Distributed-Memory Algorithms for Maximum Cardinality Matching in Bipartite Graphs

Ariful Azad, Aydın Buluç
E-mail: azad@lbl.gov, abuluc@lbl.gov
Computational Research Division
Lawrence Berkeley National Laboratory

***Abstract*—We design and implement scalable distributed-memory algorithms for maximum cardinality matching in bipartite graphs. Computing matchings on distributed-memory supercomputers is challenged by the irregular and asynchronous data access patterns in graph searches and the difficulty in processing long paths passing through multiple processors. We address these challenges by developing an algorithm based on matrix algebra. We employ bulk-synchronous matrix algebraic modules to implement graph searches, and Remote Memory Access (RMA) operations to map asynchronous light-weight graph accesses. On real matrices, our algorithm achieves up to 18x speedup when we go from 24 cores to 2048 cores of a Cray XC30 supercomputer. Even higher speedups are obtained on larger synthetically generated graphs where our algorithms show good scaling on up to 12,000 cores.**

I. Introduction

Matching is a celebrated problem in combinatorial optimization with many applications [1]. A subset of the edges of a graph is said to be *a matching* if no two edges are incident to the same vertex. Finding matchings in a graph is often used as a preprocessing step for solving sparse systems of linear equations [2]. The increasing size of the sparse systems encouraged development of many distributed-memory solvers as large-scale problems do not fit into a single node. The lack of distributed-memory matching algorithms and implementations left the preprocessing step as a bottleneck. The current state of the practice [3], [4] involves gathering the data into a single page memory node to run the serial (or multithreaded) matching code, followed by a redistribution of the data for the rest of the solver to complete. The gathering can be impossible due to limited single node memory. Even when the problem fits into a single node, the gathering incurs expensive communication (discussed in Section VI-E) and subsequent single node execution of the matching algorithm creates a scalability bottleneck in Amdahl's terms due to significantly reduced concurrency within a single node. Therefore, scalable distributed-memory algorithms are needed to compute matchings in large distributed graphs.

This paper solely focuses on maximum cardinality matchings (MCM) in a bipartite graph, $G{=}(R, C, E)$, where the vertex set V is partitioned into two disjoint sets R and C, such that every edge connects a vertex in R to a vertex in C. In this paper, we denote the number of vertices by n and the number of edges by m. Furthermore, $|R|$ and $|C|$ are denoted by n_1 and n_2, respectively where $n_1{+}n_2{=}n$. MCM problem in bipartite graphs is known to be in random NC [5], meaning that a randomized parallel algorithm solves it in polylog time using number of processors that are polynomial-bounded in problem size. However, such algorithms often need excessive numbers of processors ($n^{6.5}$ in the aforementioned paper and m^3 using a deterministic interior-point method [6], albeit not in polylog time), and hence not work efficient. Moreover, none of these algorithms have been implemented in practice. Despite decades of active research, no published result achieves continuing speedups to thousands, or even hundreds of cores. This is an informal testimony to the hardness of parallelizing the MCM problem in practice.

In this paper, we have developed a distributed-memory MCM algorithm that employs breadth-first searches (BFS) to discover augmenting paths (paths in the graph that alternate between matched and unmatched edges with unmatched vertices as endpoints) from multiple unmatched vertices, and using these augmenting paths to increase the cardinality of the matching. This algorithm, called the multi-source BFS or MS-BFS, exposes more parallelism than its competitors and has been shown to be highly efficient in terms of runtime on shared-memory multiprocessors [7]. However, similar to other augmenting-path based MCM algorithms, the computational load of the MS-BFS algorithm is highly dynamic. The number of active vertices, i.e. the size of the frontier during augmenting path searches, changes dramatically as the number of unmatched vertices decreases during execution, posing a significant challenge to harnessing the parallelism available. We use sparse vectors to represent the frontier sets, hence ensuring work efficiency during this dynamic execution. We represent the input bipartite graph by a sparse matrix and decompose the MS-BFS algorithm into matrix algebraic modules. These modules are implemented using a handful set of bulk-synchronous matrix operations, and a sparse matrix-vector multiplication (SpMV) is at the heart of this matrix algebraic formulation. For efficient execution of lighter weight steps of the algorithm, which do not have enough bulk-synchronous parallelism to exploit, we revert to asynchronous remote memory access (RMA) operations. These modifications result in a highly-parallel MCM algorithm that scales up to thousands of cores on a modern supercomputer.

Our main contributions in this paper are as follows:

- We present a highly-parallel algorithm for MCM on distributed-memory system using matrix algebra.

- We present a step-by-step mapping between graph kernels used in traditional MCM algorithm and our matrix-algebraic modules, and provide a rigorous analysis of computation and communication complexity of the parallel algorithm.
- We provide a hybrid OpenMP-MPI implementation of the MCM algorithm that attains up to 18x speedup on real matrices when we go from 24 cores to 2048 cores of a Cray XC30 supercomputer. On synthetic graphs, our algorithm can compute MCM in graphs with 32 billion edges and scales up to 12,000 cores.

II. Preliminaries

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. Given a matching M in 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 an edge (u, v) is matched, we call u and v *mates* of each other. A matching M is *maximal* if there is no other matching M' that properly contains M. M is a *maximum* cardinality matching (MCM) if $|M|\geq|M'|$ for every matching M'. The *approximation ratio* of a maximal matching is the ratio of its cardinality to the cardinality of an MCM of the graph, which is always greater than or equal to $1/2$.

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 M by one (called 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 $|M|+|\mathbb{P}|$.

In this paper, we represent a bipartite graph $G = (R, C, E)$ by an $n_1 \times n_2$ binary sparse matrix $\mathbf{A} = \{a_{ij}\}$ with m nonzeros where the vertex sets R and C correspond to rows and columns respectively, and a_{ij} is nonzero when an edge joins the ith vertex in R with the jth vertex in C. In this context, R and C are called "row vertices" and "column vertices", respectively. Note that $\mathbf{A}$ is not the adjacency matrix of $G=(R, C, E)$ since it can be unsymmetric, rectangular (when $n_1\neq n_2$), and might have nonzero entries in the diagonal when there is an edge between ith row and ith column vertices. In this paper, we will occasionally drop the adjectives "bipartite" and "cardinality" when describing our methods.

A. Algorithmic variants for cardinality matching

MCM algorithms can be broadly categorized into (a) those based on augmenting paths and (b) those based on the push-relabel method [8], [9]. This paper primarily focuses 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 [10].

Augmenting path searching in a bipartite graph is more restricted than traditional graph searches because (a) searching begins and ends on unmatched vertices, (b) the paths alternate between matched and unmatched edges, and (c) vertices can be removed from future searches based on augmenting path discoveries. The search for augmenting paths can be performed form one unmatched vertex (Single Source or SS algorithms) or from all unmatched vertices simultaneously (Multi Source or MS algorithms). The search can be performed by using the alternating BFS, alternating depth-first search (DFS), or a combination of both BFS and DFS (the Hopcroft-Karp algorithm [11]). The Hopcroft-Karp algorithm has the best asymptotic complexity of $O(m\sqrt{n})$ whereas all other algorithms take $O(mn)$ time. However, specialized multi-source DFS (the Pothen-Fan algorithm [12]) and multi-source BFS (MS-BFS) algorithms are shown to outperform the Hopcroft-Karp algorithm on most practical graphs [13], [14] despite the latter's superior asymptotic complexity.

Previous work has demonstrated that an MCM can be computed more quickly if we initialize an MCM algorithm by a maximal matching with high approximation ratio [13], [15]. Maximal matching algorithms usually come in three flavors: (a) greedy, (b) Karp-Sipser [16], and (c) dynamic mindegree. All three algorithms take $O(m)$ time, and they differ from one another based on the processing order of unmatched vertices. Even though sequential Karp-Sipser achieves higher approximation ratio than greedy and dynamic mindegree on most practical graphs [13], [15], we demonstrate that it is too expensive to maintain the dynamic order of vertices needed by Karp-Sipser on distributed memory. Section VI-A discusses the impact of maximal matching algorithm on distributed-memory MCM algorithms.

B. Previous work on parallel cardinality matching

Recents efforts in parallel matching algorithms primarily focused on shared-memory. Shared-memory implementations of Pothen-Fan, push-relabel, and MS-BFS demonstrate good scaling on multithreaded multiprocessors [8], [14], [17] and on GPU [18]. A recently developed MS-BFS-Graft algorithm [7] that employs a tree-grafting method eliminating most of the redundant edge traversals is shown to be one of best performers on modern multicore and manycore systems.

We are aware of only three results on distributed-memory cardinality matching algorithms, with only one of them solving the MCM problem. Langguth *et al.* [19] developed a distributed memory push relabel algorithm for MCM. However, their algorithm did not scale beyond 64 processors because of the difficulty in parallelizing "push" and "relabel" operations needed by the algorithm. Patwary *et al.* [20] implemented a parallel Karp-Sipser algorithm (in a general graph) on a distributed memory machine using an edge partitioning of the graph. On some real graphs, their algorithm achieved up to $38\times$ speedups on 64 processors, whereas on other graphs their algorithm did not scale at all. Finally, in a recent work [21],

Algorithm 1 MS-BFS algorithm. **Input:** A bipartite graph $G(R, C, E)$, an initial matching M. **Output:** A maximum cardinality matching M.

```
1: procedure MS-BFS(G(R, C, E), M)
2:     repeat                                   ▷ a phase of the algorithm
3:         f_c ← unmatched vertices in C        ▷ Initial column frontier
4:         ℙ ← φ                                ▷ Set of vertex-disjoint augmenting paths
5:         while f_c ≠ φ do                     ▷ an iteration in the current phase
6:             discover unvisited neighbors f_r from f_c
7:             add newly discovered augmenting paths into ℙ
8:             create next frontier f_c from the mates of matched vertices in f_r
9:         M ← M ⊕ ℙ                            ▷ augment matching by augmenting paths
10:    until an augmenting path is discovered in the current phase
```

we implemented three variants of maximal matching algorithm using matrix algebra with good scaling on up to tens of thousands of processors.

III. MCM ALGORITHM USING MATRIX ALGEBRA

A. Selecting an algorithm exposing massive parallelism

A matching algorithm that could scale to thousands of cores must expose massive parallelism and light-weight inter-processor communication. Since single-source algorithms process one unmatched vertex at a time, they do not qualify for massive parallelization. Algorithms that rely on DFS such as Pothen-Fan and Hopcroft-Karp bear less potential on higher concurrency because these algorithms employ coarse-grained parallelism, and DFS is difficult to parallelize [22]. By contrast, MS-BFS exposes massive fine-grained parallelization and structured bulk-synchronous communication patterns, which are essential to attain high performance on distributed-memory systems. Furthermore, the ability to map BFS to matrix-algebraic functions [23] makes MS-BFS even more attractive on higher concurrency. Hence, we chose MS-BFS for distributed-memory parallelization.

Algorithm 1 describe a high-level skeleton of a level-synchronous MS-BFS algorithm. The **repeat-until** block defines a phase of the algorithm where it searches for a set of vertex-disjoint augmenting paths from all unmatched vertices in C. Each phase is further divided into several level-synchronous iterations defined by the **while** loop in Algorithm 1. Each iteration starts with a column frontier f_c, discovers unvisited neighbors f_r (called row frontier) from f_c, saves new augmenting paths (if any), and creates the next frontier f_c from the mates of matched vertices in f_r. At the end of a phase, the algorithm augments the current matching by the newly discovered augmenting paths. If no augmenting path is found in the latest phase, the algorithm returns with a maximum matching.

B. Decomposing the MS-BFS algorithm

We decompose each iteration of Algorithm 1 into small modules and map them to matrix-algebraic operations. The leftmost column in Figure 1 shows detailed steps of a single iteration of Algorithm 1. Considering an initial matching shown in the bipartite graph in Figure 2, the middle two columns in Figure 1 illustrate the graph and vector representations of row and column vertices in the first iteration of Algorithm 1. The rightmost column in Figure 1 shows the matrix-algebraic functions whose details are described in Table I. At first we discuss the matrix algebraic notations, followed by the detail description of the algorithmic steps in terms of matrix and vector operations.

Representing vertex sets via vectors. We use either a dense or a sparse vector to represent a set of vertices. The difference between these two formats is that the latter does not explicitly store the nonzero entries. Given a sparse vector x, $nnz(x)$ denotes the number of nonzeros and $len(x)$ denotes the number of both zero and nonzero entries in x. We use subscripts r and c to denote vectors of row and column vertices, respectively.

The MS-BFS algorithm keeps track of both parent and root of each vertex in the current row and column frontiers. Hence, we represent each vertex by a *(parent, root)* pair and denote it by VERTEX data structure. We create a vertex with parent p and root r by calling VERTEX(p, r). For a sparse vector x of VERTEX objects, PARENT(x) and ROOT(x) return *parents* and *roots* of all vertices in x. In the first iteration of a phase, parent and root of a vertex are set to itself. While the parent of a vertex is updated in every iteration, roots are simply passed from parents to children. In our example in Figure 1, the iteration starts with all unmatched column vertices, i.e., the initial column frontier is $f_c = \{c_1, c_2, c_5\}$. In vector terms, f_c is stored in a sparse vector of length five with nonzeros in 1st, 2nd and 5th locations, i.e., $f_c = [(1, 1), (2, 2), -, -, (5, 5)]$, where "$-$" denotes a zero entry in the sparse vector. We store the mates of row and columns vertices in two dense vectors $mate_r$ and $mate_c$. If the ith row vertex is matched to the jth column vertex, then $mate_r[i]=j$ and $mate_c[j]=i$ (-1 denotes unmatched vertices). A dense vector π_r stores the parents of visited row vertices in the current phase (-1 denotes unvisited vertices). Another dense vector $path_c$ stores the start column vertices (as indices) and end row vertices (as values) of augmenting paths, i.e., $path_c[i]=j$ denotes an augmenting path from the ith column vertex to the jth row vertex.

Neighborhood exploration by SpMV (Step 1). We explore vertices from one side of a bipartite graph to the other side by using SpMV over a semiring. For the purposes of this work, a semiring is defined over (potentially separate) sets of 'scalars', and has its two operations 'multiplication' and 'addition' redefined. We refer to a semiring by listing its scaling operations, such as the *(multiply, add)* semiring. The usual semiring multiply for BFS is *select2nd*, which returns the second value it is passed. The BFS semiring is defined over two sets: the matrix elements are from the set of binary numbers whereas the vector elements are from the set of integers. This usage of a semiring is more general than the definition employed in mathematics and it is studied under the name of "heterogenous algebras" [24].

Since the frontier stores *(parent, root)* pairs, our semiring 'addition' operates either on parents or roots of a pair of

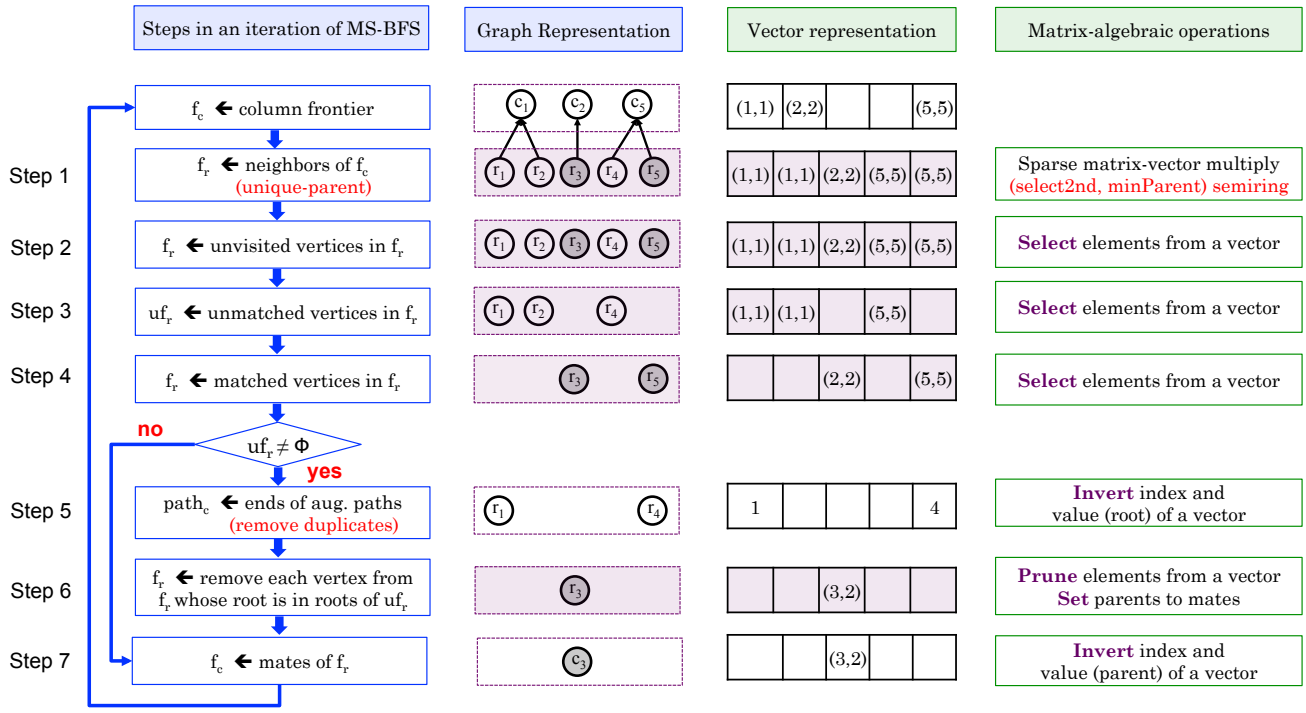

Fig. 1: The leftmost column decomposes an iteration of the MS-BFS algorithm, and the rightmost column maps each step to a matrix-algebraic function. Considering an initial matching shown in Figure 2, each row of the 2nd and 3rd columns illustrates the graph and vector representations of row and column vertices at the end of the graph/matrix operation specific to that row. In all steps except Step 5, the output is either a subset of row or column vertices where matched and unmatched vertices are shown in filled and empty circles, respectively. In the vector representation in the 3rd column, a nonzero entry in the ith location denotes the presence of ith row/column vertex and the value is $(parent, root)$ pair of the corresponding vertex. Subset of row vertices are marked in dark shade. In Step 5, we mark the augmenting paths by a dense vector $path_c$ whose indices are roots (start vertices) and values are unmatched leaves (end vertices) of augmenting paths.

TABLE I: Basic functions needed for the cardinality matching algorithm.

Function	Arguments	Returns	Example (x: sparse, y:dense, q: sparse)	Serial Complexity	Communication
IND	x: a sparse vector	local indices of nonzero entries of x	$x = [3, 0, 2, 2, 0]$ $\text{IND}(x) = [1, 3, 4]$	$O(nnz(x))$	None
SELECT	x: a sparse vector y: a dense vector $expr$: logical expr. on y assume $size(x) = size(y)$	$z \leftarrow$ an empty sparse vector for $i \in \text{IND}(x)$ **if** $(expr(y[i]))$ **then** $z[i] \leftarrow x[i]$	$x = [3, 0, 2, 2, 0]$ y=[1, -1, -1, 2, -1] $\text{SELECT}(x, y$ = -1$) = [0, 0, 2, 0, 0]$	$O(nnz(x))$	None
SET	x: a sparse vector y: a dense vector	for $i \in \text{INDEX}(x)$ $y[i] \leftarrow x[i]$	x=[3, 0, 2, 2, 0], y=[1, -1, 1, 2, -1] $\text{SET}(y, x)$=[3, -1, 2, 2, -1]	$O(nnz(y))$	None
INVERT	x: a sparse vector assume $\max(x) \leq len(x)$	$z \leftarrow$ an empty sparse vector for $i \in \text{IND}(x)$ **if** $(z[x[i]] \neq 0)$ **then** $z[x[i]] \leftarrow i$	x=[3, 0, 2, 2, 0] $\text{INVERT}(x)$=[0, 4, 1, 0, 0]	$O(nnz(x))$	AllToAll
PRUNE	x: a sparse vector q: a sparse vector	$z \leftarrow$ an empty sparse vector for $i \in \text{INDEX}(x)$ **if** $x[i] \notin q$ **then** $z[i] \leftarrow x[i]$	x=[0, 0, 5, 0, 2] q=[2, 0, 0, 4, 1] $\text{PRUNE}(x, q)$=[0, 0, 5, 0, 0]	$O(sort(\psi) + \mu \log \psi)$ where $\mu \leq \psi$, $\psi = nnz(x)$, $\mu = nnz(q)$	AllGather
SPMV	$\mathbf{A}$: a sparse matrix x: a sparse vector SR: a semiring	returns $\mathbf{A} \cdot x$	see Fig. 2	$\sum_{k \in \text{IND}(x)} nnz(\mathbf{A}(:, k))$	AllGather (row/column process grid)

vertices. In our example in Fig. 1, we used 'minParent' as 'addition' operation, which selects the value with the minimum parent index. Fig. 2 shows the execution of the SpMV $\mathbf{A} \cdot f_c$ over the *(select2nd, minParent)* semiring. The *(select2nd, minParent)* semiring can be replaced by *(select2nd, minRoot)* or *(select2nd, randRoot)* semirings, which retain value with the minimum or random roots, respectively. *(select2nd, randRoot)* semiring is useful to randomly distribute vertices among alternating trees, ensuring better balance of tree sizes.

Selecting subset of vertices (Steps 2, 3, 4). Steps 2, 3, 4 in Fig. 1 select subset of vertices from f_r based on the status (e.g, visited or matched) of the vertices, which is performed

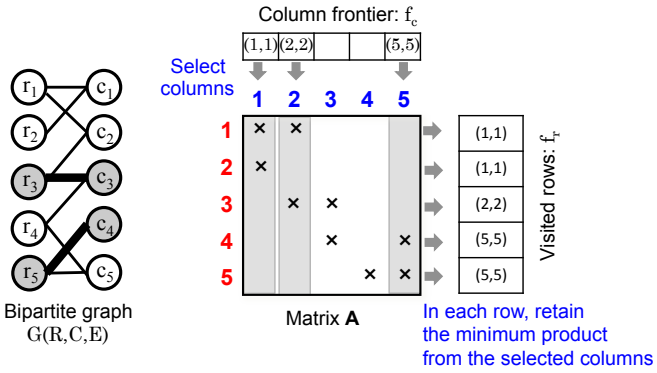

Fig. 2: Traversing a bipartite graph $G(R, C, E)$ via SpMV. The bipartite graph with five row and five column vertices is shown in the left. Matched and unmatched vertices are shown in filled and empty circles, respectively. Thin lines represent unmatched edges and thick lines represent matched edges. The binary matrix $\mathbf{A}$ represents the bipartite graph where an "×" denotes an edge in G. f_c represents the set of unmatched column vertices whose value denotes the $(parent, root)$ pairs. The sparse matrix-vector multiplication $\mathbf{A} \cdot f_c$ over the *(select2nd, minParent)* semiring first selects columns that have nonzeros in f_c (shown in gray) and then in each row, retains the value with the minimum parent from the selected columns. The indices of the result vector f_r denote row vertices explored from f_c and the value $f_r[i]$ denotes its parent and root. Image modified from our prior work [21].

by the SELECT operation in Table I. Given a sparse vector x, a dense vector y and a logical expression $expr$, the function SELECT$(x, y, expr)$ selects indices I of y where $expr(y)$ is true and returns values of the selected index. As shown in the pseudocode in Table I, SELECT only iterates on the sparse vector, hence the complexity $O(nnz(x))$.

Inverted index (Steps 5, 7). At the end of Step 4, if the algorithm discovers some unmatched vertices in f_r, it saves the newly discovered augmenting path in the $path_c$ dense vector. This step is performed by storing the newly unmatched row vertices in a sparse vector uf_r. An augmenting path from the column vertex ROOT$(uf_r[i])$ to the ith row vertex is discovered when $uf_r[i] \neq 0$. Since we keep track of augmenting paths in $path_c$ that is indexed by the roots, we compute an inverted index on uf_r, where the roots become indices in $path_c$ and the nonzero indices of uf_r become the values in the corresponding locations. If more than one augmenting path is discovered starting from the same root, we keep only one of them. The same operation is used in Step 7 to go from matched row vertices to their mates.

The inverted index is computed by the INVERT operation in Table I. Given a sparse vector x, the INVERT function returns the inverted index by swapping the indices and values of nonzero entries in x and stores the results in a new sparse vector z. When the nonzero values of x are $(parent, root)$ pairs, we specify whether parents or roots are to be swapped with the indices. If x has repeated nonzero values, only one of them is used as index in z (we keep the first index). The serial complexity of this operation is $O(nnz(x))$.

Prune vertices in trees yielding augmenting paths (Step 6). To avoid unnecessary work in expanding a tree that has already discovered an augmenting path, we prune vertices belonging to augmenting-path-yielding trees from the next frontier. If f_r and uf_r be the subsets of matched and unmatched vertices in the current row frontier, we remove vertices from f_r whose roots are present in the roots of uf_r. This step is performed by the PRUNE operation from Table I. Let $\psi = nnz(f_r)$ and $\mu = nnz(uf_r)$. Since the roots of f_r and uf_r are not sorted, the serial complexity of this operation is

$$T_{prune} = \min\big(sort(\psi) + \mu \log \psi, sort(\mu) + \psi \log \mu\big).$$

Construct next column frontier (Step 7). In the last step of an iteration, the algorithm constructs the column frontier f_c for the next iteration from the mates of pruned row frontier f_r. The step can be performed by setting the parents of f_r to their mates and calling INVERT(f_r) with parents (i.e., mates) of f_r as the indices of the new frontier.

C. The algorithm

We describe the complete matrix-based MCM algorithm in MCM-MATCH function in Algorithm 2. We pass the graph in its sparse matrix representation and initial matchings as two dense vectors $mate_r$ and $mate_c$. The algorithm returns the maximum cardinality matching by updating $mate_r$ and $mate_c$. At the beginning of each phase, MCM-MATCH initializes the parent and augmenting path vectors. For each unmatched column vertex c_i, the column frontier f_c has a nonzero entry in the ith location with both the parent and root pointing to the vertex itself. Algorithm 2 then iteratively performs seven steps discussed in detail in the previous subsection.

In every iteration of Algorithm 2, a row vertex r_i in f_r is uniquely associated to a parent c_j in f_c, and c_j is associated with its unique mate. Hence, each phase of MCM-MATCH maintains a unique alternating path from a row vertex in f_r to its root following the chain of parents and mates. Since roots are inherited from parents, all alternating trees created in a phase are vertex disjoint. The set of augmenting paths identified in a phase is vertex disjoint as well because we keep at most one augment path in each alternating tree.

The AUGMENT function in Algorithm 3 describes the process of augmenting a matching by a set of vertex disjoint augmenting paths using our matrix-algebraic notations. We start with a sparse vector v_c created from $path_c$ by removing entries with -1 and start augmenting all paths simultaneously from their end vertices. Every iteration of the **while** loop in line 3 of Algorithm 2 matches a pair of vertices from each augmenting path by employing two INVERT operations: the first call is used to jump from row vertices to their parents and the second call is used to jump from parents to the mates of parents. Let h be the longest augmenting path discovered in a phase of MCM-MATCH . Then AUGMENT requires $O(h/2)$ level-synchronous iterations to augment matching by all paths.

Algorithm 2 Maximum cardinality matching algorithm based on matrix algebra. **Inputs:** A binary $n_1 \times n_2$ sparse matrix $\mathbf{A}$ denoting a bipartite graph $G(R, C, E)$ where $|R|$=n_1, $|C|$=n_2, and $|E|$= $nnz(A)$=m. Dense vectors $mate_r$, and $mate_c$ store the initial mates of row and column vertices (-1 for unmatched vertices). **Output:** Updated $mate_r$ and $mate_c$ with an MCM.

procedure MCM-DIST($\mathbf{A}$, $mate_c$, $mate_r$)
  repeat ▷ a phase of the algorithm
    $\pi_r \leftarrow$ -1 ▷ Initialize a dense vector storing the parents of row vertices visited in this phase
    $path_c \leftarrow$ -1 ▷ Initialize a dense vector storing the end points of augmenting paths
    $f_c \leftarrow$ an empty sparse vector of size n of type VERTEX
    for $i \in$ IND($mate_c$) **do** ▷ Initial column frontier from unmatched column vertices
      if $mate_c[i] =$ -1 **then**
        $f_c[i] \leftarrow$ VERTEX(i, i) ▷ *parent* and *root* are set to itself
    while $f_c \neq \phi$ **do** ▷ an iteration in the current phase
      ▷ **Step 1: Explore neighbors of column frontier (one step of BFS)**
      $f_r \leftarrow$ SPMV($\mathbf{A}, f_c$, SR=(select2nd, minParent))
      ▷ **Step 2, 3, 4: Select unvisited, matched, and unmatched row vertices**
      $f_r \leftarrow$ SELECT($f_r, \pi_r =$ -1) ▷ Keep unvisited rows
      $\pi_r \leftarrow$ SET(π_r, PARENT(f_r)) ▷ Set parents of newly visited rows
      $uf_r \leftarrow$ SELECT($f_r, mate_r =$ -1) ▷ Unmatched row vertices in the row frontier
      $f_r \leftarrow$ SELECT($f_r, mate_r \neq$ -1) ▷ Keep matched row vertices in the row frontier
      if $uf_r \neq \phi$ **then** ▷ At least one augmenting path is found
        ▷ **Step 5: Store endpoints of newly discovered augmenting paths**
        $t_c \leftarrow$ INVERT(ROOT(uf_r)) ▷ Roots and leaves become indices and values, respectively
        $path_c \leftarrow$ SET($path_c, t_c$) ▷ Save end vertices of augmenting paths
        ▷ **Step 6: Prune vertices in trees yielding augmenting paths**
        $f_r \leftarrow$ PRUNE(f_r, ROOT(uf_r)) ▷ Remove vertices from f_r whose roots are in the roots of uf_r
      ▷ **Step 7: Construct next frontier**
      SET(PARENT(f_r), $mate_r$) ▷ Parents are set to mates
      $f_c \leftarrow$ INVERT(f_r) ▷ Use *parent* of f_r as index of f_c
    ▷ **Step 8: Augment matching by all augmenting paths discovered in this phase**
    AUGMENT($path_c$, π_r, $mate_r$, $mate_c$)
  until an augmenting path is discovered in the current phase

IV. DISTRIBUTED MEMORY ALGORITHM

A. Data distribution and storage

We use the CombBLAS framework [25] which distributes its sparse matrices on a 2D $p_r \times p_c$ processor grid. Processor $P(i, j)$ stores the submatrix $\mathbf{A}_{ij}$ of dimensions $(m/p_r) \times (n/p_c)$ in its local memory. The CombBLAS uses the doubly compressed sparse columns (DCSC) format to store its local submatrices for scalability, and uses a vector of {index, value} pairs for storing sparse vectors. To balance load across processors, we randomly permute the input matrix $\mathbf{A}$ before running the matching algorithms.

Vectors are also distributed on the same 2D processor grid. For a distributed vector v, the syntax v_{ij} denotes the local n/p length piece of the vector owned by the $P(i, j)$th processor. The syntax v_i denotes the hypothetical n/p_r or n/p_c length piece of the vector collectively owned by all the processors along the ith processor row $P(i, :)$ or column $P(:, i)$.

B. Analysis of the distributed algorithm

We measure communication by the number of *words* moved (W) and the number of *messages* sent (S). The cost of communicating a length m message is $\alpha + \beta m$ where α is the latency and β is the inverse bandwidth, both defined relative to the cost of a single arithmetic operation. Hence, an algorithm that performs F arithmetic operations, sends S messages, and moves W words takes $T = F + \alpha S + \beta W$ time.

Algorithm 3 Augment a matching by a set of vertex disjoint augmenting paths. **Inputs:** $path_c$ stores the end vertices of augmenting paths, and π_r stores the parents of row vertices. $mate_r$, and $mate_c$ store the mates of row and column vertices, respectively. All input vectors are dense where -1 represents missing values. **Output:** Updated $mate_r$ and $mate_c$.

procedure AUGMENT($path_c$, π_r, $mate_r$, $mate_c$)
  $v_c \leftarrow$ sparse vector from $path_c$ by removing entries with -1
  while $v_c \neq \phi$ **do**
    $v_r \leftarrow$ INVERT(v_c)
    $v_r \leftarrow$ SET(v_r, π_r) ▷ Set values with parents
    $v_c \leftarrow$ INVERT(v_r)
    $v_c' \leftarrow$ SET($v_c, mate_c$) ▷ Set values with mates
    $mate_c \leftarrow$ SET($mate_c, v_c$) ▷ Update mates
    $mate_r \leftarrow$ SET($mate_r, v_r$)
    $v_c \leftarrow v_c'$

We previously analyzed [21] the complexity of SpMV and INVERT operations because they are also building blocks of parallel maximal matching algorithms. Here, we expand beyond that per-iteration analysis. Since the load is extremely dynamic across iterations, we analyze the aggregate cost over all iterations in a phase. For ease of analysis, we assume that nonzeros are i.i.d. distributed in matrices and vectors. We also assume a square processor grid p_c=p_r=$\sqrt{p}$. Number of iterations is denoted by $|\text{iters}|$.

We leverage the 2D SpMV algorithms implemented in CombBLAS; both for sparse and dense vectors. 2D SpMV

algorithms has two communication phases [26]: (1) "expand", for which CombBLAS uses the allgather primitive, and (2) "fold", for which CombBLAS uses the personalized all-to-all primitive [27]. The parallel SpMV algorithm is work efficient, hence its total work is the same as its serial complexity. For graphs without good separators, the communication volume in the fold phase can be as high as the arithmetic cost. Each edge is traversed at most once within a phase in most cardinality matching algorithms, including ours. Hence the overall per-process cost of SpMV within a phase is at most the cost of one full BFS [23]:

$$T_{\text{SpMV}} = O\Big(\frac{m}{p} + \beta\big(\frac{m}{p} + \frac{n}{\sqrt{p}}\big) + |\text{iters}|\,\alpha\sqrt{p}\Big)$$

INVERT is called twice at each iteration. Once on uf_r, the set of unmatched vertices in the current frontier, and once on f_r, the set of matched vertices in the current frontier. Since the second call populates the next frontier, the total number of nonzeros inverted is exactly the same as the sum of the frontier sizes over all iterations, which is $O(n)$. Hence the per-process cost of INVERT within a phase is

$$T_{\text{INVERT}} = O\Big(\frac{n}{p} + \beta\frac{n}{p} + |\text{iters}|\,\alpha p\Big)$$

using personalized all-to-all. This makes INVERT's latency cost higher by a factor of $\sqrt{p}$, hence making it the potential bottleneck in a strong scaling regime where the latency term dominates. In the weak scaling regime, by contrast, SPMV is likely to be the bottleneck.

PRUNE gathers roots of unmatched vertices uf_r on all processors in its communication step. As before, let $\psi = nnz(f_r)$ and $\mu = nnz(uf_r)$. Assuming that the matched vertices in the current row vertices f_r are uniformly distributed across processors, per processor computation for PRUNE is

$$\min\big(sort(\frac{\psi}{p}) + \mu\log\frac{\psi}{p}, sort(\mu) + \frac{\psi}{p}\log\mu\big).$$

The communication cost to gather uf_r is $\alpha p + \beta\mu$ per processor, using the ring algorithm [28]. However, a vertex can appear at most once in uf_r in a phase because once an unmatched vertex is found, it either becomes an end vertex of an augmenting path or gets pruned if another augmenting path is discovered in the same tree. Hence, $\sum nnz(uf_r)$ summed over all iterations in a phase is bounded by $O(n)$. In practice, it is much smaller than n because the vertices matched by initial maximal matching and in previous phases of the MCM algorithm do not appear in uf_r in a phase. Therefore, the bandwidth cost for PRUNE is usually insignificant to that of SpMV. Furthermore, in contrast to INVERT, pruning is only required in a small fraction of all iterations where an augmenting path is discovered. Hence, total latency cost of PRUNE is much smaller than INVERT.

We now discuss the communication cost of augmenting a matching by k vertex-disjoint augmenting paths described in Algorithm 3. We call this algorithm level-parallel since the augmentation proceeds level by level starting from the bottom of the paths. For simplicity, we assume that k paths are uniformly distributed across p processors, i.e., each processor owns k/p vertices in each level of the paths. Let h be the length of the longest augmenting path. Each iteration of Algorithm 3 needs two INVERT operations, each of which requires two personalized all-to-all to communicate the indices and values of the sparse vector, and another personalized all-to-all to communicate the amount of data to be sent to different processors. Hence, the communication cost of Algorithm 3 is $h(6\alpha p + 4\beta\frac{k}{p} + 2\beta p)$ per processor.

Algorithm 4 Augment a matching by a set of vertex disjoint augmenting paths in a path-parallel fashion.

```
1: procedure AUGMENT_PATH( path_c, parent_r, mate_r, mate_c)
2:     for v ∈ path_c in parallel do
3:         while v ≠ −1 do
4:             u ← parent_r[v]                    ▷ MPI_GET
5:             v' ← mate_c[u]                     ▷ MPI_GET
6:             mate_c[u] ← v                      ▷ MPI_PUT
7:             mate_r[v] ← u                      ▷ MPI_PUT
8:             v ← v'
```

When k is small, which is often the case in later phases of the MCM algorithm, the latency term dominates the runtime of AUGMENT, hindering scaling to higher concurrencies. Hence, we developed another variant of augmentation that processes each augmenting path independently by using MPI RMA operations. This "path-parallel" augmentation is described in Algorithm 4 where each processor augments k/p augmenting paths asynchronously by directly manipulating the $mate$ vectors in remote processors. Each iteration of the **while** loop in Algorithm 4 uses two MPI_GET and MPI_PUT operations. However, lines 5 and 6 can be merged into a single MPI_FETCH_AND_OP requiring a total 3 RMA calls per processor per iteration. Since these RMA calls are only communicating the mate/parent information from dense vectors, the communication cost per processor per iteration is $3(\alpha + \beta)$. Thus, the total communication cost of Algorithm 4 is $\frac{k}{p}(3h\alpha + 3h\beta)$ per processor. Comparing the latency terms of Algorithm 3 and Algorithm 4, the path parallel augmentation performs better when the number of augmenting paths $k < 2p^2$. Therefore, we use this criterion to automatically switch between these two variants of augmentations and were able to reduce the augmentation time significantly.

V. EXPERIMENTAL SETUP

A. Platform

We evaluate the performance of parallel MCM algorithm on Edison, a Cray XC30 supercomputer at NERSC. In Edison, nodes are interconnected with the Cray Aries network using a Dragonfly topology. Each compute node is equipped with 64 GB RAM and two 12-core 2.4 GHz Intel Ivy Bridge processors, each with 30 MB L3 cache. We used Cray's MPI implementation, which is based on MPICH2. We used OpenMP for intra-node multithreading and compiled the code with gcc 5.2.0 with `-O2 -fopenmp` flags. To ensure better memory affinity to NUMA nodes of Edison, we used `-cc numa_node` option when submitting jobs. In our experiments, we only used square process grids because rectangular grids are not supported in CombBLAS [25]. When p cores

TABLE II: Real and synthetic problems for evaluating the maximum matching algorithm. For symmetric matrices, we store edges in both directions. Hence we report the number of nonzeros of symmetric problems to be about twice as large as the number of edges reported in the Florida sparse matrix collection [29].

Class	Graph	#Rows (n_1) ($\times 10^6$)	#Columns (n_2) ($\times 10^6$)	nnz ($\times 10^6$)	Description
Unsymmetric	IMDB	0.43	0.43	3.78	IMDB movie/actor network
	amazon-2008	0.74	0.74	5.16	Amazon book similarity network
	GL7d19	1.91	1.96	37.32	combinatorial problem
	wikipedia-20070206	3.56	3.56	45.03	Wikipedia page links
	circuit5M	5.56	5.56	59.52	Large circuit from Freescale Semiconductor
	ljournal-2008	5.36	5.36	79.02	LiveJournal social network
	cage15	5.15	5.15	99.20	3D engine fan
	HV15R	2.02	2.02	283.1	DNA electrophoresis, 15 monomers in polymer
Symmetric	hugetrace-00020	16.00	16.00	96.00	Frames from 2D Dynamic Simulations
	road_usa	23.94	23.94	115.42	USA street networks
	dielFilterV3real	1.10	1.10	178.62	High-order vector finite element method in EM
	delaunay_n24	16.77	16.77	201.33	Delaunay triangulations of random points
	nlpkkt200	16.24	16.24	896.40	Symmetric indefinite KKT matrix
Random	ER-30	1073.7	1073.7	32044	Erdős-Rényi random graphs
	G500-30	1073.7	1073.7	32044	Graph 500 graphs [30]
	SSCA-30	1073.7	1073.7	17085	Matrices generated by (SSCA#2) benchmark [31]

are allocated for an experiment, we create a $\sqrt{p/t} \times \sqrt{p/t}$ process grid where t is the number of threads per process. Unless otherwise stated, all of our experiments used 12 threads per MPI process and each MPI process was placed on a socket in a node of Edison. In our hybrid OpenMP-MPI implementation, all MPI processes perform local computation followed by synchronized communication rounds. Local computation in every matrix-algebraic kernel described in Table I is fully multithreaded using OpenMP. Only one thread in every process makes MPI calls in the communication rounds (that is `MPI_THREAD_FUNNELED` thread support is used in `MPI_Init_thread` function call). The source code of the distributed-memory MCM algorithm is publicly available as part of Combinatorial BLAS library [32].

B. Input Graphs

Table II describes a set of real matrices from the University of Florida sparse matrix collection [29] used in our experiments. We selected the largest unsymmetric and symmetric matrices that have at least several thousands of unmatched vertices after computing a maximal matching. To test the performance of our matching algorithm on larger matrices, we used RMAT [33], the Recursive MATrix generator to generate three different classes of synthetic matrices: (a) G500 matrices representing graphs with skewed degree distributions from Graph 500 benchmark [30], (b) SSCA matrices from HPCS Scalable Synthetic Compact Applications graph analysis (SSCA#2) benchmark [31], and (c) ER matrices representing Erdős-Rényi random graphs with uniform degree distributions. We use the following RMAT seed parameters to generate these matrices: (a) a=.57, b=c=.19, and d=.05 for G500, (b) a=.6, and b=c=d=.4/3 for SSCA, and (c) a=b=c=d=.25 for ER. A scale n synthetic matrix is 2^n-by-2^n. On average, G500 and ER matrices have 32 nonzeros, and SSCA matrices have 16 nonzeros per row and column. For example, a scale-30 G500 matrix (`G500-30`) has about 1 billion rows, 1 billion columns, and 32 billion nonzeros. We applied a random permutation to the input matrices to balance the memory and the computational load.

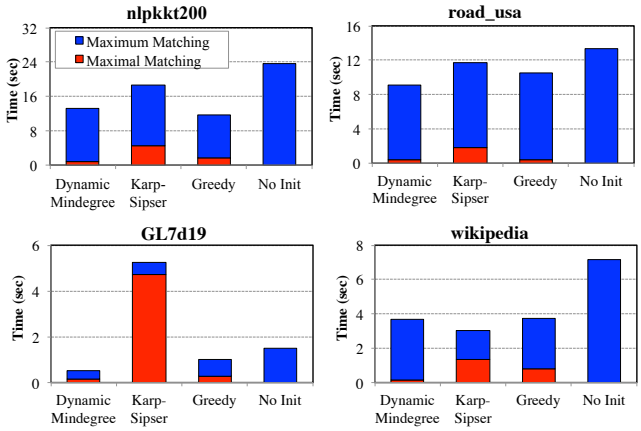

Fig. 3: Effect of distributed-memory maximal matching algorithms on the time to compute MCM on 1024 cores of Edison.

VI. Results

A. Selecting an initial maximal matching

The total runtime of an MCM algorithm often decreases when it is initialized by a maximal matching with high approximation ratio [13], [14], [15]. In our prior work [21], we developed distributed-memory Karp-Sipser, dynamic mindegree and greedy algorithms using a subset of the matrix-algebraic primitives described in Table I. We observed that on distributed memory, especially on high concurrency, the Karp-Sipser algorithm that is usually the best performer on shared-memory becomes much slower than greedy and dynamic mindegree. In Fig. 3, we demonstrate the impact of distributed maximal matching algorithms on MCM for four representative graphs. On these four graphs, Karp-Sipser is always slower than greedy and dynamic mindegree algorithms. However, the better approximation ratio obtained by Karp-Sipser may offset its slow runtime as can be seen for `wikipedia`. In fact, `wikipedia` and `cage15` are only two matrices in Table II

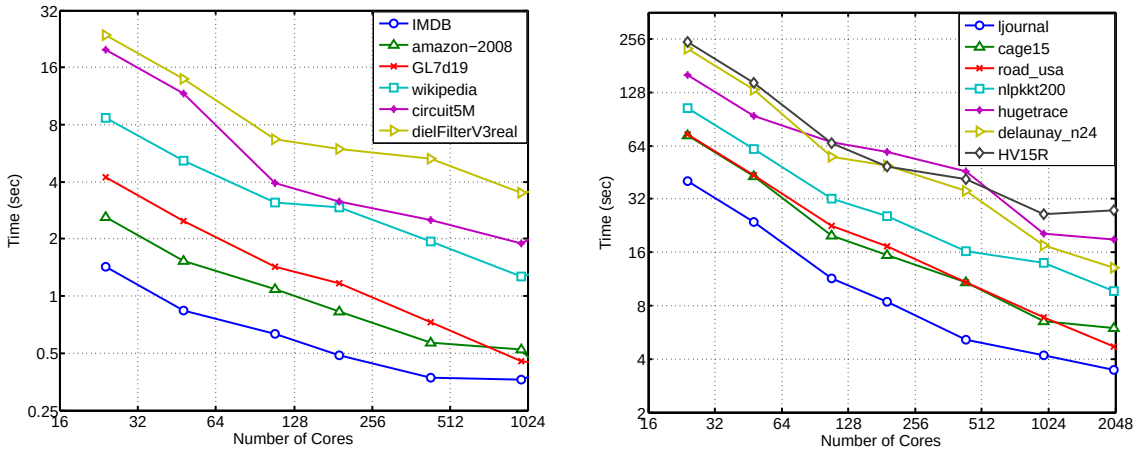

Fig. 4: Strong scaling of MCM-DIST when computing maximum matching on real matrices on Edison. The left and right subfigures show the scaling of relatively smaller and larger matrices, respectively. 12 threads are used on all concurrencies except on 24 cores where each process on a 2×2 grid employs 6 threads.

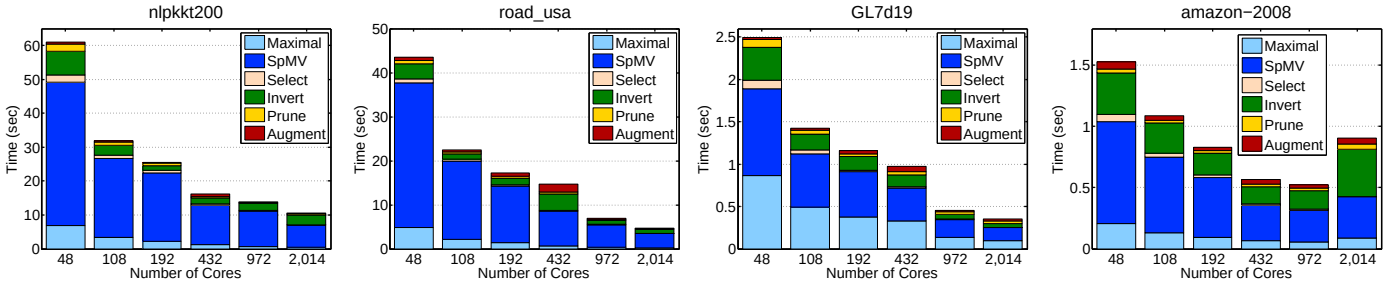

Fig. 5: Runtime breakdown of MCM-DIST for four representative graphs using 12 threads per MPI process on Edison.

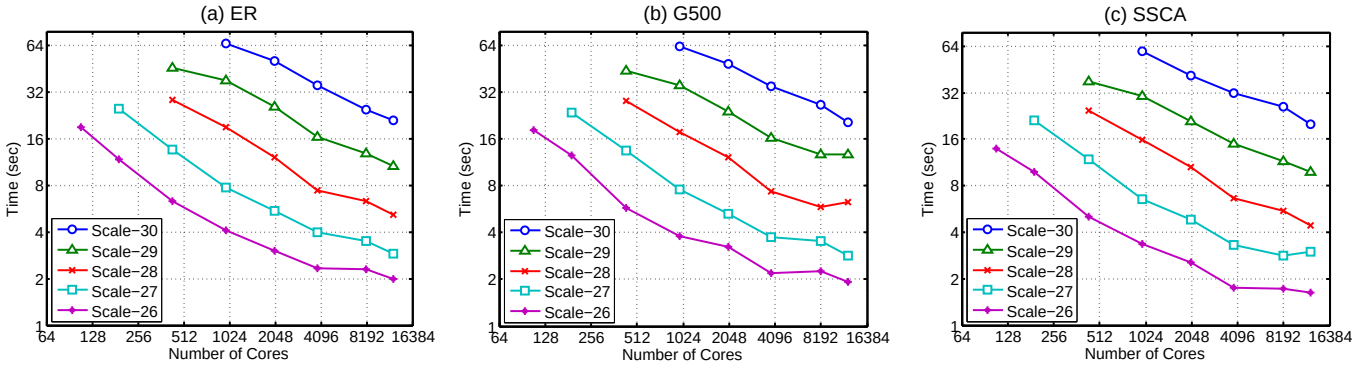

Fig. 6: Strong scaling of MCM-DIST when computing maximum matching on three classes of randomly generated graphs with five different scales on Edison.

where MCM algorithm initialized by Karp-Sipser runs the fastest. Even for these matrices, dynamic mindegree performs closely. Hence, in the rest of our experiments, we use only dynamic mindegree to initialize our MCM algorithm.

B. Scalability of distributed-memory MCM algorithm

We show the strong scaling of our distributed-memory MCM algorithm for 13 real matrices in Fig. 4. We show speedups relative to the runtime of MCM-DIST on a single node (24 cores) of Edison because we assume that the input graphs are already distributed before invoking our matching routine (discussed in Section VI-E). When we go from 24 cores to 972 cores (i.e., $40.5\times$ increase in core count), the average speedup achieved on all 13 real matrices is $9\times$ (min: $5\times$ for `amazon-2008`, max $13\times$ for `delaunay_n24`, stdev: 2.1). In Fig. 4, we show the scalability of smaller matrices (left) and larger matrices (right) separately. Computing matching on large matrices scales better than smaller matrices, as expected. For example, going from 24 cores to 2014 cores, our algorithm attains about $18\times$ and $16\times$ speedups on `delaunay_n24` and `road_usa`, respectively.

To better understand the performance of MCM-DIST and identify its bottlenecks, we show the runtime breakdown for

four representative matrices in Fig. 5. On lower concurrencies, SpMV dominates the runtime as it is the most computation-heavy task of our algorithm. On higher concurrencies, other synchronization-heavy operations, such as INVERT, become significant. For examples, on `road_usa`, SpMV takes about 80% and 60% of total runtime of MCM-DIST on 48 cores and 2014 cores, respectively. On smaller matrices such as `amazon-2008`, INVERT becomes dominant more quickly as the dropping work per process can not mask synchronization costs, which can be seen in the rightmost plot in Fig. 5.

We use large synthetic matrices to demonstrate the scalability of MCM-DIST on higher concurrency. Fig. 6 shows the strong scaling of MCM-DIST when computing MCM on ER, G500 and SSCA matrices on up to 12,288 cores of Edison. Here, we observe that the total runtime of MCM-DIST decrease by a factor of $\sqrt{t}$ when we increase the core count by a factor of t, which is reasonable considering the complexity of MCM algorithms. However, similar to real matrices, MCM-DIST stops scaling on relatively small core counts for smaller matrices. For instance, scale 26 matrices do not scale beyond 4096 cores, whereas scale 30 matrices scale up to 12,288 cores. Recall that `ER-30` and `G500-30` represent graphs with about 2 billion vertices and 32 billion edges. These graphs require more than 600GB of memory (assuming 20 bytes per edge) to store the graph, which exceeds the capacity of a single node of most existing supercomputers. For these massive graphs, distributed-memory algorithms are the only option to compute MCM. Hence, our algorithm laid the foundation to compute MCM on massive graphs on extreme scale.

C. Impact of intra-node multithreading

Thus far, we have always used 12 threads in each MPI process and pinned each process to a socket of Edison. Using 1 thread instead of 12 increases the runtime of MCM-DIST and diminishes its scalability as shown with two examples in Fig. 7. Using the same number of cores, multithreading with 12 threads (Fig. 5) makes MCM-DIST at least twice as fast as non-threaded flat MPI implementation on all concurrencies. Furthermore, non-threaded implementation stops scaling earlier than the multithreaded implementation as can be seen by comparing Fig. 5 and Fig. 7. The reason is that intra-node multithreading reduces the size of MPI communicators, reducing the inter-process communication time and synchronization overheads. The effect of multithreading is more significant on smaller matrices such as `amazon-2008` where the non-threaded implementation does not scale beyond 200 cores.

D. Impact of pruning unnecessary vertices

Pruning vertices from trees that have already discovered augmenting paths (line 22 of Algorithm 2) reduces the runtime of MCM-DIST for most real matrices. Fig. 8 shows the percentage of runtime reduced when vertex pruning is employed on 1024 cores of Edison. For all matrices except two, pruning reduces the runtime from 10% to 65%. Pruning itself is not an expensive step as can be seen in Fig. 5 because it needs

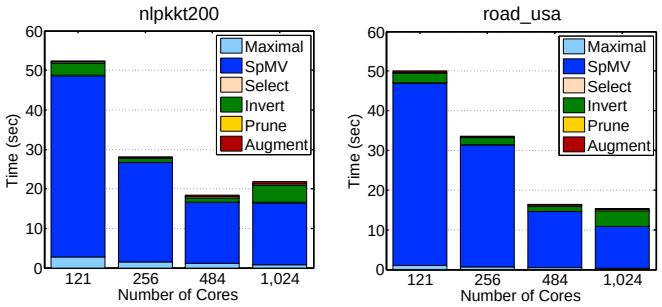

Fig. 7: Breakdown of runtime when MCM is computed using 1 thread per MPI process on Edison.

to communicate information about the roots of trees that have found augmenting paths in the current iteration.

E. Employing shared-memory algorithms on a distributed graph

If a graph fits in a single node and computing a matching is the sole objective, it makes more sense to use a shared-memory algorithm to compute MCM because the state-of-the-art shared-memory implementation [7] is usually faster than our distributed-memory algorithm when the latter is run on a single node. By contrast, if a graph is already distributed, collecting it on a single node requires expensive communication. The communication cost includes gathering the distributed graph on a selected node and scattering the computed MCM from the selected node to all nodes. Fig. 9 shows how the communication time spent in gathering and scattering operations grows quickly with the increase of the number of edges in a graph on 2048 cores of Edison. In this toy example, 2048 MPI processes are run on 2048 cores, and each process stores equal number of edges of a hypothetical distributed graph. The graph is gathered on MPI rank 0, and then row and column matching vectors are scattered from rank 0 to all MPI processes. In addition to the communication cost, considerable preprocessing time is spent in populating local data structures before the shared-memory MCM algorithm is invoked. Therefore, collecting the distributed graph on a single node is expensive and unscalable, especially when the MCM is intended for another distributed application such as a distributed-memory solver. For instance, `nlpkkt200` has about 900M nonzeros. Hence it would take about 20 seconds to collect the graph on a node and scatter the mate vectors according to Fig. 9. This communication overhead alone is twice as long as running MCM in distributed memory via MCM-DIST according to Fig. 4. Since MCM-DIST is targeted primarily to the distributed applications, we do not compare its performance with shared-memory MCM algorithms and only show speedups relative to the single node (24 cores) runtime.

VII. CONCLUSION AND FUTURE WORK

Achieving speedups for the maximum-matching cardinality problem on distributed-memory architectures have been a long standing challenge. This is due to the extremely dynamic nature of the computation, its long critical path, and the lack of scalable parallel primitives. In this work, we showed that

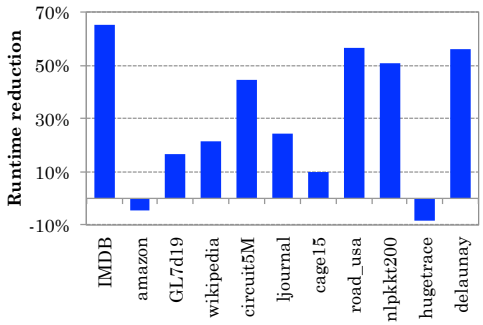

Fig. 8: The change in runtime before and after vertex pruning is employed on 1024 cores of Edison. For most graphs, pruning reduces the total time to compute maximum matching.

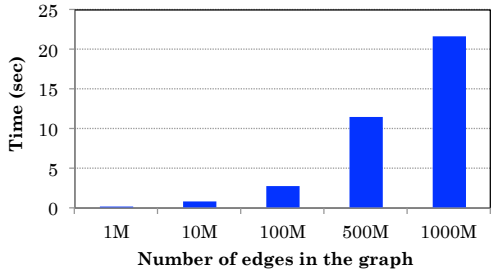

Fig. 9: The communication time spent in gathering and scattering operations with the increase of the number of edges in a graph on 2048 cores of Edison. 20 bytes are used to store an edge, and 8 bytes are used to store the mate of a vertex.

matrix-algebraic primitives, together with moderate use of one-sided communication primitives as needed, enabled our algorithms to achieve speedups to thousands of processors.

While the speedups we achieved are sublinear, they significantly improve over the current published state of the art. Future work includes implementing the tree grafting technique together with the bottom-up BFS in distributed memory and utilizing PGAS languages for better expressing one-sided communication. We also plan to develop faster, communication-avoiding algorithms for the SpMV and INVERT primitives that dominate the performance at large concurrencies.

ACKNOWLEDGMENTS

This work is supported by the Applied Mathematics Program of the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. We used resources of the NERSC supported by the Office of Science of the DOE under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] Y.-Y. Liu, J.-J. Slotine, and A.-L. Barabási, "Controllability of complex networks," *Nature*, vol. 473, no. 7346, pp. 167–173, 2011.

[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] 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.

[4] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with pardiso," *Future Generation Computer Systems*, vol. 20, no. 3, pp. 475–487, 2004.

[5] R. M. Karp, E. Upfal, and A. Wigderson, "Constructing a perfect matching is in random nc," in *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. ACM, 1985, pp. 22–32.

[6] A. V. Goldberg, S. A. Plotkin, D. B. Shmoys, and É. Tardos, "Using interior-point methods for fast parallel algorithms for bipartite matching and related problems," *SIAM J. Comput.*, vol. 21, no. 1, pp. 140–150, 1992.

[7] A. Azad, A. Buluç, and A. Pothen, "A parallel tree grafting algorithm for maximum cardinality matching in bipartite graphs," in *IPDPS*. IEEE, 2015, pp. 1075–1084.

[8] J. Langguth, A. Azad, M. Halappanavar, and F. Manne, "On parallel push–relabel based algorithms for bipartite maximum matching," *Parallel Computing*, vol. 40, no. 7, pp. 289–308, 2014.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] M. Halappanavar, A. Pothen, A. Azad, F. Manne, J. Langguth, and A. Khan, "Codesign lessons learned from implementing graph matching on multithreaded architectures," *Computer*, no. 8, pp. 46–55, 2015.

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

[19] J. Langguth, M. M. A. Patwary, and F. Manne, "Parallel algorithms for bipartite matching problems on distributed memory computers," *Parallel Computing*, vol. 37, no. 12, pp. 820–845, 2011.

[20] M. M. A. Patwary, R. H. Bisseling, and F. Manne, "Parallel greedy graph matching using an edge partitioning approach," in *HLPP'10*. ACM, 2010, pp. 45–54.

[21] A. Azad and A. Buluç, "Distributed-memory algorithms for maximal cardinality matching using matrix algebra," in *IEEE CLUSTER*, 2015.

[22] J. H. Reif, "Depth-first search is inherently sequential," *Information Processing Letters*, vol. 20, no. 5, pp. 229–234, 1985.

[23] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *SC'11*. ACM, 2011, pp. 65:1–65:12.

[24] G. Birkhoff and J. D. Lipson, "Heterogeneous algebras," *Journal of Combinatorial Theory*, vol. 8, no. 1, pp. 115–133, 1970.

[25] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *IJHPCA*, vol. 25, no. 4, 2011.

[26] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and Ü. Çatalyürek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *ACM/IEEE SC'15*, 2005.

[27] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *TPDS*, vol. 8, no. 11, pp. 1143–1156, 1997.

[28] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *IJHPCA*, vol. 19, no. 1, pp. 49–66, 2005.

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

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

[31] "SSCA benchmark," http://www.graphanalysis.org/benchmark/.

[32] "Combinatorial BLAS 1.5," http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/.

[33] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SDM*, 2004.