A Matrix-Algebraic Formulation of Distributed-Memory Maximal Cardinality Matching Algorithms in Bipartite Graphs

Ariful Azad, Aydın Buluç

Computational Research Division, Lawrence Berkeley National Laboratory

Abstract

We describe parallel algorithms for computing maximal cardinality matching in a bipartite graph on distributed-memory systems. Unlike traditional algorithms that match one vertex at a time, our algorithms process many unmatched vertices simultaneously using a matrix-algebraic formulation of maximal matching. This generic matrix-algebraic framework is used to develop three efficient maximal matching algorithms with minimal changes. The newly developed algorithms have two benefits over existing graph-based algorithms. First, unlike existing parallel algorithms, cardinality of matching obtained by the new algorithms stays constant with increasing processor counts, which is important for predictable and reproducible performance. Second, relying on bulk-synchronous matrix operations, these algorithms expose a higher degree of parallelism on distributed-memory platforms than existing graph-based algorithms.

We report high-performance implementations of three maximal matching algorithms using hybrid OpenMP-MPI and evaluate the performance of these algorithm using more than 35 real and randomly generated graphs. On real instances, our algorithms achieve up to 200× speedup on 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 16,384 cores.

1. 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 maximum cardinality matching (MCM) 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]. A matching M is *maximal* if any edge not in M is added to M, it is no longer a matching. An algorithm that computes a maximal matching is an approximation algorithm, and the ratio of maximal to maximum cardinality is the approximation ratio of the maximal matching. The primary use case of a maximal cardinality matching is in the initialization of MCM algorithms because the former can be computed much faster than the latter [4, 5, 6, 7]. 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.

Computing a matching in parallel is an interesting research topic on its own. However, our primary interest is in solving sparse systems of linear equations where matching is used as a preprocessing step [2, 3]. 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] involves gathering the data into a single page memory node to run the serial (or multithreaded) matching

Email addresses: azad@lbl.gov (Ariful Azad), abuluc@lbl.gov (Aydın Buluç)

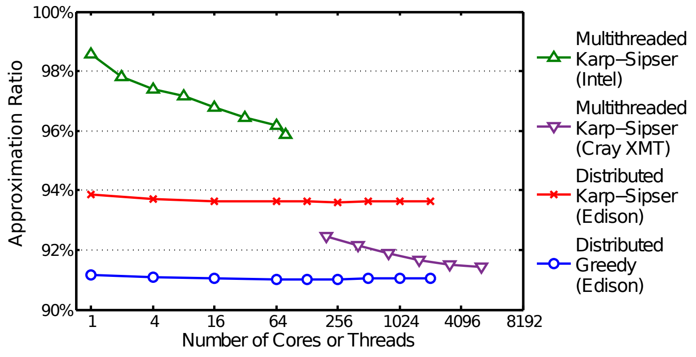

Figure 1: Matching qualities attained by Karp-Sipser and Greedy algorithms on `delaunay_n24` graph. Multithreaded algorithms are presented in earlier work [5], whereas distributed algorithms, which are matrix based, are presented here for the first time.

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 [8] 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.

In earlier work, effective serial and parallel algorithms for maximal cardinality matching have been designed and implemented on both shared and distributed memory systems [5, 9, 10, 11, 12]. To increase the cardinality of the maximal matching, existing serial and parallel algorithms process only a small fraction of unmatched vertices at a time, which imposes a vertex-processing order. This artificial ordering of vertices causes two important weaknesses of existing parallel algorithms. First, the cardinality of maximal matchings provided by existing algorithms can decrease significantly with increased concurrency because of suboptimal vertex orders. For example, Azad *et al.* [5] demonstrated that the approximation ratio (the ratio of maximal to maximum cardinality) of a multithreaded 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. Second, existing algorithms use asynchronous communication when searching for unmatched neighbors from a small subset of unmatched vertices. The difficulty of managing a large number of fine-grained asynchronous communication calls limits the scalability of existing matching algorithms.

In this paper, we address these two limitations of existing algorithms by redesigning them using matrix algebra. For this purpose, we represent the input bipartite graph by a sparse matrix and the vertex sets (including matchings) by vectors, and then decompose the matching algorithms into several independent steps. Next, sparse matrix-sparse vector multiplication (SpMSpV) is used to explore the neighborhood of unmatched vertices and vector operations are used to update the current matching. A slight modification in these matrix and vector operations give rise to three efficient maximal matching algorithms.

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 a diverse set of 33 real input graphs, we achieve an average of 121× speedup (up to 200×) on 2048 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. This is because the way we implement matrix operations leads itself to a bulk-synchronous

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 $M$
```

execution, where no explicit ordering among vertices are enforced within a single phase. Consequently, the algorithm is not effected by increased concurrency. 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. Since the primary use case of our algorithms is to initialize MCM algorithms, we extensively evaluate the impact of three maximal matching algorithms on a distributed-memory MCM algorithm. This paper expands on work first published as a conference paper [13].

2. 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 (MCM) if $|M| \geq |M'|$ for every matching M'.

2.1. 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 [14], 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.

2.2. 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 $C = \{c_1, c_2, ..., c_n\}$. 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.

Table 1: Basic functions needed for the maximal matching algorithms.

Function	Arguments	Returns	Example (x: sparse, y:dense, q: sparse)	Serial Complexity
IND	x: a sparse vector	local indices of nonzero entries of x	$x = [3, 0, 2, 2, 0]$ $\text{IND}(x) = [1, 3, 4]$	$O(nnz(x))$

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]
			$\text{SELECT}(x, y = \text{-}1) = [0, 0, 2, 0, 0]$
				$O(nnz(x))$

INVERT	x: a sparse vector
	assume $\max(x) \le len(x)$
		$z \leftarrow$ 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]
				$O(nnz(x))$

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

2.3. *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 zero 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 $\max(I) \le len(x)$, $x[I]$ 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.

2.4. *Operations on vectors and matrices*

Table 1 defines several operations on vectors and matrices, which will be used in matching algorithms. The function $\text{IND}(x)$ 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 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 1, 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 SpMSpV 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 SpMSpV $\mathbf{A} \cdot f_c$ over the *(select2nd, min)* semiring. SpMSpV returns the set of row vertices explored by f_c. In practice we use a *(select2nd, rand)* semiring where the 'addition' operation selects a random unmatched column in each row.

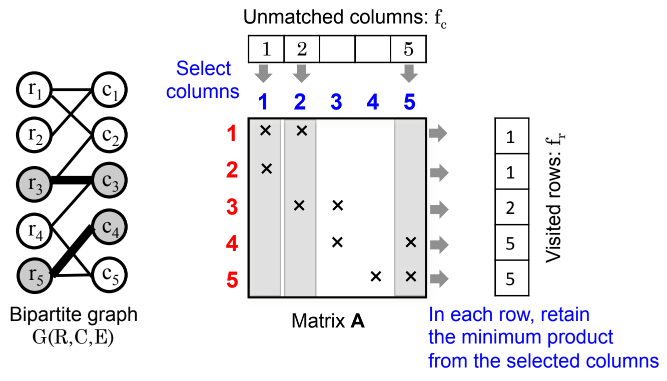

Figure 2: Illustration of traversing a bipartite graph $G(R, C, E)$ via SpMSpV. The bipartite graph with five row and five column vertices is shown in the left. Matched and unmatched vertices are shown in filled and empty circles, respectively. Thin lines represent unmatched edges and thick lines represent 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.

2.5. Related Work

There has been a large body of research on the theory of parallel matching algorithms, e.g., Karpinski and Rytter [15]. 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.* [12] 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× 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, 16]. However, their distributed-memory push-relabel algorithm did not scale well beyond 64 processors [16].

Parallel algorithms for weighted matching have also been studied [17]. Recently, Sathe *et al.* have reported 4× to 64× speedups on 1024 processors of a Cray XE6 for a parallel auction algorithm [18]. Half-approximation algorithms for weighted matching have been implemented on both shared and distributed memory computers with good speedups [19, 20, 21, 22].

3. Matrix Algebra based formulation of Matching Algorithms

3.1. 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 $\mathbf{A}$ 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.

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
    $d_c \leftarrow$ SPMSPV($\mathbf{A}^\mathsf{T}$,$f_r$, 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)
        $f_r \leftarrow$ SPMSPV($\mathbf{A}, f_c$, SR=(select2nd,rand))  ▷ 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 $f_c$
        $md_c \leftarrow$ SPMSPV($\mathbf{A}^\mathsf{T}$,$t_r$, SR=(select2nd,+))   ▷ Degrees of column vertices to the newly matched rows
        $J \leftarrow$ 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
```

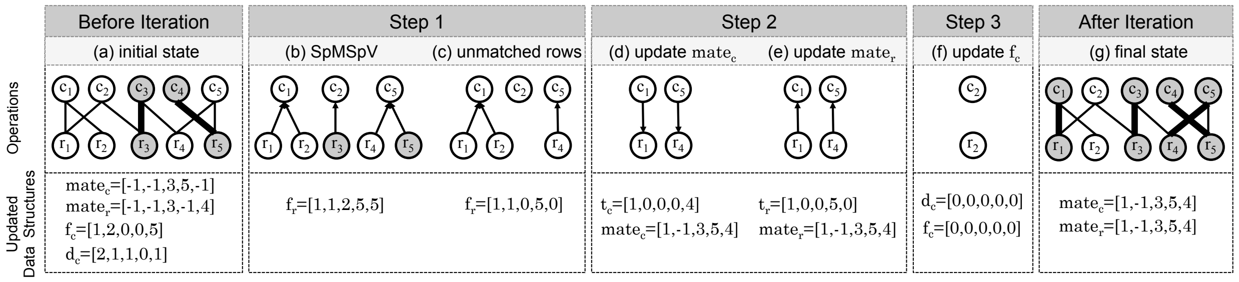

Figure 3: A working example of one iteration of a maximal matching algorithm described in Algorithm 2. Matched and unmatched vertices are shown in filled and empty circles, respectively. Thin lines represent unmatched edges and thick lines represent matched edges. The vectors f, d, and $mate$ represent 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.

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 SpMSpV over the *(select2nd, rand)* semiring. If we consider f_c to be the current frontier, the SpMSpV 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:     $f_r \leftarrow$ SPMSPV($\mathbf{A}, f_c^1$, SR=(select2nd,rand))
4: else                                                               ▷ Process other vertices
5:     $f_r \leftarrow$ SPMSPV($\mathbf{A}, f_c$, SR=(select2nd,rand))
```

3.2. 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.

3.3. 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 SPMSPV. 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.

3.4. Serial complexity

The computational complexity of every iteration of Algorithm 2 depends on the functions described in Table 1. Since SPMSPV dominates other operations in terms of serial complexity, it determines the serial runtime of the matrix-based algorithms. The cost of a single SpMSpV 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 1 as well.

First, note that two SPMSPV calls (lines 9 and 21) in Algorithm 2 use two different vectors. The first SPMSPV with the *(select2nd, rand)* or *(select2nd, mindegree)* semiring uses unmatched columns in x. By contrast, the second SPMSPV with the *(select2nd, +)* semiring uses the newly matched row vertices as the vector. Since the the set of newly matched vertices is a subset of unmatched vertices, the cost of the first

SpMSpV is often higher than the other. Hence, we only discuss the cost of the first SpMSpV. (However, two SpMSpVs traverse the graph from opposite directions, hence their costs also depend on the nonzero structure.)

In the first iteration, the SpMSpV 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. 5). However, all of our algorithms spend most of their time in the first iteration. For example, Fig. 5 shows that every algorithm spends at least 18% of their total runtime in the first iteration on `GL7d19`.

4. Distributed memory parallel algorithm

4.1. Data distribution and storage

We use the CombBLAS framework [23] 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 $P(:,i)$.

4.2. 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 SpMSpV algorithm for the case of sparse input and output vectors has previously been used in the specialized context of distributed memory BFS [24], which we leverage here. A 2D SpMSpV algorithm for the case of dense vectors is also provided in CombBLAS. As discussed before, serial SpMSpV 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 SpMSpV 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 SpMSpV has cost

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

using the ring algorithm, which is the default algorithm for large ($\geq$ 512KB) messages on many MPI implementations such as MPICH [25]. 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

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

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

$$T_{\text{Invert}} = nnz(x_{ij}) + \alpha(p-1) + \beta\, nnz(x_{ij}) = \frac{n}{p} + \alpha(p-1) + \beta\frac{n}{p}$$

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 SpMSpV 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 SpMSpV as:

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

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, SpMSpV 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 SpMSpV would make our algorithm even more scalable.

5. Experimental Setup

5.1. 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 OpenMP for intra-node multithreading and compiled the code with gcc 4.9.2 with `-O2 -fopenmp` flags. Cray's MPI implementation on Edison is based on MPICH2. 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.

5.2. Input Graphs

Table 2 describes a representative set of matrices from the University of Florida sparse matrix collection [26]. We consider a diverse collection of rectangular, unsymmetric and symmetric matrices so that our conclusions apply to most practical matrices. To test the performance of maximal matching algorithm on larger matrices, we used RMAT [27], the Recursive MATrix generator to generate two classes of synthetic matrices: (a) G500 matrices representing graphs with skewed degree distributions from Graph 500 benchmark [28], and (b) ER matrices representing Erdős-Rényi random graphs with uniform degree distributions. A scale n synthetic matrix is 2^n-by-2^n. To generate G500 matrices, we use RMAT seed parameters a=.57, b=c=.19, and d=.05. On average, G500 and ER matrices have 16 nonzeros per row and column. For example, a scale-30 G500 matrix (`G500-30`) has 1 billion rows, 1 billion columns, and 16 billion nonzeros.

Table 2: Test problems for evaluating the matching algorithms.

Class	Graph	#Rows (m) ($\times 10^6$)	#Columns (n) ($\times 10^6$)	nnz ($\times 10^6$)	Description
Rectangular	watson_2	0.35	0.68	1.85	linear programming problem
	wheel_601	0.90	0.72	2.17	combinatorial problem
	stormG2_1000	0.53	1.38	3.46	linear programming problem
	LargeRegFile	2.11	0.80	4.94	circuit simulation problem
	cont1_l	1.92	1.92	7.03	linear programming problem
	Rucci1	1.98	0.11	7.79	least squares problem
	degme	0.18	0.66	8.13	linear programming problem
	tp-6	0.14	1.01	11.53	linear programming problem
	rel9	9.89	0.27	12.67	combinatorial problem
	GL7d19	1.91	1.96	37.32	combinatorial problem
	relat9	12.36	0.55	38.96	combinatorial problem
	spal_004	0.01	0.32	46.17	linear programming problem
Unsymmetric	amazon0312	0.40	0.40	3.20	electromagnetics problem
	t2em	0.92	0.92	4.59	electromagnetics problem
	ohne2	0.18	0.18	6.87	semiconductor device problem
	atmosmodm	1.49	1.49	10.32	computational fluid dynamics problem
	rajat31	4.69	4.69	20.32	circuit simulation problem
	FullChip	2.98	2.98	26.62	circuit simulation problem
	RM07R	0.38	0.38	37.46	computational fluid dynamics problem
	circuit5M	5.56	5.56	59.52	circuit simulation problem
	ljournal-2008	5.36	5.36	79.02	LiveJournal social network
	cage15	5.15	5.15	99.20	DNA electrophoresis problem
	HV15R	2.02	2.02	283.07	computational fluid dynamics problem
	it-2004	41.29	41.29	1150	2004 web crawl of .it domain
	sk-2005	50.64	50.64	1949	2005 web crawl of .sk domain
Symmetric	coPapersDBLP	0.54	0.54	60.98	Citation networks in DBLP
	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	178.61	electromagnetics problem
	delaunay_n24	16.77	16.77	201.33	delaunay triangulations of random points
	europe_osm	50.91	50.91	216.22	Europe street networks
	rgg_n_2_24_s0	16.78	16.78	530.22	undirected random graph
	nlpkkt240	27.99	27.99	1531	symmetric indefinite KKT matrix
Random	ER-26	67.11	67.11	1024	Erdős-Rényi random graphs
	RMAT-26	67.11	67.11	1024	RMAT random graphs (param: .57,.19,.19,.05)

6. Results

6.1. Approximation ratios obtained by maximal matching algorithms

A maximal matching with higher approximation ratio often leaves less work for a MCM algorithm. Therefore, it can be used to measure the quality of a maximal matching. Fig. 4 shows the approximation ratios obtained by three parallel matching algorithms for different matrices on 1024 cores of Edison. In every case, each algorithm matches at least 80% of the vertices matched in a maximum matching. Fig. 4 shows that our Karp-Sipser implementation attains the best approximation ratio for about 65% of the matrices from Table 2. The Greedy algorithm returns with the best approximation ratio only for `ohne2` matrix whereas Dynamic Mindegree provides the best cardinality for the rest of the problems. In the next subsection we discuss the impact of maximal matching on the time to compute a maximum matching.

In our prior work [13], we compared our matrix based algorithms to serial matching software by Duff *et al.* [4]. We showed that the approximation ratio obtained by our algorithms are slightly smaller than serial graph-based algorithms that match one vertex at a time. Especially 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. By contrast, our matrix-based

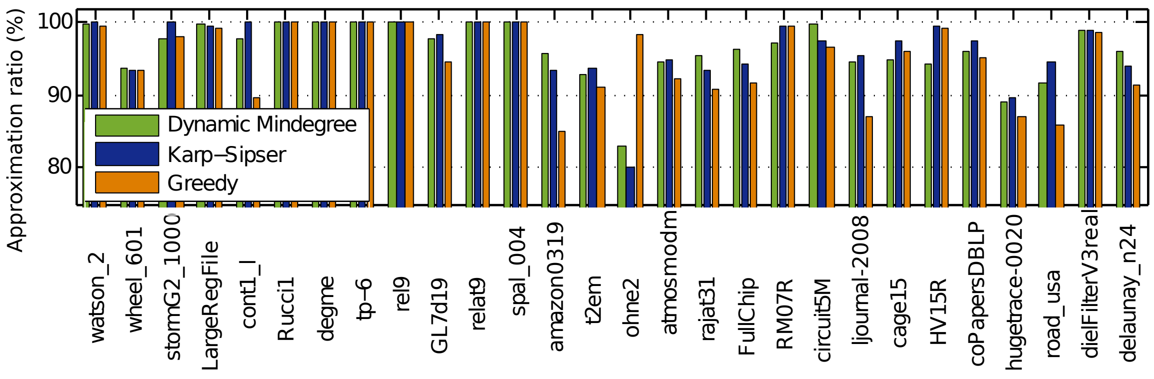

Figure 4: Approximation ratios attained by Greedy, Karp-Sipser, and Dynamic Mindegree algorithms on 1024 cores of Edison.

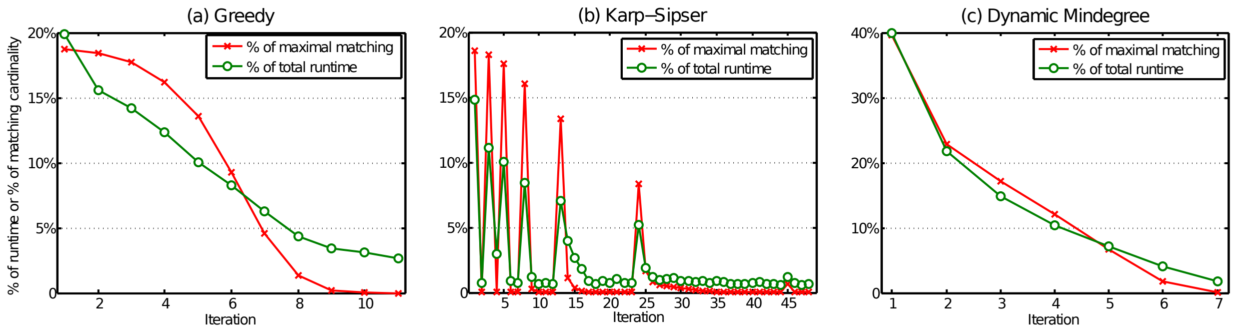

Figure 5: Fraction of time spent and fraction of vertices matched (percentage of the final maximal matching cardinality) in each iteration of the matching algorithms on `GL7d19` graph on 1024 processors.

algorithms do not impose any ordering of vertices and process all unmatched vertices simultaneously in order to increase concurrency. Therefore, the matching quality of the latter is often slightly smaller than serial graph-based algorithms. However, being free from a vertex-processing order, matrix-based algorithms are able to maintain the same approximation ratio on all concurrencies. On the other hand, the cardinality of parallel graph-based algorithms may decrease rapidly with increased concurrency [5]. This behavior is explained with an example in Fig. 1 where a cardinality of a multithreaded Karp-Sipser algorithm decreases by more than 3% on several thousands of threads. Even worse, the cardinality (as well as runtime) of multithreaded algorithms fluctuate from one run to another on the same concurrency due to dynamic allocation of vertices to thread and the scheduling of threads. By contrast, matchings obtained from matrix-based algorithms are reproducible on all concurrency.In our implementation, we randomly permute input matrices for load balance. If different permutations are used on different concurrencies, we might observe a small variation in matching cardinality (less than 0.1% [13]). This small variation can in practice be completely eliminated if the random permutations are fixed for all concurrencies.

6.2. Progression of algorithms

The amount of work performed in an iteration of maximal matching algorithms varies considerably from one iteration to another. The iterations of Karp-Sipser come in batches of k iterations. First $k-1$ iterations of each batch match degree-1 vertices (line 3 of Algorithm 3) followed by one iteration of the Greedy algorithm when no degree-1 vertices are available (line 5 of Algorithm 3). For example, iterations 8-12 is a batch of iterations in Fig. 5(b). By contrast, the Greedy and Dynamic Mindegree algorithms pack several iterations of Karp-Sipser into a single iteration because the former algorithms do not have any restriction when processing unmatched vertices. Therefore, Greedy and Dynamic Mindegree often require fewer iteration than Karp-Sipser and their workloads decrease consistently from one iteration to the next as shown in Fig. 5 for `GL7d19` graph.

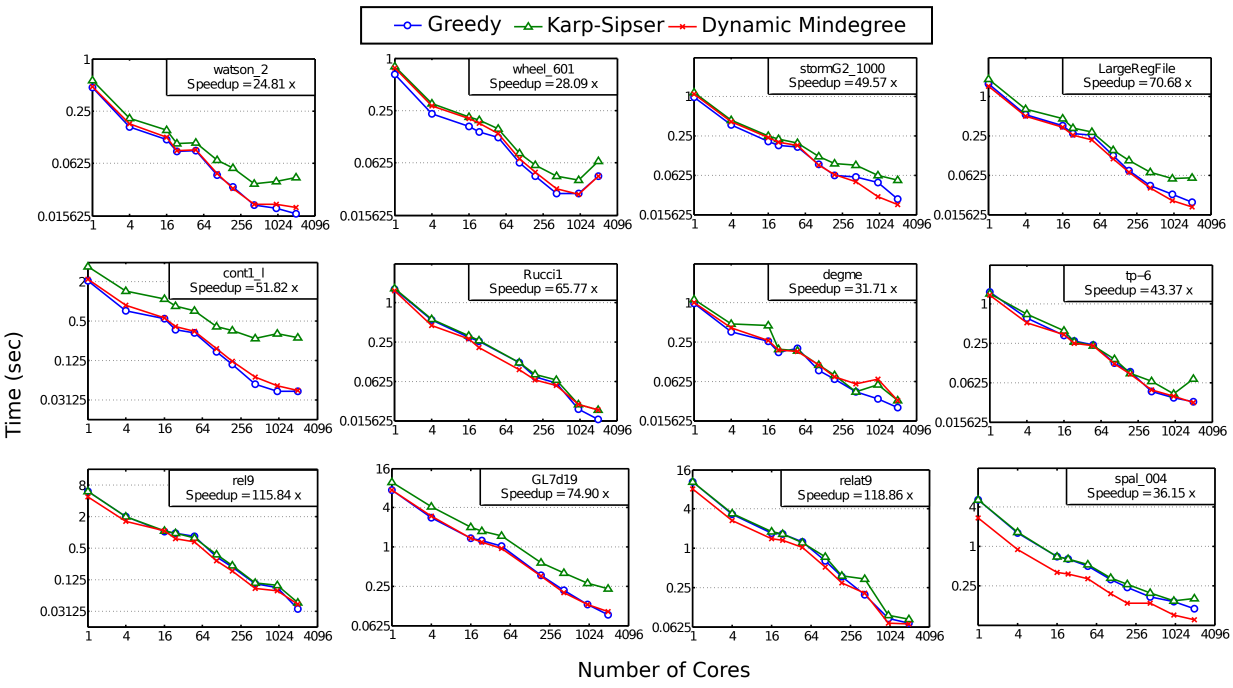

Figure 6: Strong scaling of maximal matching algorithms on 12 rectangular matrices on Edison.

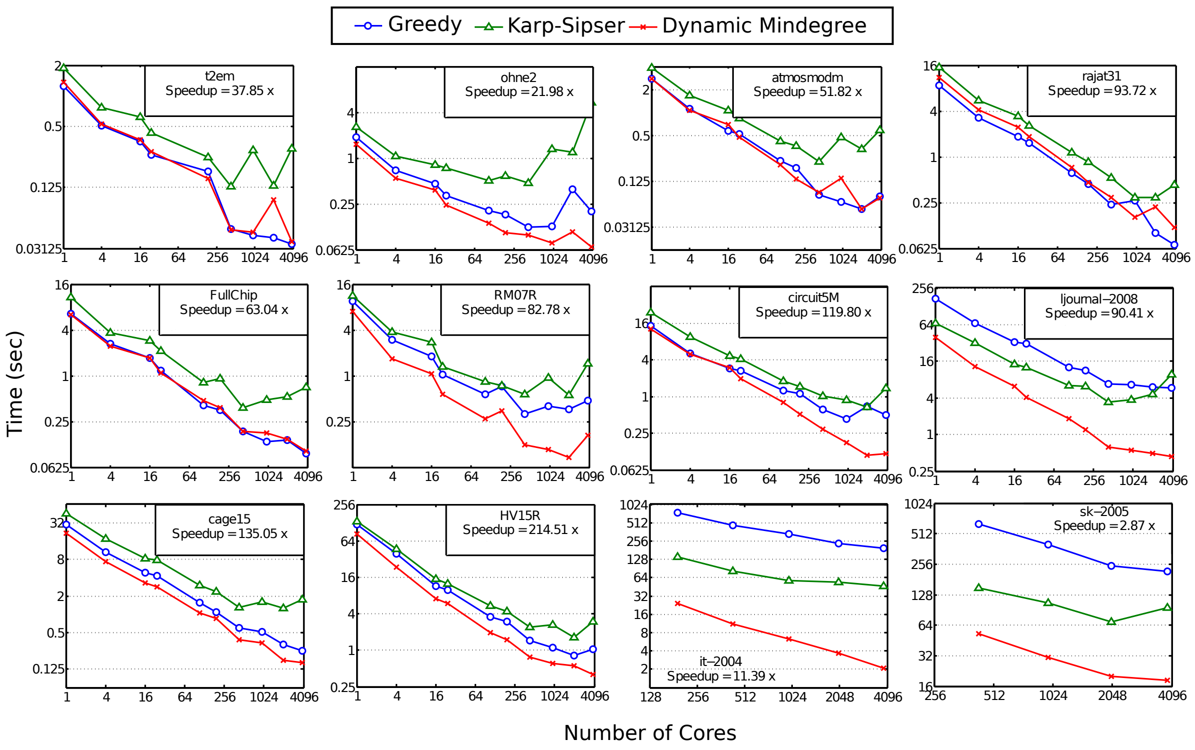

Figure 7: Strong scaling of maximal matching algorithms on 12 unsymmetric matrices on Edison. The speedup is reported as the ratio of runtime on the lowest and highest number of cores on which a particular matrix was run.

The number of iterations influences the runtime of maximal matching algorithms significantly. A higher number of iterations means more synchronization steps and less work per processor in each iteration, which negatively impacts the performance. The number of iterations needed by each algorithm is influenced by input graphs. On some graphs Karp-Sipser takes many more iterations than its competitors. For example, on `GL7d19`, Greedy algorithm takes 11, Karp-Sipser takes 48, and Dynamic Mindegree takes 7 iterations before completion (Fig. 5), and on `ohne2`, Greedy algorithm takes 33, Karp-Sipser takes 370, and Dynamic Mindegree takes 14 iterations. Consequently, on these problems Karp-Sipser performs badly on high concurrency as can see seen in Fig. 10. On other graphs, all of these algorithms take the same number of iterations. For example, all three algorithms take 3-4 iterations on `Rucci1`. Hence, on this graph, their performance is very similar as shown in Fig. 10 and Fig. 6.

6.3. Scalability

We show the strong scaling of parallel maximal matching algorithms for rectangular matrices in Fig. 6, for unsymmetric matrices Fig. 7, and for symmetric matrices Fig. 8. Matching algorithms achieve more than 90× speedups on 50% of real matrices when we go from 1 core to 1024 cores of Edison. Our algorithms scale better on larger matrices than on smaller matrices, as expected. For example, the Dynamic Mindegree algorithm attains 238× speedup on `hugetrace-00020`, but only 25× on `watson_2`. For these small graphs, processors do not have enough work on higher concurrency, which limits the performance.

The scalability of matrix-based algorithm on higher number of processors can be realized on larger synthetic graphs. Fig. 9 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 16 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× speedup when we go from 1,024 to 16,384 processors. By contrast, on `RMAT-26`, Greedy algorithm achieves only 3× 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. 9. When we go from 1,024 to 16,384 cores (i.e., 16× increase), Dynamic Mindegree achieves about 15×, 10×, and 6× speedups on `RMAT-30`, `RMAT-28`, and `RMAT-26`, whereas Karp-Sipser achieves about 8× and 3× speedups on `RMAT-30` and `RMAT-28`, and no speedup on `RMAT-26`. The greedy algorithm seats 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).

The weak scaling of the matrix-based algorithms is also good when compute maximal matchings on RMAT synthetic graphs from scale 28 to 32 on 526 to 8192 cores of Edison [13]. We observe that the number of processors increased by 16× slows down our algorithms by a factor less than 2×. Hence, the matrix-based algorithms can utilize even larger number of processors if large enough graphs become available.

6.4. Impact of in-node multithreading and randomization in selecting parents

We use in-node multithreading in all primitives in implementing maximal matching algorithms. When 12 threads are used in each MPI process, we observe a performance increase of 2-4× relative to nonthreaded implementation presented in the conference paper [13]. In addition, the new implementation scales up to 4096 cores whereas the previous flat MPI versions stopped scaling at 1024 cores on most real matrices.

We also observed that randomization in selecting parents helps Karp-Sipser and Greedy algorithms. Consequently, we use the (Select2nd, rand) semiring, as opposed to the (Select2nd, min) semiring that was used in the conference paper [13]. The new approach results in decreased computation time for 60% of the problems for the Karp-Sipser algorithm and 80% of the problems for the Greedy algorithm. These savings are primarily due to the new implementation taking fewer iterations to complete. Similarly, the cardinality of the matchings also increased as a result this modification. Due to randomization in selecting parents, average cardinalities of the Karp-Sipser and Greedy algorithms are increased by 0.4% and 0.8%, respectively.

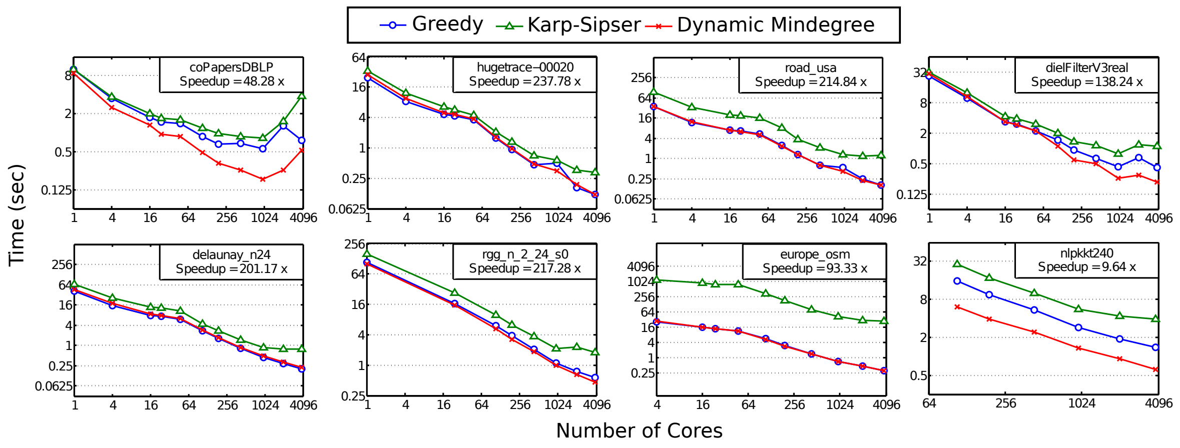

Figure 8: Strong scaling of maximal matching algorithms on 8 symmetric matrices on Edison. The speedup is reported as the ratio of runtime on the lowest and highest number of cores on which a particular matrix was run.

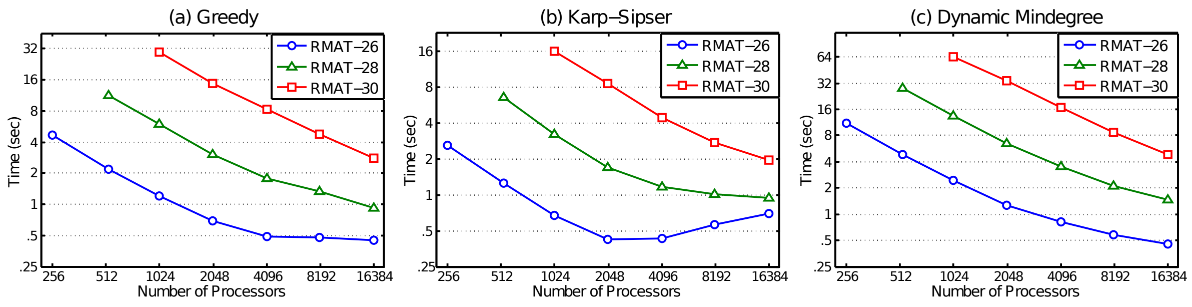

Figure 9: Strong scaling of maximal matching algorithms on RMAT graphs.

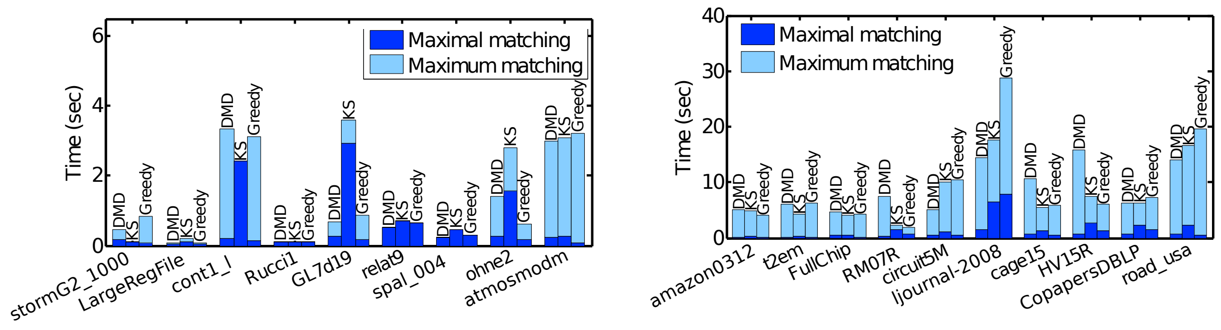

Figure 10: Time taken to compute a distributed-memory maximum cardinality matching when initialized by Greedy, Karp-Sipser, and Dynamic Mindegree algorithms on 1024 cores of Edison. A subset of problems were selected from Table 2. Smaller and larger matrices are shown in the left and right subfigures, respectively.

6.5. Impact of maximal matching on MCM

The total runtime of an MCM algorithm often decreases when it is initialized by a maximal matching with high approximation ratio [4, 5, 11] because the latter can be computed much quickly than the former. We demonstrate the impact of three maximal matching algorithms on the runtime of a distributed-memory MCM algorithm in Fig. 10 where the dark and light blue colors denote runtimes to compute maximal and maximum matchings, respectively. The distributed-memory MCM algorithm is described in our recent paper [8], and its source code is publicly available as part of Combinatorial BLAS library [29]. Fig. 10 shows that larger cardinalities of maximal matchings often lead to faster runtime of the MCM algorithm (light blue color). For example, when Karp-Sipser returns a matching with the highest approximation ratio, the runtime of the MCM algorithm initialized by the Karp-Sipser algorithm is often the fastest. However, the total time to compute an MCM, which is the summation of runtimes of maximal and maximum matching algorithms, is determined by both the approximation ratio and runtime of the maximal matching algorithms. For example, for `ljournal-2008` matrix, Karp-Sipser attains the highest approximation ratio (Fig. 4), and the runtime of MCM algorithm initialized by Karp-Sipser is the smallest (Fig. 10). However, the total time to compute MCM on `ljournal-2008` is the smallest when we initialize the MCM algorithm by Dynamic Mindegree because of the higher runtime of Karp-Sipser. In summary, the total time to compute an MCM (maximal + maximum matchings) is the smallest for 55% of matrices in Table 2 when Dynamic Mindegree is used, for 35% matrices when Karp-Sipser is used, and 10% matrices when Greedy algorithm is used.

7. 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.

Distributed-memory Karp-Sipser is almost always the slowest among the three algorithms we considered as shown in Figs. 6, 7, and 8 (see the discussion in Subsection 6.2 for empirical reasons). Karp-Sipser's slow runtime combined with its poor scalability makes it less attractive on higher concurrency despite it yielding the highest approximation ratio for most practical problems as shown in Fig. 4. The greedy algorithm is often not used in practice because of its small approximation ratio. Hence, we argue that the dynamic mindegree algorithm be used on higher core count when computing a maximal matching on a new graph. However, dynamic mindegree is not the universally best option because the approximation ratio attained by these algorithms depends heavily on the structure of the graph where the matching is being computed.

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) [30], which allows efficient SpMSpV 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 SpMSpV 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] A. Azad and A. Buluç, “Distributed-memory algorithms for maximum cardinality matching in bipartite graphs,” in *IPDPS*. IEEE, 2016.

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

[10] J. Magun, “Greeding matching algorithms, an experimental study,” *Journal of Experimental Algorithmics*, vol. 3, p. 6, 1998.

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

[12] 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.

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

[14] R. M. Karp and M. Sipser, “Maximum matching in sparse random graphs,” in *FOCS’81*. IEEE, 1981, pp. 364–375.

[15] M. Karpinski and W. Rytter, *Fast parallel algorithms for graph matching problems*. Clarendon Press, 1998, vol. 98.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] F. Manne and M. Halappanavar, “New effective multithreaded matching algorithms,” in *IPDPS*. IEEE, 2014, pp. 519–528.

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

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

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

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

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

[28] “Graph500 benchmark,” www.graph500.org.

[29] “Combinatorial BLAS 1.5,” http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/.

[30] 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.