Distributed-Memory Algorithms for Maximal Cardinality Matching using Matrix Algebra

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

***Abstract*—We design and implement distributed-memory parallel algorithms for computing maximal cardinality matching in a bipartite graph. Relying on matrix algebra building blocks, our algorithms expose a higher degree of parallelism on distributed-memory platforms than existing graph-based algorithms. In contrast to existing parallel algorithms, empirical approximation ratios of the new algorithms are insensitive to concurrency and stay relatively constant with increasing processor counts. On real instances, our algorithms achieve up to $300\times$ speedup on 1024 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 16,384 processors.**

I. Introduction

A matching in a graph is a set of edges without common vertices, and the number of edges in a matching is its cardinality. Computing a matching of maximum cardinality is an important combinatorial problem in scientific computing with applications to permute a matrix to its block triangular form (BTF) via the Dulmage-Mendelsohn decomposition of bipartite graphs [1], [2], and to compute minimum-weight matchings used by sparse direct solvers [3]. Most practical algorithms that compute maximum cardinality matchings initialize themselves by a matching of maximal cardinality because the latter can be computed much faster than the former [4], [5], [6], [7]. Hence, a scalable parallel algorithm for maximal cardinality matching is a prerequisite to parallel maximum matching and other dependent problems. In this paper, we design and implement distributed-memory parallel algorithms for computing matchings of maximal cardinality on an unweighted bipartite graph.

In earlier work, effective parallel algorithms for maximal cardinality matching have been designed and implemented on both shared and distributed memory systems [5], [8], [9], [10], [11]. These algorithms achieve parallelism by concurrently searching for unmatched neighbors from unmatched vertices. Even though the existing algorithms perform well on a small number of processors, they often fail to scale beyond several hundreds of processors. Limited scalability is due to the communication overheads of finding partner vertices to match and updating the matching, which together dominate the runtime. More importantly, the cardinality of matching returned by these graph-based algorithms decreases significantly with the increased concurrency. For example, Azad *et al.* [5] demonstrated that the approximation ratio (the ratio of maximal to maximum cardinality) of a multithreaded

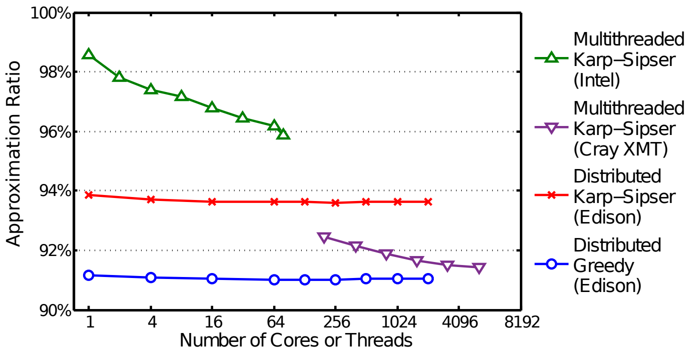

Fig. 1. Matching qualities attained by Karp-Sipser and Greedy algorithms on `delaunay_n24` graph. Multithreaded algorithms are presented in earlier work [5], whereas distributed algorithms, which are matrix based, are presented here for the first time.

Karp-Sipser algorithm decreases significantly on a Cray XMT multiprocessor with more than six thousands threads. Fig. 1 shows that the quality of matchings from the multithreaded Karp-Sipser decreases by more than 2% on 80 threads of an Intel multiprocessor and by another 1% on 6400 threads of a Cray XMT massively multithreaded multiprocessor. This reduction in matching quality is undesirable because the reduced cardinality may significantly increase the runtime of other dependent algorithms (e.g., a maximum matching algorithm) that use a maximal matching as an initializer.

We address the limitations of existing graph-based algorithms by redesigning them using matrix algebra. We present three maximal matching algorithms in matrix algebra. We represent the input graph by a sparse matrix and the vertex sets (including matchings) by vectors, and then decompose the matching algorithms into several steps. The algorithms employ sparse matrix-vector multiplication (SpMV) to search for unmatched rows in the matrix from unmatched columns and vector manipulations to update the current matching.

We show with an extensive set of real and randomly generated problems that our matrix-based algorithms are more amenable to parallelization on distributed-memory platforms. Over diverse set of 11 real input graphs, we achieve an average of $202\times$ speedup (up to $300\times$) on 1024 cores of a Cray XC30 supercomputer (Edison). Even higher speedups are obtained on larger synthetically generated graphs where our algorithms show good scaling on up to 16,384 processors, making them the first algorithms for maximal matching that scale to tens of thousands of processors. Furthermore, unlike previous algorithms, the cardinality of matching obtained by our algorithms is insensitive to concurrency, and remains the same even on several thousands of processors. For example, Fig. 1 demonstrates that the newly developed matrix-based Karp-Sipser algorithm outputs matchings with statistically the same quality on 1 to 2048 cores of Edison.

II. 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. Given a matching M in a 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 an edge (u, v) is matched, we call u and v *mates* of each other. Given a matching M, the *unmatch-degree* of a vertex v is the number of unmatched vertices adjacent to v in the graph. 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'.

The cardinality of the maximum matching is the *matching number* of the graph. The *approximation ratio* of a maximal matching is the ratio of its cardinality to the matching number of the graph. Every maximal matching has an approximation ratio greater than or equal to $1/2$. This ratio is used to measure the quality of maximal matching.

This paper solely focuses on maximal cardinality matchings 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. Consequently, we will occasionally drop the adjectives "bipartite" and "maximal cardinality" when describing our methods.

A. Variants of maximal matching algorithms

The function Maximal-Match-Graph described in Algorithm 1 computes a maximal matching in a bipartite graph $G(R, C, E)$. The algorithm traverses the neighborhood of an unmatched vertex v_c in C, and if an unmatched neighbor v_r in R is found, the edge (v_c, v_r) is included in the matching. The order in which the unmatched vertex v_c is selected in Algorithm 1 defines several variants of maximal matching algorithms. If v_c is selected at random then the algorithm is called the Greedy algorithm. In the Karp-Sipser algorithm [12], vertices with one unmatched neighbor (called degree-1 vertices) are processed before vertices with higher unmatched-degrees. When there is no degree-1 vertex, Karp-Sipser works similar to the Greedy algorithm. Finally, when vertices are selected in the ascending order of unmatch-degrees, Algorithm 1 turns into a Dynamic Mindegree algorithm.

B. Representing a bipartite graph via a sparse matrix

Let G=(R, C, E) be an undirected and unweighted bipartite graph with $|R|$=m and $|C|$=n. Without loss of generality, we assume that $m{\geq}n$. Consider an arbitrary ordering of vertices in each vertex part of G, $R = \{r_1, r_2, ..., r_m\}$ and

Algorithm 1 A maximal matching algorithm based on edge traversal. **Input:** A bipartite graph $G(R, C, E)$. **Output:** A maximal cardinality matching M.

```
1: procedure MAXIMAL-MATCH-GRAPH($G(R, C, E)$)
2:     $M \leftarrow \phi$
3:     $Q \leftarrow C$                                   ▷ Unmatched columns
4:     while $Q \neq \phi$ do
5:         $v_c \leftarrow$ a vertex from $Q$             ▷ Algorithmic variants
6:         if $(v_c, v_r) \in E$ and $v_r$ is unmatched then
7:             $M \leftarrow M \cup (v_c, v_r)$
8:         $Q \leftarrow Q \setminus \{v_c\}$
       return $M$
```

$C = \{c_1, c_2, ..., c_n\}$. Then, we represent G by an $m \times n$ binary sparse matrix $\mathbf{A}$ with $|E|$ nonzero entries (i.e., $nnz(\mathbf{A})$=$|E|$) such that $\mathbf{A}(i, j)$=1 when there is an edge between the ith row vertex r_i and jth column vertex c_j. By a reverse construction, we can also create a bipartite graph from a binary matrix. Fig. 2 shows an example of representing a bipartite graph with a sparse matrix. Note that $\mathbf{A}$ can be unsymmetric, rectangular (when $m{\neq}n$), and might have nonzero entries in the diagonal (when there are edges of the form (r_i, c_i)). Hence, $\mathbf{A}$ is not the adjacency matrix of the bipartite graph G since the actual adjacency matrix is an $(m+n)\times(m+n)$ square matrix with zero diagonal.

C. Representing matching and 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 nonzero entries and $len(x)$ denotes the number of both zero and nonzero entries in x. For a dense vector x, $nnz(x)$= $len(x)$. Given a sparse/dense vector x and an index vector I with $\max(I){\leq}$ $len(x)$, $x[I]$ selects the nonzero entries from indices specified by I. We use subscripts r and c to denote vectors of row and column vertices, respectively.

In our matching algorithms, we store the mates of row and columns vertices in two dense vectors $mate_r$ and $mate_c$. If the ith row vertex r_i is matched to the jth column vertex c_j, then $mate_r[i]$=j and $mate_c[j]$=i. $mate_r[i]$ is set to -1 when r_i is unmatched. Consider a graph with five column vertices and f_c =$\{c_1, c_2, c_5\}$ to be a subset of column vertices in the graph. Then, we store f_c in a sparse vector of length five with nonzeros in 1st, 2nd and 5th locations: f_c =$[\times, \times, 0, 0, \times]$. Here, $len(f_c)$=5, $nnz(f_c)$=3. Therefore, the indices of the nonzero entries of a sparse vector represent the actual vertices, whereas the values stored in nonzero entries store pointers to other vertices such as parents or mates. We show several examples of sparse and dense vectors in Fig. 3 in the context of a matching algorithm.

D. Operations on vectors and matrices

Table I defines several operations on vectors and matrices, which will be used in matching algorithms. The function Ind(x) returns the local indices of nonzero entries of a sparse vector x. Since we need to copy $nnz(x)$ indices, the complexity of this operation is $O(nnz(x))$. Given a sparse

Function	Arguments	Returns	Example	Serial Complexity	Need Comm?
IND	x: a sparse vector	local indices of nonzero entries of x	sparse vector: $x = [3, 0, 2, 2, 0]$ $\text{IND}(x) = [1, 3, 4]$	$O(nnz(x))$	No

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**
		$z[i] \leftarrow x[i]$
			sparse vector: $x = [3, 0, 2, 2, 0]$
			dense vector: $y = [1, -1, -1, 2, -1]$
			$\text{SELECT}(x, y = -1) = [0, 0, 2, 0, 0]$
				$O(nnz(x))$	No

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** $z[x[i]] \leftarrow i$
			sparse vector: $x = [3, 0, 2, 2, 0]$
			$\text{INVERT}(x) = [0, 4, 1, 0, 0]$
				$O(nnz(x))$	Yes

SPMV	$\mathbf{A}$: a sparse matrix x: a sparse vector SR: a semiring	returns $\mathbf{A} \cdot x$	see Fig. 2	$\sum_{k \in \text{IND}(x)} nnz(\mathbf{A}(:, k))$	Yes

TABLE I
BASIC FUNCTIONS NEEDED FOR THE CARDINALITY MATCHING ALGORITHM.

vector x, a dense vector y and a logical expression $expr$, the function $\text{SELECT}(x, y, expr)$ selects indices I of y where $expr(y)$ is true and returns $x[I]$. As shown in the pseudocode in Table I, SELECT only iterates on the sparse vector, hence the complexity $O(nnz(x))$. Given a sparse vector x, the INVERT function returns the inverted index by swaping the indices and values of nonzero entries in x and stores the results in a new sparse vector z. If x has repeated nonzero values, only one of them is used as index in z (we keep the first index).

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 breadth-first search (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 less strict than the definition used in mathematics.

Consider a bipartite graph $G(R, C, E)$, its matrix representation $\mathbf{A}$, and a set of column vertices f_c. Then, Fig. 2 shows the execution of the SpMV $\mathbf{A} \cdot f_c$ over the *(select2nd, min)* semiring. SpMV returns the set of row vertices explored by f_c. The *(select2nd, min)* semiring can be replaced by *(select2nd, max)* or other equivalent semirings.

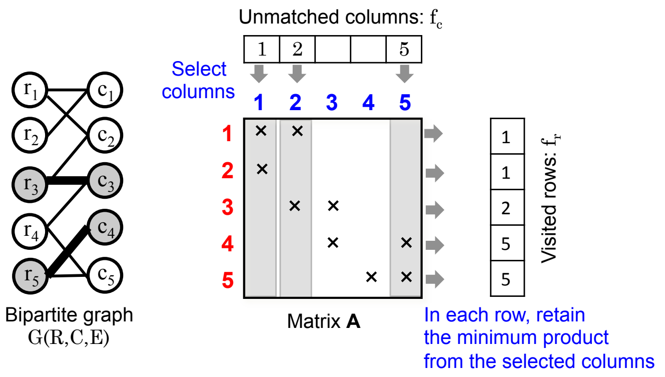

Fig. 2. Illustration of traversing a bipartite graph $G(R, C, E)$ via SpMV. The bipartite graph with five row and five column vertices is shown in the left. Matched and unmatched vertices are shown in filled and empty circles, respectively. Thin lines represent unmatched edges and thick lines represent matched edges. The binary matrix $\mathbf{A}$ represents the bipartite graph where an "x" denotes an edge in G. f_c represents the set of unmatched column vertices. The sparse matrix-vector multiplication $\mathbf{A} \cdot f_c$ over the *(select2nd, min)* semiring first selects columns that have nonzeros in f_c (shown in gray) and then in each row, retains the minimum product 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 the column vertex that explored the ith row vertex.

E. Related Work

There has been a large body of research on the theory of parallel matching algorithms, e.g., Karpinski and Rytter [13]. However, most of these algorithms are based on parallel random access machine (PRAM) model, and they are often impractical on modern parallel platforms. Considerable interest in parallel algorithms has been observed recently with work on approximation as well as exact algorithms on shared and distributed memory platforms. Patwary *et al.* [11] have 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. Langguth *et al.* describe their work on parallelizing the Push-Relabel algorithm for bipartite maximum matching on both shared and distributed-memory platforms [7], [14]. However, their distributed-memory push-relabel algorithm did not scale well beyond 64 processors [14].

Parallel algorithms for weighted matching have also been studied [15]. Recently, Sathe *et al.* have reported $4\times$ to $64\times$ speedups on 1024 processors of a Cray XE6 for a parallel auction algorithm [16]. Half-approximation algorithms for weighted matching have been implemented on both shared and distributed memory computers with good speedups [17], [18], [19], [20].

III. MATRIX ALGEBRA BASED FORMULATION OF MATCHING ALGORITHMS

A. The greedy matching algorithm

The function MAXIMAL-MATCH-MTX in Algorithm 2 describes the greedy matching algorithm using matrix algebra building blocks. As inputs, the algorithm takes a matrix

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

```
procedure MAXIMAL-MATCH-MTX($\mathbf{A}$, $mate_c$, $mate_r$)
    $f_r \leftarrow$ A sparse vector of size $m$ with $f_r[i] = 1$                ▷ Unmatched row vertices
    $f_c \leftarrow$ A sparse vector of size $n$ with $f_c[i] = i$                ▷ Unmatched column vertices
    $\mathbf{A}^\mathsf{T} \leftarrow$ TRANSPOSE($\mathbf{A}$)                    ▷ Transpose matrix
    $d_c \leftarrow$ SPMV($\mathbf{A}^\mathsf{T}$, $f_r$, SR=(select2nd,+))       ▷ Degrees of column vertices to unmatched row vertices
    $f_c \leftarrow$ SELECT($f_c, d_c > 0$)                                       ▷ Remove isolated vertices
    repeat
        ▷ Step 1: Discover unmatched rows from unmatched columns (one step of BFS)
        $f_r \leftarrow$ SPMV($\mathbf{A}$, $f_c$, SR=(select2nd,min))            ▷ Explore row vertices from unmatched column vertices
        $f_r \leftarrow$ SELECT($f_r, mate_r = -1$)                               ▷ Unmatched visited row vertices

        ▷ Step 2: Update matching
        $t_c \leftarrow$ INVERT($f_r$)                 ▷ For each column vertex, select one of its children if available
        $J \leftarrow$ IND($t_c$)
        $mate_c[J] \leftarrow t_c$                     ▷ Match column vertices with their selected children
        $t_r \leftarrow$ INVERT($t_c$)                 ▷ Selected row vertices pointing to their unique parents
        $I \leftarrow$ IND($t_r$)
        $mate_r[I] \leftarrow t_r$                     ▷ Match an unmatched row vertex with its unique parent

        ▷ Step 3: Update unmatched column vertices $f_c$
        $md_c \leftarrow$ SPMV($\mathbf{A}^\mathsf{T}$, $t_r$, SR=(select2nd,+))  ▷ Degrees of all column vertices to the newly matched row vertices
        $J \leftarrow$ IND($md_c$)                     ▷ Indices of column vertices adjacent to the newly matched row vertices
        $d_c[J] \leftarrow d_c[J] - md_c$              ▷ Update unmatch-degrees of column vertices
        $f_c \leftarrow$ SELECT($f_c, mate_c = -1$)    ▷ Keep unmatched columns
        $f_c \leftarrow$ SELECT($f_c, d_c > 0$)        ▷ Keep unmatched columns with positive degrees
    until no vertex is matched in the last iteration
```

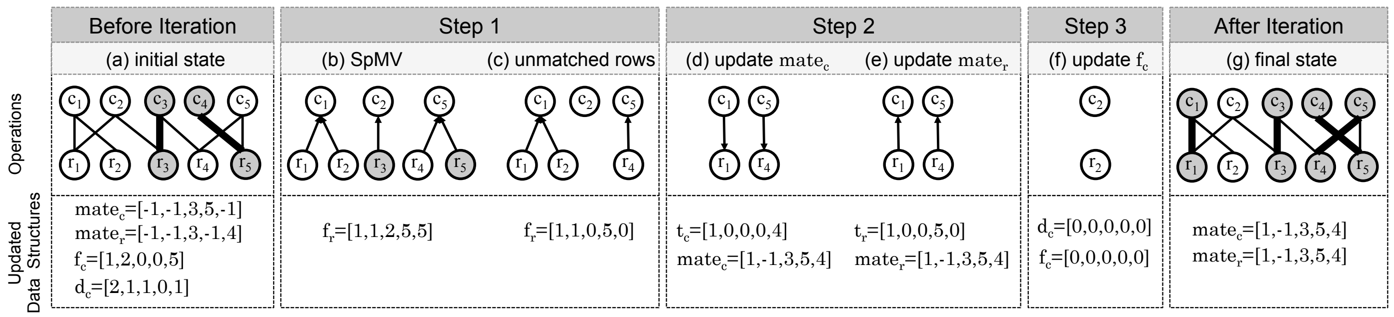

Fig. 3. A working example of one iteration of a maximal matching algorithm described in Algorithm 2. Matched and unmatched vertices are shown in filled and empty circles, respectively. Thin lines represent unmatched edges and thick lines represent matched edges. The vectors f, d, and $mate$ represent the current unmatched vertices, unmatch-degree of vertices, and mates of the matched vertices. Subscripts r and c denote row and column vertices, respectively. The temporary vectors t_r and t_c store unmatched row and column vertices that are matched in this iteration (see Step 2 of Algorithm 2). (a) An initial matching and associated data structures before an iteration, (b) exploring the neighbors of unmatched column vertices f_c where arrows direct from children to parents, (c) keep only the unmatched row vertices as children, (d) match column vertices to their unique children, (e) match row vertices to their unique parents, (f) update f_c by removing newly matched columns and columns with no unmatched neighbors, and (g) a maximal matching is obtained. In Subfigs. (d) and (e), arrows show the direction of matching.

$\mathbf{A}$ representing a bipartite graph and two dense vectors $mate_r$, and $mate_c$ storing the mates of row and column vertices. MAXIMAL-MATCH-MTX returns a maximal cardinality matching by updating $mate_r$ and $mate_c$. At first, we create two sparse vectors f_r and f_c storing the unmatched row and column vertices. The values of f_c are set to their indices to facilitate BFS traversals. We keep both $\mathbf{A}$ and its transpose $\mathbf{A}^\mathsf{T}$ so that we can traverse the graph from both row and column vertices. The dense vector d_c of size n stores the unmatch-degree of column vertices. Initially, we compute d_c by multiplying $\mathbf{A}^\mathsf{T}$ by f_r over the *(select2nd, +)* semiring.

One pass over the **repeat-until** block in Algorithm 2 defines an *iteration* of the algorithm. Fig. 3 demonstrates the execution of one iteration of the MAXIMAL-MATCH-MTX function. In this example, the bipartite graph has five row vertices and five column vertices. Two of the vertices are matched before the current iteration (Subfig. 3(a)). We divide each iteration of MAXIMAL-MATCH-MTX into three steps described below.

Step 1: Explore graph from unmatched column vertices. Let f_c be the set of unmatched column vertices at the beginning of an iteration. In this step, we discover a set of row vertices f_r reachable from f_c by using SpMV over the *(select2nd, min)* semiring, as illustrated in Fig. 2. If we consider f_c to be the current frontier, the SpMV is essentially conducting one iteration of BFS-based graph traversal. In this context, vertices in f_r have unique parents in f_c and construct a forest of unit height as shown in Fig. 3(b). The parents of row vertices are stored as nonzero values in f_r. Since we are only interested in unmatched vertices, the matched rows are removed from f_r (Subfig. 3(c)), which concludes Step 1 of the algorithm.

Step 2: Update matching. We update mates of column vertices by calling INVERT(f_r) that selects a unique child for each vertex in f_c and update $mate_c$ accordingly. Note that, in Step 1, an unmatched column vertex in f_c might have acquired more than one child, e.g., r_1 and r_2 are children of c_1 in Fig. 3(b). In this case, INVERT matches a vertex in f_c to exactly one of its children. To update $mate_r$, we execute INVERT on the newly matched column vertices. This process is described between lines 13–18 of Algorithm 2 and illustrated in Subfigs. 3(d) and 3(e).

Step 3: Update unmatched column vertices. After updating mates, we remove the newly matched columns from f_c. For example, after matching c_1 and c_5 in Subfig. 3(d), we have a single unmatched column vertex c_2. We could start the next iteration with $f_c = \{c_2\}$. Since c_2 has no unmatched neighbor, we can remove it from f_c to reduce work in future iterations. For this purpose, we update the unmatch-degree d_c of column vertices by first computing the degree md_c of column vertices to the newly matched row vertices and then subtracting md_c from d_c (lines 21–23 of Algorithm 2). In our example in Fig. 3, we have no more unmatched vertices, hence the algorithm returns with a maximal matching shown in Subfrig. 3(g).

Algorithm 3 Modified Step 1 of Algorithm 2 needed for the Karp-Sipser algorithm.

```
1: $f_c^1 \leftarrow$ SELECT($f_c, d_c = 1$)                ▷ degree-1 column vertices
2: if $nnz(f_c^1) \neq 0$ then                         ▷ Process degree-1 vertices
3:     $f_r \leftarrow$ SPMV($\mathbf{A}, f_c^1$, SR=(select2nd,min))
4: else                                                ▷ Process other vertices
5:     $f_r \leftarrow$ SPMV($\mathbf{A}, f_c$, SR=(select2nd,min))
```

B. The Karp-Sipser algorithm

We can convert Algorithm 2 into the Karp-Sipser algorithm by replacing Step 1 with Algorithm 3. Here, f_c^1 is the set of unmatched column vertices that have unmatch-degree equal to 1. f_c^1 is easy to compute because we update unmatch-degree of column vertices in Step 3 of every iteration in Algorithm 2. If f_c^1 is not empty, we explore the neighborhood of these degree-1 vertices (lines 3–4 of Algorithm 3) and try to match them, otherwise we proceed with the greedy algorithm.

C. The Dynamic Mindegree algorithm

We can convert Algorithm 2 into the Dynamic Mindegree algorithm by using the *(select2nd, mindegree)* semiring in line 9. Here, the vector entries are {parent, degree} pairs for the SPMV. Hence, the *(select2nd, mindegree)* semiring operates on a set of binary numbers and a set of pairs of integers. As before, *select2nd* returns the second value it is passed, i.e., the {parent, degree} pair. The *mindegree* operation takes two {parent, degree} pairs and returns the pair with minimum degree. The *mindegree* operation can be implemented as a function or a lambda expression in C++. The rest of Algorithm 2 remains unchanged for Dynamic Mindegree.

D. Serial complexity

The computational complexity of every iteration of Algorithm 2 depends on the functions described in Table I. Since SPMV dominates other operations in terms of serial complexity, it determines the serial runtime of the matrix-based algorithms. The cost of a single SpMV with a sparse vector x depends on the number of nonzeros of x and their locations. The number of multiplications is $\sum_{k \in \text{IND}(x)} nnz(\mathbf{A}(:,k))$, as listed in Table I as well.

First, note that two SPMV calls (lines 9 and 21) in Algorithm 2 use two different vectors. The first SPMV with the *(select2nd, min)* semiring uses unmatched columns in the vector x. By contrast, the second SPMV with the *(select2nd, +)* semiring uses the newly matched row vertices as the vector. Since the number of newly matched vertices is smaller than the number of unmatched vertices, the cost of the first SPMV is often higher than the other. Hence, we only discuss the cost of the first SPMV. (However, two SpMVs traverse the graph from opposite directions, hence their costs also depend on the nonzero structure.)

In the first iteration, the SPMV in line 9 of Algorithm 2 uses a dense vector because all column vertices are unmatched at the beginning. Hence, the cost of the first iteration of Greedy, Dynamic Mindegree, and Karp-Sipser (when there is no degree-1 vertices in the input graph) algorithms is $O(nnz(\mathbf{A}))$. However, in the presence of degree-1 vertices in the input graph, the cost of the first iteration of Karp-Sipser depends on the number of degree-1 vertices. The cost of subsequent iterations depend significantly on the algorithm and input graphs (e.g., see Fig. 6). However, all of our algorithms spend most of their time in the first iteration. For example, Fig. 6 shows that every algorithm spends at least 18% of their total runtime in the first iteration on `GL7d19`.

IV. DISTRIBUTED MEMORY PARALLEL ALGORITHM

A. Data distribution and storage

We use the CombBLAS framework [21] 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.

A 2D SpMV algorithm for the case of sparse input and output vectors has previously been used in the specialized context of distributed memory BFS [22], which we leverage here. A 2D SpMV algorithm for the case of dense vectors is also provided in CombBLAS. As discussed before, serial SpMV performs $\sum_{k \in \textsc{Ind}(x)} nnz(\mathbf{A}(:,k))$ multiplications. The total work of the parallel algorithm is the same (i.e. our parallel SpMV is work efficient), but the load balance depends on the exact distribution of nonzeros in $\mathbf{A}$ and x. Hence, we will be analyzing the parallel running time with the assumption that nonzeros of $\mathbf{A}$ and x are i.i.d. distributed. This provides a lower bound on the running time and the actual observed performance can be worse in the presence of load imbalance.

The allgather phase of SpMV has cost

$$T_{ag} = \alpha(p_r - 1) + \beta \frac{p_r - 1}{p_r}\, nnz(x_i)$$

using the ring algorithm, which is the default algorithm for large ($\geq$ 512KB) messages on many MPI implementations such as MPICH [23]. Recall that x_i is the n/p_c length piece collectively owned by the ith processor column, which needs to be gathered at each processor on that processor column. The all-to-all phase, assuming the pairwise-exchange algorithm that is typical for long messages in many MPI implementations, has the cost

$$T_{a2a} = \alpha(p_c - 1) + \beta \sum_{k \in \textsc{Ind}(x)} nnz(\mathbf{A}_{ij}(:,k))$$

By contrast, INVERT requires a permutation of vector entries among all processors, and has per-processor cost of

$$\begin{aligned} T_{\textsc{Invert}} &= nnz(x_{ij}) + \alpha(p-1) + \beta\, nnz(x_{ij}) \\ &= \frac{n}{p} + \alpha(p-1) + \beta \frac{n}{p} \end{aligned}$$

using personalized all-to-all. INVERT is also work-efficient but communication intensive. We perform certain approximations to make these analyses comparable to each other. First we use asymptotics, $p - 1 \approx p$, second we assume a square processor grid $p_r{=}p_c{=}p$, and finally we assume i.i.d. distributed nonzeros in $\mathbf{A}$ and x.

If x is f percent full, and $\mathbf{A}$ has on average d nonzeros per column, then the arithmetic cost of SpMV per processor is

$$nnz(x_i)\, nnz(\mathbf{A}_{ij}(:,k)) = \frac{fn}{\sqrt{p}} \frac{d}{\sqrt{p}} = \frac{fnd}{p} = f \frac{nnz(\mathbf{A})}{p}$$

for some column k. In the worst case, i.e., in the absence of nonzero collusions, the amount of words moved due to all-to-all is the same as the arithmetic cost. Allgather phase moves $fn/\sqrt{p}$ words, resulting in a total cost for SpMV as:

$$T_{\textsc{SpMV}} = \frac{nd}{p} + 2\alpha\sqrt{p} + f\beta\Big(\frac{nd}{p} + \frac{n}{\sqrt{p}}\Big)$$

From that, we see that INVERT has a factor of $\sqrt{p}$ higher latency cost, which hurts performance on large concurrencies and smaller matrices where the latency term dominates. In other words, INVERT is the potential bottleneck in the strong scaling regime. On the other hand, in the weak scaling regime, SPMV is projected to be the bottleneck when $\sqrt{p} > d$ due to worse scaling of the allgather phase. Any future reductions in communication costs of SPMV would make our algorithm even more scalable.

V. Experimental Setup

A. Platform

We evaluate the performance of parallel matching algorithms 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 Intel C++ Compiler (icc 15.0.1) to compile the code, and Cray's MPI implementation on Edison is based on MPICH2. We use one thread per MPI process in all of our experiments (flat MPI).

B. Input Graphs

Table II describes a representative set of graphs from the University of Florida sparse matrix collection [24] and several randomly generated instances. We generated random graphs with skewed degree distribution by using the recursive graph generator called RMAT. We set the RMAT parameters a, b, c, and d to $0.57, 0.19, 0.19, 0.05$ respectively and the average degree to 16 unless otherwise stated. These parameters are identical to the ones used for generating synthetic instances in the Graph500 BFS benchmark [25]. Like Graph500, to compactly describe the size of a graph, we use the scale variable to indicate that the graph has 2^{scale} vertices.

We work with the matrices directly without constructing any graphs. We interpret the matrices in the following way. Let A be an $m \times n$ matrix with nnz nonzero entries. We consider the matrix representing an undirected bipartite graph $G(R \cup C, E)$ where every row (column) of A is represented by a vertex in R (C), and each nonzero entry $A[i,j]$ of A is represented by two edges (r_i, c_j) and (c_j, r_i) connecting the ith row and jth column vertices. We maintain both A and its transpose so that we can search graphs from both directions.

VI. Results

A. Quality of matching

We measure the quality of a maximal matching by its approximation ratio. The matching number needed to compute the approximation ratio is computed by the serial maximum matching software by Duff *et al.* [4]. When the graph is larger than the capacity of the local memory such as the randomly

Class	Graph	#Rows (m) ($\times 10^6$)	#Columns (n) ($\times 10^6$)	nnz ($\times 10^6$)	Maximum Card. (%)	Description
Scientific	hugetrace	16.00	16.00	96.00	100.00	Frames from 2D Dynamic Simulations
	delaunay_n24	16.77	16.77	201.33	100.00	Delaunay triangulations of random points
Scale-free	amazon0312	0.40	0.40	6.40	99.56	Amazon product co-purchasing network
	coPapersDBLP	0.54	0.54	60.98	99.95	Citation networks in DBLP
Networks	wikipedia	3.56	3.56	90.06	58.70	Wikipedia page links
	road_usa	23.94	23.94	115.42	94.90	USA street networks
Rectangular	LargeRegFile	2.11	0.80	4.94	100.00	circuit simulation problem
	Rucci1	1.98	0.11	7.79	100.00	least squares problem
	tp-6	0.14	1.01	11.53	100.00	linear programming problem
	GL7d19	1.91	1.96	37.32	100.00	combinatorial problem
	spal_004	0.01	0.32	46.17	100.00	linear programming problem
Random	ER-26	67.11	67.11	2147.48	100.00	Erdős-Rényi random graphs
	RMAT-26	32.80	32.80	2103.85	51.02	RMAT random graphs (param: .57,.19,.19,.05)

TABLE II

TEST PROBLEMS FOR EVALUATING THE MATCHING ALGORITHMS. THE PROBLEMS ARE GROUPED INTO FIVE CLASSES. IF NEEDED, THE MATRIX IS TRANSPOSED SO THAT $m \geq n$. MAXIMUM MATCHING CARDINALITIES ARE SHOWN AS A FRACTION OF THE MINIMUM OF m AND n.

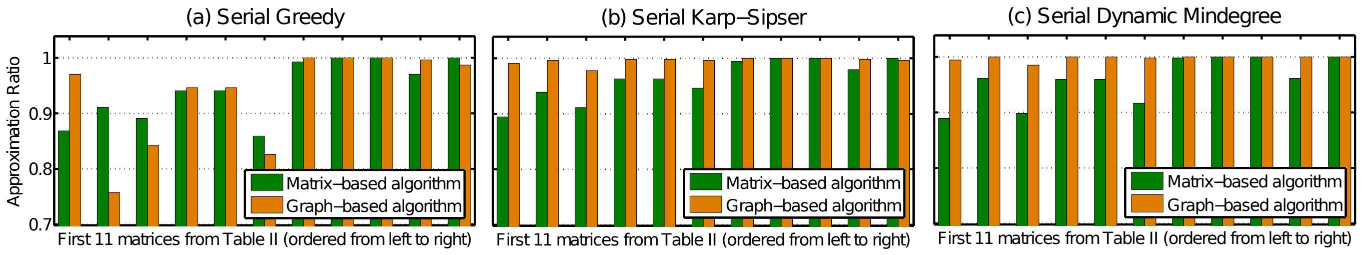

Fig. 4. Approximation ratios attained by serial matrix- and graph-based algorithms.

generated ER and RMAT graphs, we used a preliminary version of our distributed-memory maximum matching algorithm.

At first, we compare the quality of serial matrix-based algorithms with the sequential graph-based algorithms developed by Duff *et al.* [4]. Fig. 4 shows the comparisons. For Greedy algorithm, there is no clear winner: graph-based algorithm performs better on 5 problems, whereas matrix-based algorithm performs better on 4 problems. However, for Karp-Sipser and Dynamic Mindegree, graph-based algorithms consistently outperform the matrix-based algorithms. This behavior is not unexpected because the original Karp-Sipser and Dynamic Mindegree algorithms process vertices based on their unmatch-degrees that are updated after matching every vertex. Since our matrix-based algorithms process all unmatched vertices simultaneously in order to increase concurrency, the matching quality might reduce slightly. However, the matching quality obtained from a parallel graph-based algorithm decreases rapidly as we increase the concurrency [5]. Using Fig. 1, we already discussed in the introduction that a graph-based Karp-Sipser algorithm could lose more than 3% of the matching quality on multithreaded multiprocessors with several thousands of threads. By contrast, on the same graph, the quality of matrix-based algorithms remains statistically the same on thousands of processors as shown in Fig. 1.

We quantify the variability of matching quality by the coefficient of variation (CV) of approximation ratios on different number of processors. Let $r_1, r_2, ..., r_p$ be the approximation ratios achieved by an algorithm on $1, 2, ..., p$ processors. Then, CV is computed as the ratio of standard deviation of r_i to their

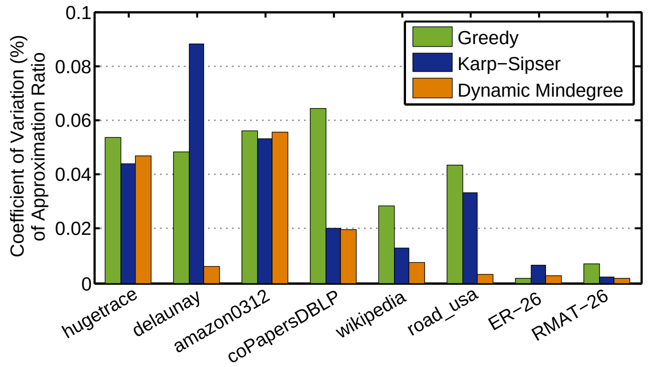

Fig. 5. The stability of matching qualities of matrix-based matching algorithms. The algorithms are run on 1 to 1024 processors for real graphs, and on 128 to 8,100 processors for random graphs.

mean and expressed as a percentage. The higher the CV of an algorithm the more variable its matching quality is. Fig. 5 shows the CVs of matching quality of three maximal matching algorithms on eight input graphs from Table II. Rectangular matrices are not shown because every algorithm achieves 100% approximation ratio on these graphs, hence CV becomes zero. We observe that all of our algorithms are quite stable on these eight graphs as we vary the number of processors. The highest variation is observed on `delaunay_n24` by Karp-Sipser, which is still smaller then 0.1%. The small variations in matching quality originate from the random permutations of input matrices performed for load balance. Hence, this less than 0.1% CV, albeit small, can in practice be completely eliminated if the random permutations are fixed for all concurrencies.

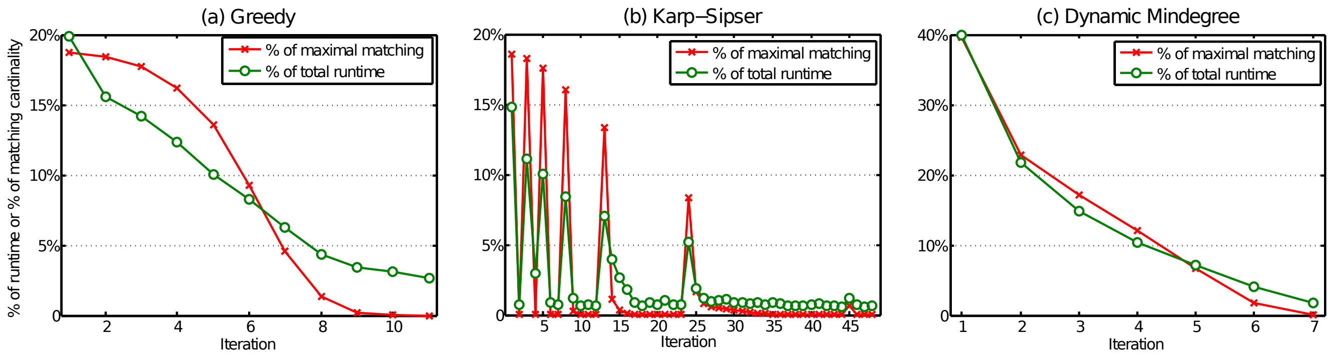

Fig. 6. Fraction of time spent and fraction of vertices matched (percentage of the final maximal matching cardinality) in each iteration of the matching algorithms on GL7d19 graph on 1024 processors.

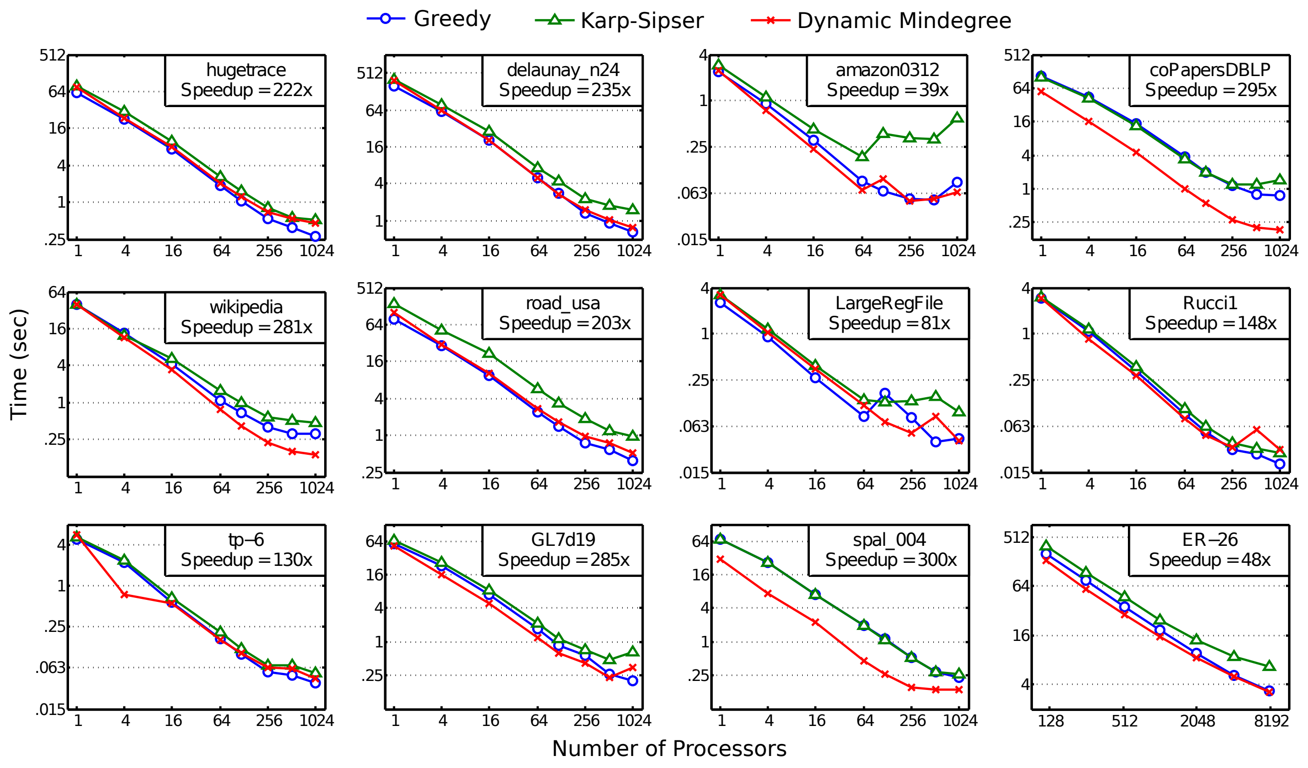

Fig. 7. Strong scaling of maximal matching algorithms on 12 graphs. On each graph (except ER-26), we report the maximum speedup attained by the fastest algorithm on 1024 cores of Edison. On ER-26, the speedup is reported on 8192 cores relative to the runtime on 128 cores.

B. *Progression of algorithms*

The number of iterations needed by three matching algorithms differs significantly depending on input graphs. Generally, Karp-Sipser needs the largest number of iterations and Dynamic Mindegree needs the smallest. For example, Fig. 6 shows the fraction of time spent and fraction of vertices matched (relative to the cardinality of the final maximal matching) in each iteration of our algorithms on GL7d19 graph. On this graph, Greedy algorithm takes 11, Karp-Sipser takes 48, and Dynamic Mindegree takes 7 iterations before completion. Both Greedy and Dynamic Mindegree match fewer new vertices and take shorter time as the algorithms progress from one iteration to the next. Hence, first few iterations of these two algorithms perform most of the work. By contrast, the iterations of Karp-Sipser can be grouped into few *iteration-clusters*, where an iteration-cluster begins with a spike and ends before the next spike in Fig. 6(b). For example, iterations 8-12 create the 4th iteration-cluster in Fig. 6(b). In the first iteration of an iteration-cluster (except the first iteration-cluster), we have no degree-1 vertices; hence the algorithm performs one step of the Greedy algorithm (line 6 of Algorithm 3). In subsequent iterations within the iteration-cluster, the algorithm matches degree-1 vertices as they are discovered. The Dynamic Mindegree algorithm often packs an iteration-cluster of Karp-Sipser into a single iteration. Therefore, the number of iteration-clusters in Karp-Sipser often equals to the number of iterations required by Dynamic Mindegree, e.g., they are the equal on GL7d19 as shown in Fig. 6. Since Dynamic Mindegree needs fewer synchronization

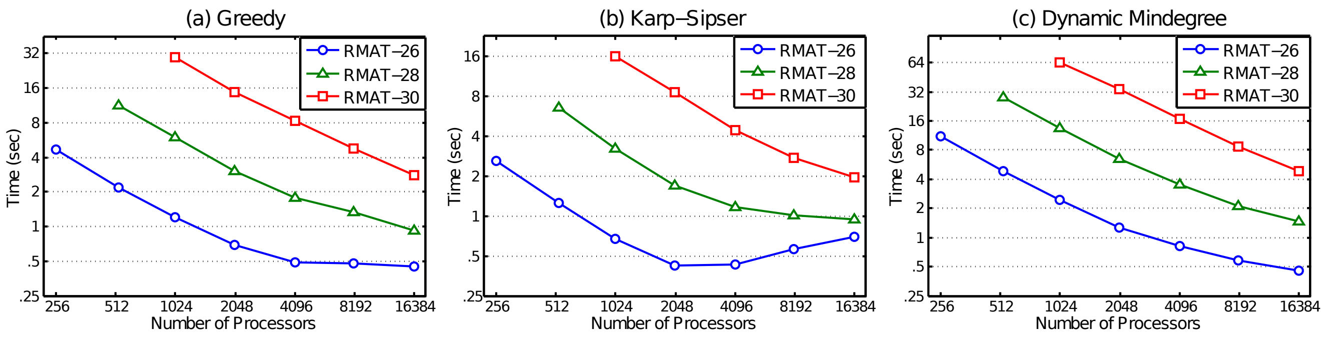

Fig. 8. Strong scaling of maximal matching algorithms on RMAT graphs.

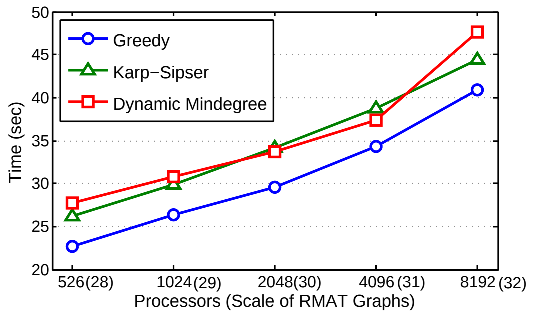

Fig. 9. Weak scaling of maximal matching algorithms on RMAT graphs. The number of processors doubles for each increase in scale of RMAT graphs. When we increase the number of processors by $64\times$, the algorithms slows down by a factor less than $2\times$.

points, it usually runs faster than Karp-Sipser at the expense of slightly smaller matching cardinality.

C. Scalability

We show the strong scaling of three maximal matching algorithms in Fig. 7. On seven out of eleven real input graphs, matching algorithms achieve more than $200\times$ speedups on 1024 cores. Two other graphs (`Rucci1` and `tp-6`) achieve more than $130\times$ speedups. Only two graphs show moderate scaling: `amazon0312` 39×, and `LargeRegFile` 81×. In fact, our algorithms stop scaling after 64 processors on `amazon0312` and `LargeRegFile` because these graphs have the fewest number of edges (6.4 and 4.94 millions respectively) in our problem set. For these small graphs, processors do not have enough work when p is greater than 64, which limits the performance.

The scalability of matrix-based algorithm on higher number of processors can be realized on larger graphs. For example, in Fig. 7, matching on `ER-26` runs $48\times$ faster when we increase p by $64\times$. Fig. 8 shows the strong scaling of RMAT random graphs on up to 16,384 cores. Recall that `RMAT-30` denotes a graph with about 1 billion vertices and 34 billion edges. We observe that larger graphs scale better than smaller graph on very large number of processors, as expected. For example, on `RMAT-30`, every algorithm achieves more than $8\times$ speedup when we go from 1,024 to 16,384 processors. By contrast, on `RMAT-26`, greedy algorithm achieves only $3\times$ speedup and Karp-Sipser stops scaling for the same range of processors. Therefore, our algorithms have the ability to scale on very large number of processors as long as we have enough work to utilize the available computing resources. Among three algorithms presented in this paper, Dynamic Mindegree scales the best and Karp-Sipser scales the worst on large number of processors as can be seen in Fig. 8. When we go from 1,024 to 16,384 cores (i.e., $16\times$ increase), Dynamic Mindegree achieves about $15\times, 10\times$, and $6\times$ speedups on `RMAT-30`, `RMAT-28`, and `RMAT-26`, whereas Karp-Sipser achieves about $8\times, 3\times$, and $0\times$ speedups on `RMAT-30`, `RMAT-28`, and `RMAT-26`. The greedy algorithm sits in between Dynamic Mindegree and Karp-Sipser. According to our discussed in the previous subsection, Dynamic Mindegree scales better than Karp-Sipser because the former requires fewer iteration than the latter (i.e., the former performs more work than the latter per iteration).

We now turn to the weak scaling of the matching algorithms. Fig. 9 shows the weak scaling on RMAT random graphs where we run our algorithm from scale 28 to 32. Note that RMAT graph at scale=32 has $2^{32} = 4$ billions vertices and 68 billion edges. For weak scaling, the number of processors doubles for each increase in scale of RMAT graphs. We observe that the number of processors increased by $64\times$ slows down our algorithms by a factor less than $2\times$. Hence, the matrix-based algorithms can utilize even larger number of processors if large enough graphs become available.

D. Breakdown of runtime

Fig. 10 shows the breakdown of where the runtime is spent by the Dynamic Mindegree algorithm on 64 and 1024 cores of Edison. On 64 cores, Dynamic Mindegree spends about $40\% - 60\%$ time on graph traversal where the algorithm searches for row vertices from unmatched column vertices (Step 1 of Algorithm 2) and about $20\%–30\%$ time on updating degrees and unmatched vertices (Step 3 of Algorithm 2). Updating matching (Step 2 of Algorithm 2) is a communication intensive step; therefore it takes less than 20% time on 64 cores. However, on 1024 cores, updating matching becomes the most expensive step for most smaller graphs because of the increased communication needed by this step. However, larger graphs such as `RMAT-26`, `ER-26`, and `road_usa` can still hide the communication time of updating matching and spend less than 10% of total runtime on this step.

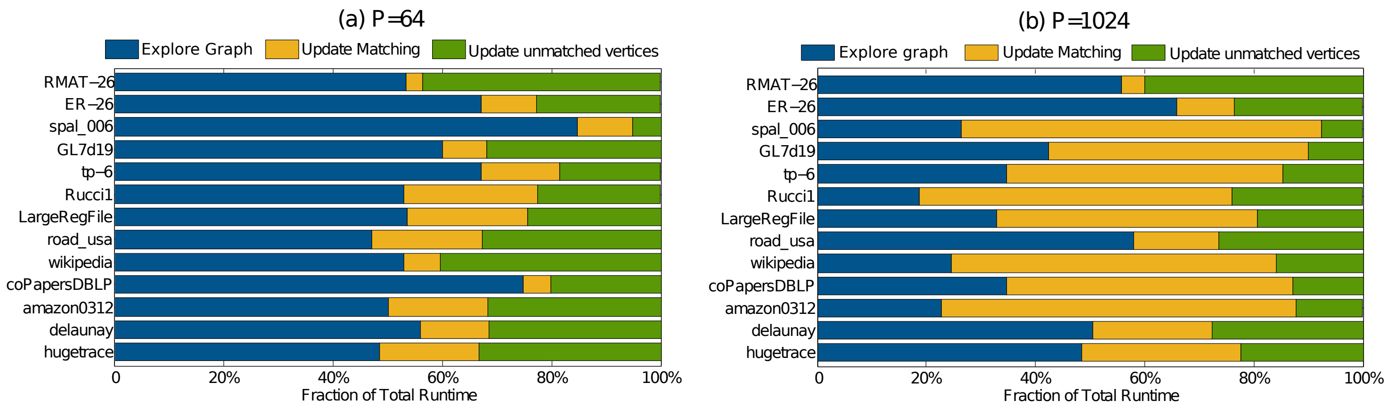

Fig. 10. Breakdown of time spent on different steps of the Dynamic Mindegree algorithm run on 64 and 1024 cores of Edison.

VII. Conclusion and Future Work

Using matrix-algebraic primitives, we present the first distributed-memory algorithms for maximal cardinality matching in bipartite graphs that scale to tens of thousands of processors. We represent three different algorithms in the same matrix algebraic framework, only with minimal modifications to the underlying semiring operations and data structures. All three algorithms benefit from fast parallel implementations of a handful of matrix-algebraic primitives that they are built upon. Unlike previous algorithms, ours maintain a stable approximation ratio that is insensitive to increasing concurrency, a trait that is important for exascale-level parallelism.

Finding a distributed data structure that can be used to traverse the bipartite graph from both sides without storing both $\mathbf{A}$ and $\mathbf{A}^{\mathsf{T}}$ would reduce the storage requirements by up to 50%. Such a data structure, Compressed Sparse Blocks (CSB) [26], which allows efficient SpMV on both $\mathbf{A}$ and $\mathbf{A}^{\mathsf{T}}$ without explicitly storing $\mathbf{A}^{\mathsf{T}}$, exists in shared memory. Developing a distributed-memory CSB, which can perform SpMV not just with dense vectors but also sparse vectors, is considered future work.

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] 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] 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] 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.
[5] 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.
[6] 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.
[7] 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.
[8] J. C. Setubal, "Sequential and parallel experimental results with bipartite matching algorithms," *Univ. of Campinas, Tech. Rep. IC-96-09*, 1996.
[9] J. Magun, "Greeding matching algorithms, an experimental study," *Journal of Experimental Algorithmics*, vol. 3, p. 6, 1998.
[10] J. Langguth, F. Manne, and P. Sanders, "Heuristic initialization for bipartite matching problems," *Journal of Experimental Algorithmics*, vol. 15, pp. 1–3, 2010.
[11] 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.
[12] R. M. Karp and M. Sipser, "Maximum matching in sparse random graphs," in *FOCS'81*. IEEE, 1981, pp. 364–375.
[13] M. Karpinski and W. Rytter, *Fast parallel algorithms for graph matching problems*. Clarendon Press, 1998, vol. 98.
[14] 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.
[15] D. P. Bertsekas and D. A. Castañon, "Parallel synchronous and asynchronous implementations of the auction algorithm," *Parallel Computing*, vol. 17, pp. 707–732, September 1991.
[16] M. Sathe, O. Schenk, and H. Burkhart, "An auction-based weighted matching implementation on massively parallel architectures," *Parallel Computing*, vol. 38, no. 12, pp. 595–614, 2012.
[17] U. V. Catalyurek, F. Dobrian, A. Gebremedhin, M. Halappanavar, and A. Pothen, "Distributed-memory parallel algorithms for matching and coloring," in *IPDPSW*. IEEE, 2011, pp. 1971–1980.
[18] F. Manne and R. H. Bisseling, "A parallel approximation algorithm for the weighted maximum matching problem," in *PPAM'07*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 708–717.
[19] M. Halappanavar, J. Feo, O. Villa, A. Tumeo, and A. Pothen, "Approximate weighted matching on emerging manycore and multithreaded architectures," *IJHPCA*, vol. 26, no. 4, pp. 413–430, 2012.
[20] F. Manne and M. Halappanavar, "New effective multithreaded matching algorithms," in *IPDPS*. IEEE, 2014, pp. 519–528.
[21] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *IJHPCA*, vol. 25, no. 4, 2011.
[22] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *SC'11*. ACM, 2011, pp. 65:1–65:12.
[23] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *IJHPCA*, vol. 19, no. 1, pp. 49–66, 2005.
[24] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, p. 1, 2011.
[25] "Graph500 benchmark," www.graph500.org.
[26] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *SPAA*. ACM, 2009, pp. 233–244.