ **do**
9:          **if** $|F| < numUnvisitedY / \alpha$ **then**
10:             $F \leftarrow$ TOPDOWN$(G, F, ...)$
11:         **else**
12:             $R \leftarrow$ unvisited Y vertices
13:             $F \leftarrow$ BOTTOMUP$(G, R, ...)$
14:     ▷ **Step 2:** frontier F becomes empty. Augment matching.
15:     **for** every unmatched vertex $x_0 \in X$ **in parallel do**
16:         **if** an augmenting path P from x_0 is found **then**
17:             Augment matching with P
18:     ▷ **Step 3:** Construct frontier for the next phase
19:     $F \leftarrow$ GRAFT$(G, visited, parent, root, leaf, mate)$
20: **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:      $Q \leftarrow \emptyset$ ▷ next frontier (thread-safe queue)
3:      **for** $x \in F$ **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

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

---

Algorithm 5 Updating pointers in BFS traversals.

1: $parent[y] \leftarrow x, \; visited[y] \leftarrow 1, \; root[y] \leftarrow root[x]$
2: **if** $mate[y] \neq -1$ **then**
3:      $Q \leftarrow Q \cup \{mate[y]\}$ ▷ thread safe
4:      $root[mate[y]] \leftarrow root[y]$
5: **else**   ▷ an augmenting path is found
6:      $leaf[root[x]] \leftarrow y$ ▷ 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:      $Q \leftarrow \emptyset$ ▷ next frontier (thread-safe queue)
3:      **for** $y \in R$ **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

Algorithm 7 Rebuild frontier for the next phase.

1: **procedure** GRAFT$(G, visited, parent, root, leaf, mate)$
2:      $activeX \leftarrow \{x \in X : root[x] \neq -1 \; \& \; leaf[root[x]] = -1\}$
3:      $activeY \leftarrow \{y \in Y : root[y] \neq -1 \; \& \; leaf[root[y]] = -1\}$
4:      $renewableY \leftarrow \{y \in Y : root[y] \neq -1 \; \&$
5:                          $leaf[root[y]] \neq -1\}$
6:      **for** $y \in renewableY$ **in parallel do**
7:          $visited[y] \leftarrow 0, \; root[y] \leftarrow -1$
8:      **if** $|activeX| > |renewableY| / \alpha$ **then**   ▷ tree grafting
9:          $F \leftarrow$ BOTTOMUP$(G, renewableY, ...)$
10:     **else**   ▷ regrow active trees
11:         $F \leftarrow$ unmatched X vertices
12:         **for** $y \in activeY$ **in parallel do**
13:             $visited[y] \leftarrow 0, \; root[y] \leftarrow -1$
14:         **for** $x \in activeX$ **in parallel do**
15:             $root[y] \leftarrow -1$
        return F ▷ return frontier for the next phase

augmenting paths by multi-source BFS, (2) augmenting the current matching with the augmenting paths, and (3) rebuilding the frontier for the next phase by the tree-grafting mechanism.

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 with the augmenting paths, and (3) rebuilding 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 [9] 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 a 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* pointers. 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 [16]). A vertex is inserted in Q in a thread-safe way (line 3 of Algorithm 5). To reduce memory contentions 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. This approach is similar to the implementation of `omp_csr` reference code of Graph500 benchmark [17]. 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. Unlike the TOPDOWN function, a vertex y in R is processed by a single thread in the BOTTOMUP function. Hence, *visited* flags 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 top-down BFS. Otherwise, the bottom-up BFS is used. The parameter α is greater than one since the bottom-up BFS traverses a fraction of the neighbors of R. In our experiments, we found that $\alpha \sim 5$ performs better for the MS-BFS-Graft algorithm.

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 15–17 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 an active tree, (2) $activeY$ is the the set of Y vertices in an active tree, and (3) $renewableY$ is the the set of Y vertices in a renewable tree. 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 the 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.

C. Correctness of the algorithm

Theorem 1: The MS-BFS-Graft algorithm finds a maximum cardinality matching M in a bipartite graph $G(X \cup Y, E)$.

Proof: To obtain a contradiction, assume that M is not a maximum cardinality matching. Then by Berge's theorem, 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. The MS-BFS-Graft algorithm discovers no augmenting path in the last phase, and thus there are no renewable trees when the

algorithm terminates. Hence, each vertex u on the augmenting path P is either active or inactive (a vertex is active if it is in an active tree). x_0 is an active vertex because it being an unmatched vertex in X is the root of an active tree. Suppose that all vertices in the alternating path $P' = (x_0, y_1, x_1, ..., y_i, x_i)$ with $0 \leq i < k$ are active. Since P is a path, (x_i, y_{i+1}) is an edge in G. Therefore, y_{i+1} is an active vertex because it must be explored by x_i (and possibly by other active vertices as well) in the BFS traversal and included in an active tree. Furthermore, if y_{i+1} is matched, its mate x_{i+1} is also active. By induction, every vertex $u \in P$ is an active vertex.

If the unmatched vertex y_{k+1} belongs to another active tree $T(x_j)$ with $x_j \neq x_0$, then there exists an M-alternating path from the unmatched vertex x_j (root of this active tree) to the unmatched vertex y_{k+1} consisting entirely of vertices and edges in $T(x_j)$. This is an M-augmenting path that the MS-BFS-Graft algorithm would have discovered since it belongs to the active tree $T(x_j)$. This contradiction proves that the MS-BFS-Graft algorithm terminates without an augmenting path in G, and there does not exist an augmenting path in G with respect to M. Hence M is a maximum cardinality matching.

Time 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 $O(mn)$.

IV. Experimental Setup

A. Methodology and Implementation Details

We evaluate the performance of parallel matching algorithms on Mirasol, an Intel Nehalem-based machine, and on a single node of Edison, a Cray XC30 supercomputer at NERSC. The specifications of these systems are described in Table IV-A. We implemented our algorithms using C++ and OpenMP. For atomic memory access, we used compiler builtin functions __sync_fetch_and_add and __sync_fetch_and_or. We store current and next frontiers in parallel queues, similar to the implementation of omp_csr reference code of the Graph500 benchmark [17]. In particular, we assign a small private queue for 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. These queue management scheme improves the scalability of our matching algorithm significantly across multiple sockets.

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 environment variable GOMP_CPU_AFFINITY on Mirasol and KMP_AFFINITY on Edison. Both of these machines 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 are running using the numactl command.

Feature	Edison (24-core)	Mirasol (40-core)
Architecture	Ivy Bridge	Westmere-EX
Intel Model	E5-2695 v2	E7-4870
Clock rate	2.4 GHz	2.4 GHz
#Sockets	2	4
Cores/socket	12	10
Threads/socket	24	20
DRAM size	64 GB	256 GB
L3 cache/socket	30 MB	30 MB
Compiler	icc 14.0.2	gcc 4.4.7
Optimization	-O2	-O2

TABLE I
DESCRIPTION OF THE SYSTEMS USED IN THE EXPERIMENTS.

B. Input Graphs

Table IV-A describes a representative set of bipartite graphs from the University of Florida sparse matrix collection [18] and from randomly generated instances. 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 (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 $|E| = 2 \cdot nnz(A) = m$. The RMAT random graph with skewed degree distribution is generated by the Graph500 implementation [17]. In Table IV-A, we group the problems into three classes based on application areas where they arise. Note that the matching number is relatively low for problems in the last group.

V. Results

A. Relative performance of algorithms

We compare the performance of the MS-BFS-Graft algorithm with the PF (with fairness) and PR algorithms, because the latter algorithms performed better in several recent studies [4], [5], [8]. The multithreaded implementation of the PF and PR algorithms are taken from Azad *et al.* [4] and Langguth *et al.* [5], respectively. As suggested in the previous work on PR algorithm [5], [19], we set the queue limit to 500, and the relabel frequency to 2 on one thread and to 16 on 40 threads.

Fig. 3 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. We compute the relative speedups of an algorithm by dividing its runtime by the runtime of the slowest algorithm (i.e., the slowest algorithm for a graph has 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 no observe any performance improvement for the first class of graphs (scientific computing and random instances with high matching number) for serial runs. However, the serial MS-BFS-Graft algorithm runs 4.9x faster than PR and 8.2x faster than PF for the scale-free graphs, and 11.3x faster than PR and 5.5x faster than PF for the web and road networks. The performance improvement of the MS-BFS-Graft algorithm is more significant on 40 threads (Fig. 3

Class	Graph	#Vertices (M)	#Edges (M)	Avg. Degree	Maximal Card. (%)	Maximum Card. (%)	Description
Scientific Computing	hugetrace	32.00	96.00	3.00	98.68	100.00	Frames from 2D Dynamic Simulations
	delaunay_n24	33.55	201.33	6.00	98.58	100.00	Delaunay triangulations of random points
	kkt_power	4.13	29.23	7.08	99.03	100.00	Optimal power flow, nonlinear optimization
	rgg_n24_s0	33.55	530.23	15.80	99.97	99.99	Random geometric graph
Scale-free	coPaperDBLP	1.08	60.98	56.41	97.93	99.95	Citation networks in DBLP
	amazon0312	0.80	6.40	7.99	93.82	99.56	Amazon product co-purchasing network
	cit-Patents	7.55	33.04	4.38	97.42	97.99	Citation network among US Patents
	RMAT	8.38	255	30.4	96.12	96.78	RMAT random graph (param: .45,.15,.15,.25)
Networks	road_usa	47.89	115.42	2.41	93.68	94.90	USA street networks
	wb-edu	19.69	114.31	5.81	79.90	80.50	Web crawl on .edu domain
	web-Google	1.83	10.21	5.57	67.45	68.75	Webgraph from the Google prog. contest, 2002
	wikipedia	7.13	90.06	12.62	58.33	58.70	Wikipedia page links

TABLE II

TEST PROBLEMS FOR EVALUATING THE MATCHING ALGORITHMS. THE PROBLEMS ARE GROUPED INTO THREE CLASSES: (TOP) SCIENTIFIC COMPUTING AND RANDOM INSTANCES, (MIDDLE) SCALE FREE GRAPHS, (BOTTOM) ROAD AND WEB NETWORKS. MATCHING CARDINALITIES ARE SHOWN AS A FRACTION OF TOTAL NUMBER OF VERTICES. THE MAXIMAL MATCHING IS COMPUTED BY THE KARP-SIPSER ALGORITHM.
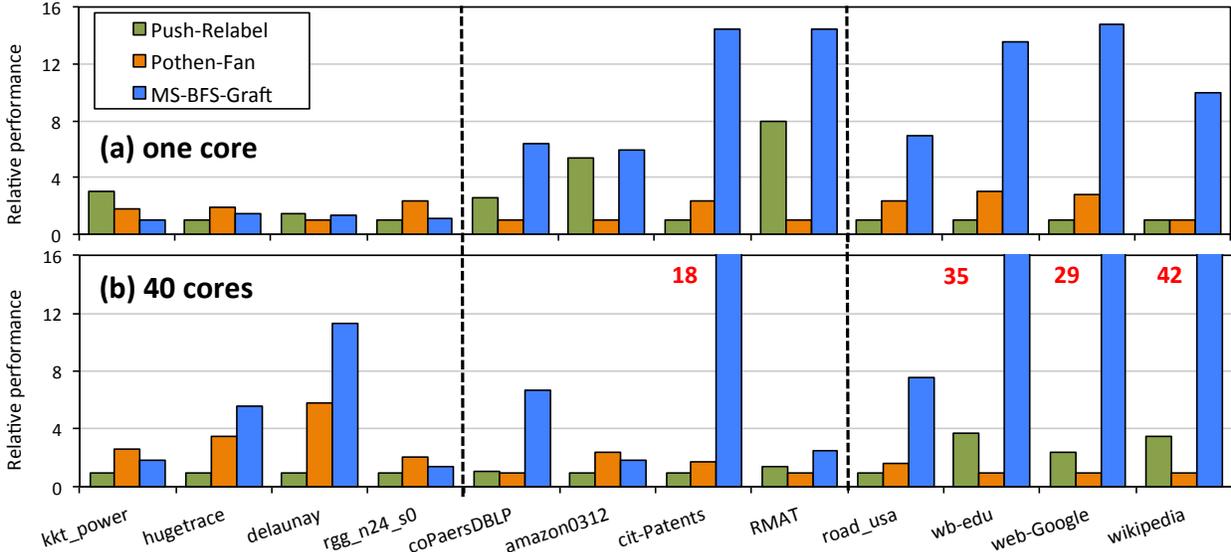

Fig. 3. Relative runtimes of MS-BFS-Graft, PR, and PF algorithms with respect to the slowest of these three algorithms on Mirasol using (a) one core and (b) 40 cores. In Subfig. (b), we cut the y-axis at 16 and show large values beside the bars. Dashed vertical lines separate different classes of graphs.

(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) scale-free: 7.1x to PR, 5.1x to PF and (3) networks: 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 scale free graphs (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.

B. 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. We run each algorithm 10 times for each input graphs 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 10 runs, to the mean of runtimes (μ): $\psi = \frac{\sigma}{\mu} \times 100$.

For all input graphs, 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%). In the MS-BFS-Graft algorithm, the granularity of work per thread is small; therefore, it can balance the work evenly among threads. By contrast, the DFS-based algorithm assigns each thread to explore a DFS tree, exposing a greater potential for load imbalance. The stable parallel performance of MS-BFS-Graft makes it an attractive candidate on future architectures.

C. Search rate

We report the search rate in millions of edges traversed per second (MTEPS), defined by the ratio of the number of traversed edges to the runtime. The MS-BFS-Graft algorithm traverses fewer edges at a faster rate relative to the DFS-based algorithms. Fig. 4 shows that MS-BFS-Graft searches at a rate 2-12 times faster than the PF algorithm for different input
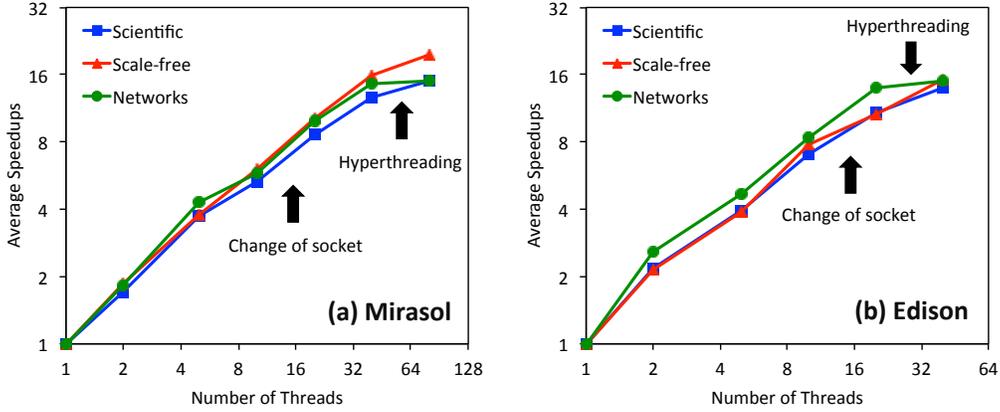
Fig. 5. Strong scaling of the MS-BFS-Graft algorithm for three classes of input graphs on (a) Mirasol and (b) Edison. For each class of graphs, we compute the speedups of individual graphs with respect to the serial MS-BFS-Graft algorithm and take the average.
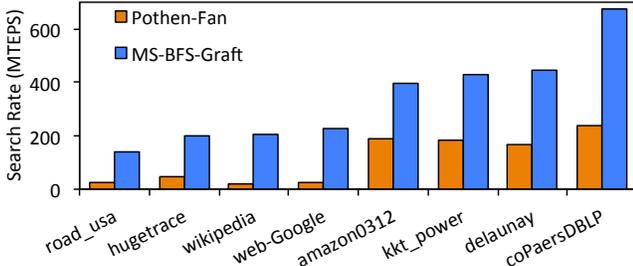

Fig. 4. Search rates of the MS-BFS-Graft and Pothen-Fan algorithms for different input graphs on Mirasol with 40 threads.

Fig. 6. Breakdown of time spent on different steps of the MS-BFS-Graft algorithm for different graphs.

graphs on Mirasol. The improvement is larger for graphs with low matching number, e.g., our algorithm searches 12x faster for `wikipedia` and 10x faster for `web-Google`.

The search rates of the matching algorithms are lower than those reported for direction-optimized BFS [9] for several reasons. First, the latter computes the search rate using the total number of edges in the graph (not the actual number of edges traversed) divided by the runtime. Second, matching algorithms use a specialized BFS, searching from only one part in a bipartite graph where the searches from the other part involve finding the unique mate of each vertex. Third, each phase of MS-BFS-Graft searches different subgraphs of the input, and these subgraphs get smaller in the course of the algorithm. Finally, the time taken by the augmentation step is also included in the search rate of the matching algorithm.

D. Scalability

Fig. 5 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. The results show good scaling on every class of graphs on these architectures. Using all available cores (without hyperthreading), the average speedup of problems in Table IV-A 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].

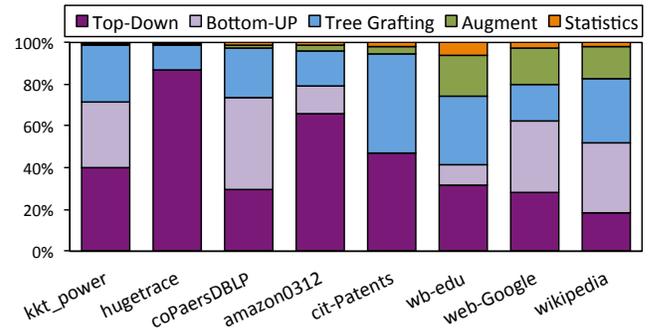In NUMA machines, scaling is more uniform within a single socket than across sockets. By using an efficient queue implementation (discussed earlier), we achieve excellent speedups on multiple sockets on both machines. On average (over all problem instances), we observe 22% performance improvement on 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. Hence, the MS-BFS-Graft algorithm is expected to scale better than its competitors on the future manycore systems with hardware threads.

E. Breakdown of runtime

Fig. 6 shows the breakdown of the 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 IV-A, at least 40% of the time is spent on the BFS traversal. However, graphs with high matching number (e.g., `hugetrace`, `kkt_power`) spend a larger proportion of total runtime on BFS traversal, whereas graphs
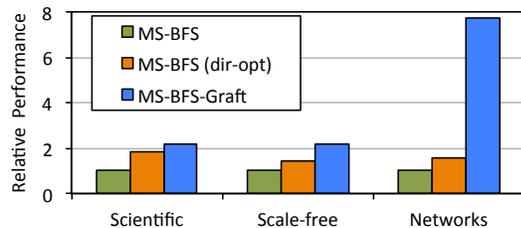
Fig. 7. Performance improvement of the parallel MS-BFS algorithm by direction-optimized BFS and tree-grafting on different classes of graphs.

with low matching number (e.g., `wb-edu`, `wikipedia`, etc.) spend more time on the augmentation and tree grafting steps.

F. Performance contributions

Fig. 7 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 low matching number benefit most from tree grafting, with 7.8x.

Fig. 8 shows the size of BFS frontiers at two phases of the MS-BFS and MS-BFS-Graft algorithms on `copaperDBLP`. 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.

VI. 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 low 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 low matching numbers. The newly developed MS-BFS-Graft algorithm scales up to 80 threads of an Intel multi-processor, 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 [20]. The MS-BFS-Graft algorithm employs level synchronous BFSs for which efficient distributed algorithms exist [21]. In 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 [22], and computing the block triangular form.
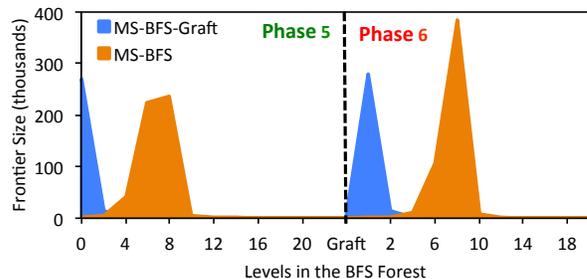

Fig. 8. The size of frontiers in the 5th and 6th phases (separated by a vertical line) of the MS-BFS and MS-BFS-Graft algorithms on `copaperDBLP`. MS-BFS-Graft employs tree-grafting between two consecutive phases.

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

[7] 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.

[8] 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.

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

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

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

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

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

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

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.