Parallel Graph Algorithms in Constant Adaptive Rounds: Theory meets Practice

Soheil Behnezhad
University of Maryland
soheil@cs.umd.edu

Laxman Dhulipala[*]
MIT CSAIL
laxman@mit.edu

Hossein Esfandiari
Google Research
esfandiari@google.com

Jakub Lacki
Google Research
jlacki@google.com

Vahab Mirrokni
Google Research
mirrokni@google.com

Warren Schudy
Google Research
wschudy@google.com

ABSTRACT

We study fundamental graph problems such as *graph connectivity*, *minimum spanning forest* (MSF), and approximate *maximum (weight) matching* in a distributed setting. In particular, we focus on the *Adaptive Massively Parallel Computation* (AMPC) model, which is a theoretical model that captures MapReduce-like computation augmented with a distributed hash table.

We show the first AMPC algorithms for all of the studied problems that run in a constant number of rounds and use only $O(n^\epsilon)$ space per machine, where $0 < \epsilon < 1$. Our results improve both upon the previous results in the AMPC model, as well as the best-known results in the MPC model, which is the theoretical model underpinning many popular distributed computation frameworks, such as MapReduce, Hadoop, Beam, Pregel and Giraph.

Finally, we provide an empirical comparison of the algorithms in the MPC and AMPC models in a fault-tolerant distributed computation environment. We empirically evaluate our algorithms on a set of large real-world graphs and show that our AMPC algorithms can achieve improvements in both running time and round-complexity over optimized MPC baselines.

1 Introduction

The *Adaptive Massively Parallel* (AMPC) model has been recently introduced by Behnezhad *et al.* [19]. It is an extension of the *Massively Parallel Computation* (MPC) model [16, 41, 44], which captures popular parallel processing frameworks,

[*]Work done as a student researcher at Google Research.

such as Mapreduce [31], Hadoop [8], Flume [4, 27], Beam [7], Giraph [62] and Pregel [51].

The MPC model has been extensively studied in theory in recent years [3, 5, 6, 9, 15, 17, 18, 22, 30, 36, 46, 49, 60], and its theoretical capabilities and limitations are relatively well-understood. In the context of graph algorithms, a significant limitation of the model is, roughly speaking, the fact that initially each node only knows its immediate neighbors, and exploring a larger neighborhood requires multiple rounds. This phenomenon is formalized in the widely believed 1-VS-2-CYCLE conjecture [37], which states that in the MPC model, distinguishing between a cycle of length n and two cycles of length $n/2$ requires $\Omega(\log n)$ computation rounds. The conjecture implies a number of conditional lower bounds for fundamental graph problems such as matching, independent sets, coloring, connectivity, etc., see [21, 37] or a recent result of [56] for an extensive overview of the implications of the conjecture. The main motivation behind the AMPC model is to alleviate such hardness results.

At a high level, in the MPC model computation is distributed among a number of machines and proceeds in synchronous rounds. The space available on each machine is assumed to be much smaller than the size of the input. Within each round, each machine may send (a limited number of) messages to other machines, and all these messages constitute the input to the following round, i.e., they are delivered only when the next round begins. Hence, each machine can only access a limited part of the input at each round, that is, only the messages that it receives.

The AMPC model *extends* the MPC model by storing all messages sent in a single round in a distributed read-only hash table (also known as a distributed *key-value store*). In the following round, all machines have *random read-access* to the hash table, subject to the same constraints on the amount of communication as in the MPC model. As shown in [19], this extension is useful in designing graph algorithms, as, intuitively, it allows a machine knowing the identity of a vertex to explore its local neighborhood. (On the other hand, Charikar *et al.* [28] showed that some of the known unconditional lower bounds in the MPC model apply to the AMPC model as well.)

The AMPC model is inspired by empirical studies of large-scale graph computations that utilize a distributed hash table [15, 45]. In the paper introducing the model, Behnezhad *et al.* [19] argued that the AMPC model is realis-

Table 1: Summary of our results, compared to the results of [19]. We assume the available space per machine is n^δ for some constant $\delta < 1$. We consider the setting where the space per machine is sublinear in the number of vertices of the graph, that is, $S = O(n^\epsilon)$. All results require $\tilde{O}(m)$ total space over all machines (except the fourth row). D denotes the diameter of the graph. We note that an algorithm for the minimum spanning tree problem also solves the connectivity problem.

Problem	AMPC – **This Paper**		AMPC – [19]	MPC
	Bound	Implementation?	Previous Bound	
Connectivity	$O(1)$		$O(\log\log_{m/n} n)$	$O(\log D + \log\log_{m/n} n)$ [21]
Minimum spanning tree	$O(1)$	✓	$O(\log\log_{m/n} n)$	$O(\log n)$
Matching	$O(\log\log n)$		—	$\widetilde{O}(\sqrt{\log n})$ [38]
Matching (with $O(m + n^{1+\epsilon})$ space*)	$O(1)$	✓	—	$\widetilde{O}(\sqrt{\log n})$ [38]
Maximal independent set		✓	$O(1)$	$\widetilde{O}(\sqrt{\log n})$ [38]
1-vs-2-Cycle		✓	$O(1)$	$O(\log n)$ (folklore)

tic, that is, it can be implemented by augmenting a parallel processing framework, such as MapReduce, Hadoop or Beam, with a distributed hash table. At the same time, the hybrid system would maintain the fault-tolerance properties of the framework used and should not suffer from query contention.

Implementing the AMPC model efficiently relies on using an efficient distributed hash table. Developing a high performance distributed hash table has become a popular topic of research in the systems community in recent years. The existing implementations are able to support hundreds of millions of queries per second, with query latency as low as a few microseconds [33]. These technologies are increasingly available today, both at a hardware level [11, 29], and at the level of software systems such as eRPC [42], and key-value databases built over RDMA [34, 54].

The results of Behnezhad *et al.* [19] show that that using AMPC model allows one to give algorithms running in significantly fewer rounds compared to the best solutions in the MPC model. However, up until today it has not been studied how these gains would translate to the empirical running times. In this paper we provide the first empirical evaluation of the AMPC model, and further improve the theoretical results by Behnezhad [19], by giving AMPC algorithms for basic graph problems that run in only $O(1)$ rounds.

1.1 Our Contribution

In this paper, we study several fundamental graph problems in the AMPC model, present the first distributed algorithms for these problems that run in a constant number of rounds, and provide the first empirical evaluation of this model. Our results along with a comparison with the state-of-the-art both in the MPC model and the AMPC model are provided in Table 1. Below, we elaborate more on these results.

Connected components. Our first result is an AMPC algorithm for computing the *connected components* of an undirected graph in $O(1)$ rounds. We note that finding connected components, apart from being a basic graph problem, has a number of practical applications and is one of the most commonly used graph-processing analytics [61].

Note that before the AMPC model has been introduced, the problem of finding connected components has been studied in a setting very similar to the AMPC model itself. Namely, [45] gave an algorithm running in $O(\log n \log\log n)$ rounds and showed its good practical performance. However, it has been later shown that the empirical running time can be further improved by using an MPC algorithm [48]. We note that in the MPC model, the best known algorithm for finding connected components runs in $O(\log D + \log\log_{m/n} n)$ rounds, where D is the diameter of the input graph [21].

Minimum spanning forest. Our next notable result is an AMPC algorithm for computing *minimum spanning forest* (MSF) in a constant number of rounds. This problem has a number of applications in clustering [15]. In particular, one can use this algorithm together with a simple sorting step, and our connectivity algorithm to find any desired level of a single-linkage hierarchical clustering [71]. Similarly to the problem of connectivity, finding MSF can be solved in $O(\log n)$ rounds in the MPC model (or slightly faster, if one assumes that the diameter of each tree in the minimum spanning forest is limited [6, 21]).

Our algorithms for finding connected components and MSF that run in $O(1)$ rounds directly improve upon the state-of-the-art AMPC algorithms by Behnezhad *et al.* [19] that run in $O(\log\log_{m/n} n)$ rounds. At the same time, under 1-vs-2-Cycle conjecture [37], both problems require $\Omega(\log n)$ rounds in the MPC model (or $\Omega(\log D)$ rounds if we parametrize the problem by graph diameter D [21]). Hence, with respect to the number of rounds, our results improve significantly over the best MPC solutions.

Matchings. Furthermore, we give two AMPC algorithms for the maximal matching problem. Both algorithms require $O(n^\epsilon)$ space per machine where $\epsilon \in (0, 1)$ can be any arbitrarily small constant. The first algorithm uses $\widetilde{O}(m + n)$ total space and $O(\log\log n)$ rounds. The second algorithm uses $O(m + n^{1+\epsilon})$ total space and takes $O(1)$ rounds. The best known previous algorithm in this regime of space can be obtained by simulating the MPC algorithm of [38] which requires $\widetilde{O}(\sqrt{\log n})$ rounds. We note that an $O(\log\log n)$ round MPC maximal matching algorithm is also known [22], which unlike ours, requires $\Omega(n)$ space per machine.

Empirical evaluation. Finally, we provide the first empirical evaluation of the AMPC model in a fault-tolerant distributed setting. We experimentally evaluate four AMPC algorithms for the Maximal Independent Set (MIS), Maximal Matching (MM), Minimum Spanning Forest, and 1-vs-2-Cycle problems, and compare these algorithms with state-of-the-art MPC baselines for these problems. We implemented all of our algorithms (both in MPC and AMPC models) in Flume-C++ [4], which is a highly optimized dataflow framework similar to Apache Beam. The AMPC algorithms were additionally allowed to use a distributed hash table service. We provide a detailed experimental study of our implementations in a shared production data center using a maximum of 100 machines. We investigate the effect of enabling caching and multithreading optimizations which enable good practical performance for AMPC algorithms. Finally, we study the performance of our AMPC

algorithms relative to strong MPC baselines and show significant speedups for all problems that we study. Our results provide promising evidence for the practical applicability of algorithms in the AMPC model.

2 Model

In this section we give a concise formal definition of the *Adaptive Massively Parallel Computation (*AMPC*)* model (the model used in this paper is identical to the model defined in [19]). In the AMPC model, we are given an input of size N, which is processed by a collection of P machines each having space S. The total space used by the computation is just the total space of all machines, i.e., $T = S \cdot P$. We assume that $S = \Theta(N^\epsilon)$, where $\epsilon \in (0, 1)$ is a constant. We note that while the running times of our algorithms often depend on $1/\epsilon$ we often omit this dependency, since $O(1/\epsilon) = O(1)$.[1]

In addition to the input and machines, in the AMPC model there is collection of *distributed hash tables (DHT)* that we denote by $\mathcal{D}_0, \mathcal{D}_1, \mathcal{D}_2, \ldots$. Each hash table stores a collection of key-value pairs: given a key a DHT returns all corresponding values. We require each key-value pair to use a constant number of words. At the start of the computation, the input data is stored in $\mathcal{D}_0$ and uses a set of keys known to all machines (e.g., consecutive integers).

An AMPC computation consists of a number of *rounds*. In the i-th round, each machine can read data from $\mathcal{D}_{i-1}$ and write to $\mathcal{D}_i$. Within a single round, each machine can make up to $O(S)$ reads (*queries*) and emit $O(S)$ *writes*, and perform arbitrary computation. Each query and write queries/writes a single key-value pair. The total *communication* that a machine performs per round is equal to the total number of queries and writes. A salient aspect of the model is that the total communication of each machine within a round is bounded by $O(S)$ which is strictly sublinear in the input size. We note that all of the algorithms described in both [19] and this paper perform near-linear computation per machine in each round, making the algorithms proposed in both papers work-efficient up to poly-logarithmic factors. However, the AMPC model does not place a bound on the amount of of computation permitted on a machine, as this is a usual assumption of the MPC model.

Fault-Tolerance An important characteristic of the AMPC model is that it is amenable to *fault tolerant* implementation, which is one of the key features of the MPC model, and one of the reasons underpinning its widespread adoption. A fault tolerant implementation of AMPC can be derived by observing that each DHT can be made fault-tolerant [19].

Caching and Query Contention One natural concern related to implementing the AMPC algorithms is the possibility that all S machines query for the same key, potentially causing contention on a single machine, or set of machines storing the key. The authors of [19] provide an algorithmic justification for how any set of queries made by the machines in a given round can be handled with negligible overall contention by caching the results of queries on each machine. In this paper, we empirically verify that caching is required for good performance in the AMPC model (see Section 5.3).

Relationship to other models The AMPC model is closely related to the MPC model. Indeed, every MPC algo-

rithm can be simulated by a corresponding AMPC algorithm in the same total space and round-complexity ([19] provides a description of the simulation). Due to known simulations of PRAM algorithms on MPC [41, 44], the AMPC model is also able to simulate existing PRAM algorithms in the EREW, CREW, CREW, and MultiPrefix PRAM models [25].

3 Connectivity & Minimum Spanning Forest

In this section we give our algorithms for computing minimum spanning forest (MSF), as well as for computing connected components. We provide full proofs in the full version of the paper [20]. Formally, we prove the following result.

THEOREM 1. *There is an* AMPC *algorithm for computing minimum spanning forest and connected components of an undirected weighted graph in $O(1)$ rounds using total space $T = O(m + n \log^2 n)$, w.h.p.*

Our result for connectivity can be actually obtained from the algorithm for minimum spanning forest. Indeed, once we find any spanning forest, the connected components can be found by applying the forest connectivity algorithm of [19] which takes $O(1)$ rounds and uses $O(n \log n)$ queries. Hence, in the following part of this section we focus on an algorithm for computing MSF in $O(1)$ rounds of AMPC. Our algorithm improves on the $O(\log \log_{T/n} n)$ round connectivity and MSF algorithms of Behnezhad *et al.* [19] whenever $T = n^{1+o(1)}$.

Algorithm 2 provides the pseudocode for our MSF algorithm. If the input graph is dense, i.e. $m = \Omega(n^{1+\epsilon})$, the algorithm runs the algorithm of Behnezhad *et al.* which finishes in $O((1/\epsilon) \log(1/\epsilon))$ rounds [19]. To handle the case when $m = o(n^{1+\epsilon})$, the algorithm ternarizes the graph, that is, replaces every vertex of degree $k > 3$ with a cycle of length k, with each edge associated with one vertex on the cycle. The ternarization step ensures that every vertex has degree ≤ 3, but will make the number of vertices in the ternarized graph asymptotically equal to the number of edges. The algorithm then runs a local procedure from every vertex which discovers a subset of edges in the minimum spanning forest. We show that after contracting the graph based on this discovered fragment, the remaining graph has a factor of $n^{\epsilon/2}$ fewer vertices. At this point we can afford to invoke the dense routine from Behnezhad *et al.* on the contracted graph which solves the problem in $O((1/\epsilon) \log(1/\epsilon))$ rounds.

The local procedure run at each vertex is to simply run Prim's algorithm, a classic MSF algorithm, from the vertex until either of the two following stopping conditions is met. Firstly, to ensure that a machine does not perform too many queries, the algorithm truncates the local search if it exceeds n^ϵ queries. Secondly, running n^ϵ queries per vertex results may result in performing $O(n^{1+\epsilon})$ queries in total, which would use superlinear total space on sparse graphs. To restrict the total space to nearly linear, we sample a random priority for every vertex, and terminate the search if a vertex v's Prim search visits a vertex u with priority less than v. We argue that the overall query cost is $O(m \log n)$ w.h.p. by relating the cost of each query to a certain problem on randomized search trees (treaps).

We use the following algorithms from [19] as a black-box.

PROPOSITION 3.1. *There is an* AMPC *algorithm, DENSEMSF, which computes the minimum spanning forest of an undirected graph in $O((1/\epsilon) \log \log_{T/n} n)$ rounds w.h.p. where the total space $T = \Omega(m + n)$.*

Algorithm 1. TRUNCATEDPRIM($G = (V, E)$), where $\Delta(G) \leq 3$

1: For any vertex v in the graph pick a rank $\pi(v)$ uniformly at random from $(0, 1)$, and let π be the permutation obtained by sorting the vertices based on their ranks.
2: Each vertex v is assigned to a machine μ_v chosen uniformly at random from m/n^ϵ machines.
3: **for** each machine μ in parallel **do**
4:    $F_\mu = \emptyset$
5:    **for** each v s.t. $\mu_v = \mu$ **do**
6:      Run Prim's algorithm at v, stopping either when
7:        (1) v has explored $n^{\epsilon/2}$ vertices,
8:        (2) v's component is fully explored, or
9:        (3) v adds an edge to a vertex u s.t. $\pi(u) < \pi(v)$.
10:      $E_v \leftarrow$ MST edges discovered by v.   ▷ emitted as part of the output
11:      **if** v stops due to case (3) **then**
12:        $F_\mu \leftarrow F_\mu \cup \{(v, u)\}$
13: Apply the algorithm from Proposition 3.2 to $F = \cup_\mu F_\mu$, which contracts directed trees in F to their roots, and let $C : V \to V$ be a mapping representing the contraction.
14: Let $G'(V', E')$ be the graph obtained by contracting G according to C, with isolated vertices removed.
15: **return** $(\cup_{v \in V} E_v, G')$

PROPOSITION 3.2. *There exists an* AMPC *algorithm,* FORESTCONNECTIVITY, *that solves the forest connectivity problem in* $O(1/\epsilon)$ *rounds of computation w.h.p. using* $T = O(n \log n)$ *total space w.h.p.*

Note that this algorithm bears a close resemblance to the forest connectivity algorithm of [19]. One important difference, however, is that the forest connectivity algorithm of [19] applies a routine which shrinks the number of vertices in G by a factor of n^ϵ iteratively for $O(1/\epsilon)$ rounds. The idea is to shrink the graph until the maximum size of a tree is $O(n^{\epsilon/2})$. Unfortunately, a similar idea does not work here, since although we show that the number of vertices in G decreases by a factor of n^ϵ after one application of Algorithm 1, after contraction the graph is no longer ternarized. Importantly, the number of edges in the contracted graph may be asymptotically equivalent to the number of edges in the original graph, which would result in the algorithm making no progress. Instead, we use the fact that one application of Algorithm 1 shrinks the number of vertices by a factor of n^ϵ, which is sufficient to apply the algorithm from Proposition 3.1 to the contracted graph.

LEMMA 3.3. *The contracted graph produced by Algorithm 1 has a factor of* $\Omega(n^{\epsilon/2})$ *fewer vertices than* G.

At a high level, to prove the lemma, we view F_μ as a collection of trees. Then, we show that each vertex is a root of one tree with probability $O(n^{-\epsilon/2})$ (thanks to first stopping condition of Prim's algorithm in Algorithm 1). Hence, the number of trees in F_μ and consequently the number of vertices in the contracted graph shrinks by a factor of $\Omega(n^{\epsilon/2})$.

Next, we use the condition from Line 9 of Algorithm 1 to bound the total communication of the algorithm.

LEMMA 3.4. *Algorithm 1 uses* $O(n \log n)$ *queries w.h.p.*

Algorithm 2. MSF($G = (V, E)$)

1: **if** $m < n^{1+\epsilon/2}$ **then**
2:    Let $G'(V', E')$ be a degree bounded version of G, obtained by replacing every vertex v with degree > 3 with a cycle of length $deg(v)$, connecting each edge of v to its corresponding vertex in the cycle. Let the weights of the dummy edges be denoted by $\perp$, chosen to be less than the weight of the lightest edge in E.
3:    $(G''(V'', E''), E_T) \leftarrow$ TRUNCATEDPRIM(G'). Note that $|V''| = O(m^{1-\epsilon/2})$.
4:    $E'_T \leftarrow$ edges obtained from applying the algorithm from Proposition 3.1 to G''.
5:    **return** $E_T \cup E'_T$, with all edges with weight $\perp$ removed.
6: **return** edges obtained from applying the algorithm from Proposition 3.1 to G

The following lemma argues that the vertices can be assigned to machines in such a way that every machine performs $O(n^\epsilon)$ queries per round w.h.p. The space-complexity bound follows directly from our bound on the number of queries. Since the algorithm performs a constant number of steps, each implementable in $O(1/\epsilon)$ rounds of AMPC, the round-complexity follows.

LEMMA 3.5. *Algorithm 1 runs in* $O(1/\epsilon)$ *rounds and* $O(n \log n)$ *space w.h.p.*

By putting together the above lemmas, we obtain:

LEMMA 3.6. *Algorithm 2 computes the minimum spanning forest of an undirected graph in* $O(1/\epsilon \log(1/\epsilon))$ *rounds w.h.p. using total space* $T = O(m \log n)$ *w.h.p..*

To prove Theorem 1 we must improve the query complexity of the algorithm from $O(m \log n)$ to $O(m + n \log^2 n)$. To that end, we use a sampling scheme by Karger *et al..* [43] used in the single-machine linear-time algorithm for computing minimum spanning tree. By using basic algorithmic techniques on trees, i.e. finding lowest common ancestors and heavy-light decomposition, we show that the scheme can be implemented in a constant number of AMPC rounds.

4 Matching

Maximum matching and its natural extension maximum weight matching are among the most fundamental combinatorial optimization problems with a wide range of applications. Particularly, maximum weight matching is an important subroutine in balanced partitioning and hierarchical clustering, see e.g., [10] and the references therein.

In this section, we consider the matching problem in the AMPC model. Our main result is an efficient algorithm for the unweighted *maximal matching* problem. We then use this algorithm as a black-box to get an algorithm for approximate maximum weight matching and related problems.

THEOREM 2. *There is an* AMPC *algorithm which with probability* $1 - 1/\text{poly}(n)$ *computes a random greedy maximal matching using* $O(n^\delta)$ *space per machine (for any constant $\delta \in (0, 1)$) and:*

1. *In* $O(\log \log n)$ *rounds using* $\widetilde{O}(m + n)$ *total space.*
2. *In* $O(1)$ *rounds using* $O(m + n^{1+\epsilon})$ *total space for any constant* $\epsilon > 0$.

Algorithm 3. Algorithm for maximal matching.

1: For any edge e in the graph, pick a rank $\pi(e)$ uniformly at random from $(0, 1)$ and let π be the permutation obtained by sorting the edges based on their ranks.
2: $G_1 \leftarrow G$
3: **for** i in $1 \ldots k = \lceil \log_2 \log_2 \Delta \rceil + 1$ **do**
4:    **if** $\Delta(G_i) > 10 \log n$ **then**
5:      Let H_i be the subgraph of G_i containing its edge e iff $\pi(e) \in [0, \Delta^{-0.5^i}]$.
6:    **else**
7:      $H_i \leftarrow G_i$.
8:    Find matching $M_i = \mathsf{GreedyMM}(H_i, \pi)$ by running the MIS algorithm of Proposition 4.2 on the line graph of H_i, using permutation π.
9:    $G_{i+1} \leftarrow G_i[V \setminus V(M_i)]$.
10: **return** matching $M_1 \cup M_2 \cup \ldots \cup M_k$.

The *random greedy maximal matching* is obtained by considering the edges one by one in a random order and adding each edge to the matching if none of its endpoints has been previously matched. Theorem 2 leads to the following results.

COROLLARY 4.1. *The same guarantee as in Theorem 2 also applies to $1 + \epsilon$ approximate maximum matching, $2 + \epsilon$ approximate maximum weight matching, and 2 approximate minimum vertex cover.*

We use the following MIS algorithm of [19] as a black-box.

PROPOSITION 4.2. *[19] There is a randomized* AMPC *algorithm which computes a random-greedy maximal independent set in $O(1)$ rounds using $O(n^\delta)$ space per machine (for any constant $\delta \in (0, 1)$) and $\widetilde{O}(m)$ total space. The bound on the total space holds in expectation and the rest of the bounds hold with high probability.*

Our algorithm uses the following relation between maximal independent set and maximal matching. Given an undirected graph G, the *line graph* of G is obtained by having a vertex for each edge of G and connecting any two of these vertices that correspond to edges sharing an endpoint in G. It is well-known that the set of vertices in the maximal independent set of the line graph of a graph G forms a maximal matching of G. Therefore, having Proposition 4.2 which gives an efficient AMPC MIS algorithm, one may hope to be able to directly get an efficient algorithm for maximal matching as well. Unfortunately, this is not the case. The main hurdle is that the line graph can be significantly larger than the graph itself and thus the total space of an algorithm constructing the line graph may be super-linear in the number of edges of the original graph. To get around this, we take two different approaches leading to the two bounds obtained in Theorem 2.

For the first bound, our main idea is to run the MIS algorithm on the line graph of a smaller edge-sampled subgraph of the original graph G, commit the edges of the obtained (not maximal) matching to the output (which results in pruning this graph) and then recursively handle the residual graph. We show that $O(\log \log \Delta)$ iterations suffice to find a maximal matching, where Δ denotes the maximum degree of the input graph, while also ensuring that the total required space remains $\widetilde{O}(m)$.

For the second bound, first notice that the line-graph of a graph with m edges and maximum degree Δ may have up to $m\Delta$ edges. Hence, if we explicitly construct the line-graph and run the MIS algorithm as a black-box we may require $\Omega(m\Delta)$ total space. We note that a simple adaption of the idea used in the algorithm of Proposition 4.2 can be used to reduce the total space to $O(m^{1+\epsilon})$. The idea is to run the algorithm without explicitly constructing the line-graph a-priori, but rather generating it on the fly. To improve this further to $O(m + n^{1+\epsilon})$, we use the structure of line-graphs and use a more efficient query process for matchings as opposed to the one (due to [70]) used in Proposition 4.2 for maximal independent sets.

4.1 Theorem 2 (Part 1)

Notation. We need a few definitions to formalize the algorithm. Given a graph $G(V, E)$ and a subset V' of V, we use $G[V']$ to denote the induced subgraph of G on vertex set V'. Furthermore, given a matching M in G, we use $V(M)$ to denote the vertices matched in M. Also, given a permutation π over the edges in E, we denote by $\mathsf{GreedyMM}(G, \pi)$ the greedy maximal matching obtained by iterating over the edges in the order of π. Finally, $\Delta(G)$ denotes the maximum vertex degree in G. The algorithm is formalized as Algorithm 3. It is easy to verify that each iteration of the for loop can be implemented in $O(1)$ rounds of AMPC so long as the space per machine is $n^{\Omega(1)}$. Therefore, the number of rounds is indeed $O(\log \log \Delta)$. In the remainder of this section, we prove the space bounds and the algorithm's correctness. We need the following well-known degree-reduction property of lexicographically first maximal matching in our analysis. See for instance [23, Lemma A.1].

PROPOSITION 4.3. *Let G be any undirected graph, π be a random permutation on the edges of G, and p be a parameter in $(0, 1)$. The maximum degree in graph $G[V \setminus V(\mathsf{GreedyMM}(G_p, \pi))]$ is w.h.p. $5 \log n / p$ where G_p is the subgraph of G including only the p fraction of edges with the lowest rank in π.*

Using the above proposition, we obtain the following.

LEMMA 4.4. *For any i, the maximum degree in graph G_i is at most $5\Delta^{0.5^{i-1}} \log n$ w.h.p.*

LEMMA 4.5. *Algorithm 3 is correct and uses $\tilde{O}(m)$ space.*

PROOF. Since $G_{i+1} = G_i[V \setminus V(M_i)]$, all the matchings $M_1, \ldots, M_k$ will be vertex disjoint and thus their union is a valid matching of the graph. It remains to show that this is a maximal matching. Recall from Lemma 4.4 that the maximum degree in graph G_i is $5\Delta^{0.5^i} \log n$. Therefore, for $k' = \lceil \log_2 \log_2 \Delta \rceil$, we can bound the maximum degree in $G_{k'}$ by

$$5\Delta^{0.5^{\lceil \log_2 \log_2 \Delta \rceil}} \log n \leq 5\Delta^{\frac{1}{\log_2 \Delta}} \log n = 10 \log n.$$

This means that in the next iteration $k = k' + 1$, the condition of Line 4 does not hold and we have $H_k = G_k$ and thus the maximal matching M_k of H_k is also a maximal matching of G_k. Thus, $M_1 \cup \ldots \cup M_k$ is a maximal matching of G. $\square$

4.2 Theorem 2 (Part 2)

Due to space limits, we give only a proof sketch and provide the full proof in the full version of the paper [20].

Suppose that we are given a random permutation π over the edges, and are tasked to determine whether an edge e is matched in the corresponding random greedy matching M. The original query process of [70] is as follows for matchings: We iteratively pick the incident edge f to e with the lowest rank in π. If f happens to have a higher rank than e, that is, if no edge of lower rank than e is incident to e, then e must be in the matching. Otherwise, we recursively query f. If f happens to be part of the matching, e is not and we can terminate the process; if f is not in the matching, we are unsure about the status of e and proceed to the next incident edge to e.

A truncated variant of this process proposed in [19] follows the same recursive idea, but truncates it if the total number of recursive queries exceeds n^ϵ. A similar analysis as done in [19] for MIS which builds on that of [70] shows that if we run this truncated query process on all the edges, the number of edges whose query process is truncated is small enough that significant progress on the whole graph is made. Therefore, by repeating this process $O(1)$ times, we get a maximal matching. Particularly it is shown in [19] that:

LEMMA 4.6. *Suppose that we run the n^ϵ-truncated query process above, and remove all edges that are known to join the matching along with their incident edges. Then $O(1/\epsilon)$ applications of this process makes the graph empty.*

The problem with this approach is that since we conduct n^ϵ queries from each edge, the total number of queries (and equivalently the total communication/space) can be $\Theta(mn^\epsilon)$.

To reduce the total space to $O(m+n^{1+\epsilon})$, instead of edges, we start the query processes from the *vertices* and truncate them. The query process of a vertex v, basically iterates over the incident edges of v in the increasing order of their ranks and runs the edge query process. Once an incident edge in the matching is found, the vertex query process terminates.

The truncated variant of the vertex query process above is natural: once the number of recursive queries for the vertex exceeds n^ϵ, we stop and mark the vertex as unsettled. It is clear that since there are n vertices and each queries only n^ϵ portion of the graph, the total space needed is only $O(m + n^{1+\epsilon})$. In the full version [20], we show that this vertex-truncated process also makes enough progress on the graph that after $O(1)$ rounds, we find a maximal matching.

LEMMA 4.7. *Suppose that we run the n^ϵ-truncated query process above from the vertices. Then we remove all vertices known to be matched along with their incident edges. Then $O(1/\epsilon)$ applications of this process makes the graph empty.*

5 Empirical Evaluation

In this section we provide an empirical evaluation of the AMPC model using hundreds of threads on production machines in a large data center. We summarize the main experimental results described in this section below:

- A case-study of AMPC and MPC implementations of MIS (Section 5.3), including a discussion of different optimizations, round-complexity, communication and a detailed evaluation explaining their performance.
- AMPC implementations of Maximal Matching (Section 5.4) and Minimum Spanning Forest (Section 5.5), detailed experimental comparisons explaining their performance, and a comparison to the state-of-the-art MPC algorithms that we implemented in this paper.

- Experimental evaluation of the 1-vs-2 Cycle problem, comparing AMPC vs MPC implementations on synthetic cycle graphs (Section 5.6).
- A discussion studying potential scalability bottlenecks in our algorithms, as well as the potential for applying our approach to a broader set of problems (Section 5.7).

5.1 Environment & Implementation

We first provide context and rationale for the implementations and environment utilized in our evaluation.

Distributed Environment. In this paper we focus on a *fault-tolerant distributed environment* which reflects the challenges of running large-scale graph computations in shared production data center. The setting used in our experiments has recently been described in Tirmazi *et al.* [67]. In this setting, batch jobs are typically run at low priorities (i.e., using resources that are currently not used by high priority jobs), which makes them susceptible to preemptions. While running batch jobs at higher priorities is possible, it is also much more costly [1, 2] and thus often not done in practice. This is why systems like MapReduce, Hadoop or Flume-C++ [4] have strong fault tolerance properties and write the results of each computation round to durable storage.

The focus of our empirical evaluation is to study the performance of AMPC algorithms, and compare them with state-of-the-art MPC algorithms using a distributed computation framework with good fault-tolerance properties.

Implementing AMPC and MPC Algorithms. We implement both the AMPC and MPC algorithms studied in this paper in C++ using Flume-C++ [4, 27]. Flume-C++ is a highly-optimized fault-tolerant parallel data processing framework, whose API is similar to the open source system, Apache Beam. A Flume-C++ implementation consists of *stages*, that consume inputs generated by previous stages, and emit outputs that can be consumed by later stages. Flume automatically handles performing fusion and inter-stage optimizations. Compared to other popular data processing frameworks, such as Spark [72] and Timely Dataflow [55], the main difference between Flume-C++ and these systems is that the Flume-C++ runtime ensures that workers write the outputs of stages in the computation to durable storage for the duration of the job. This logging enables fault-tolerance, which is critical in the shared data center we run our jobs in.

To implement the AMPC algorithms, we use a custom distributed key-value store optimized for lookups and high throughput. The implementation used in our experiments takes advantage of hardware support for Remote Direct Memory Access (RDMA), which is a widely available technology [11, 12, 29]. We also experiment with an alternate implementation of the key-value store which only uses TCP/IP, but unless explicitly specified, our experiments use the RDMA-based implementation. The Flume-C++ implementations in this paper amount to a few hundred lines of C++ each. We emphasize that the *only difference* between our AMPC and MPC implementations is that the AMPC codes can query a key-value store within a Flume stage.

Machine Configuration. Our experiments are performed on machines from a production datacenter. Each machine contains two 2-way hyper-threaded 2.3GHz Intel 18-core CPUs, for a total of 72 hyper-threads per machine. Each machine is equipped with 262GB of RAM, and a 20Gbps NIC. We use a maximum of 100 machines to solve all problems, although we note that in practice we use far fewer than

Table 2: Graph inputs, including vertices and edges, diameter, the number of components, and the size of the largest component. We mark diameter values where we are unable to calculate the exact diameter with * and report a lower bound on the diameter observed in prior experimental work [32].

Dataset	n	m	Diam.	Num. CC	Largest CC
$2 \times k$	$2 \times k$	$2 \times k$	k	2	k
OK	3.07M	234.4M	9	1	3.1M
TW	41.6M	2.4B	23*	2	41.6M
FS	65.6M	3.6B	32	1	65.6M
CW	0.978B	74.7B	132*	23,794,336	0.950M
HL	3.56B	225.8B	331*	144,628,744	3.35B

72×100 hyper-threads (we request 400 hyper-threads but the experiments may use slightly more depending on cluster availability). Our experiments are run on a shared cluster, and may thus compete for resources with other jobs, and run virtually alongside other jobs on the same machine. To mitigate these effects, we ran all of our experiments using relatively high priorities. Thus, although we run our jobs in a fault-tolerant environment, our jobs typically do not experience failures. As a further precaution, we also run our experiments 3 times and report the median running time. We note that the difference in running times across different trials was not significant (within 10%).

5.2 Graph Data

We evaluate our algorithms on a representative set of real-world graphs of varying sizes. Several of our datasets are from the SNAP network suite: ***com-Orkut (OK)*** is an undirected graph of the Orkut social network, and ***Friendster (FS)*** is an undirected graph of the Friendster social network. ***Twitter (TW)*** is a graph of the Twitter network, where edges represent the follower relationship [47]. The ***ClueWeb (CW)*** graph is a Web graph from the Lemur project at CMU and Google, obtained from the LAW collection of datasets [26]. ***Hyperlink2012 (HL)*** is a directed hyperlink graph obtained from the WebDataCommons dataset where nodes represent web pages [53]. The Twitter, ClueWeb, and Hyperlink2012 graphs are originally directed graphs, so we symmetrize them before applying our algorithms. We test our 2-cycle algorithms on a family of massive high-diameter graphs consisting of two cycles on k vertices each ($2 \times k$ graphs) Finally, we test our minimum spanning forest algorithm on the same graph inputs where the weight of an edge (u, v) is proportional to $\deg(u) + \deg(v)$.

5.3 Case Study: Maximal Independent Set

To provide insights into how our implementations are programmed, in this subsection we give a detailed description of our Flume-C++ AMPC and MPC implementations of Maximal Independent Set (MIS) algorithms.

AMPC Algorithm. Figure 1 provides high-level pseudocode for the AMPC MIS algorithm studied in this paper. In our pseudocodes, we use several concepts from [27] (also used in Apache Beam), which we now describe. A **PCollection** is a potentially distributed, multi-element data set. A **KV<T, S>** is a key-value pair, whose key has type T and value has type S. A **DoFn<T, S>** is an operation that transforms a PCollection<T> to a PCollection<S>. The graph input is a PCollection of KV mapping NodeIds to Nodes. A **Node** is a list of NodeIds of the neighbors. Thus, every key-value pair in the graph represents a single vertex.

Our implementation is based on the $O(1)$ round AMPC algorithm described by Behnezhad *et al.* [19]. The algorithm finds the *lexicographically first MIS* over a random ordering of

```
1   // Uses hashing to determine a priority for each node.
2   uint64 NodePriority(NodeId node_id);
3
4   class IsInMIS : DoFn<KV<NodeId, Node>, NodeId> {
5     IsInMIS(const string& kv_store_id) {
6       kv_store = KVStore(kv_store_id); }
7     void Do(const KV<NodeId, Node>& node,
8             const EmitFn<NodeId>& emit) {
9       if (InMIS(node.key, node.val)){ emit(node.key); }}
10    // Check if any directed neighbors are in MIS. If
11    // none are in, this node is in.
12    bool InMIS(NodeId node_id, const Node& node) {
13      for (NodeId neighbor_id : node.neighbors()) {
14        // Fetch the neighbor's neighbors.
15        Node neighbor = Lookup(neighbor_id);
16        // Recursively query.
17        if (InMIS(neighbor_id, neighbor)) return false;
18      }
19      return true;
20    }
21    // Synchronously query the key-value store for the
22    // given ID, and return deserialized node.
23    Node Lookup(NodeId id);
24    KVStore* kv_store;
25  };
26
27  PCollection<NodeId> MIS(
28          const PCollection<KV<NodeId, Node>>& graph) {
29    // (1) Sort vertex neighborhoods based on priority.
30    // Only preserve edges to higher-priority neighbors.
31    PCollection<KV<NodeId, Node>> directed_graph =
32        DirectEdgesUsingPriority(graph, NodePriority);
33    // (2) Write directed graph to the key-value store.
34    auto kv_store_id =
35        WriteToKVStore(directed_graph);
36    // (3) Apply the IsInMIS DoFn over nodes.
37    PCollection<NodeId> ind_set = directed_graph.ParDo(
38        IsInMIS(kv_store_id));
39    return ind_set;
40  }
```

Figure 1: Pseudocode for our AMPC MIS algorithm.

the vertices. For convenience, we assume that each vertex is given a random priority defining its rank in the permutation.

Behnezhad *et al.* [19] show that the recursive algorithm of Yoshida *et al.* [70] can be adapted to run in $O(1)$ AMPC rounds. The algorithm of Yoshida *et al.* is based on the following recursive idea: each vertex belongs to the MIS if and only if none of its lower priority neighbors belongs to the MIS. Somewhat surprisingly, as shown in [70], the natural way of turning this property into a recursive function gives an algorithm that runs in linear total time ($O(m)$), even if the recursion is run separately from each vertex of the graph and with no memoization.

The AMPC algorithm of [19] runs this recursion in $O(1/\epsilon)$ steps. After step i all vertices that needed to make $O(n^{i \cdot \epsilon})$ recursive calls learned whether they belong to the MIS. This multi-stage approach turned out not to be needed in practice, and our AMPC implementation only needs 2 rounds of computation to find the MIS. See Figure 1 for the pseudocode.

MPC Algorithm. Our MPC algorithm is a recent $O(\log n)$-round *rootset-based* algorithm for computing the lexicographically-first MIS with respect to a random permutation on the vertices [24]. This algorithm was recently shown to have a $O(\log n)$-round complexity by Fischer and Noever [35]. By specifying the same source of randomness, both the MPC and AMPC algorithms compute the same MIS.

Conceptually, the rootset-based algorithm begins by drawing a random number for each vertex in the graph. Then, it

```
1   // Uses hashing to determine a priority for each node.
2   uint64 NodePriority(NodeId node_id);
3
4   PCollection<NodeId> MIS(
5       PCollection<KV<NodeId, Node>> graph) {
6     // Repeatedly extract a rootset
7     vector<PCollection<NodeId>> independent_set;
8     while (graph.numNodes() > 0) {
9       // (1) Find all nodes that have priority lower
10      // than their all neighbors. Since each node knows
11      // its neighbors, and the priorities are computed
12      // using hashing, this does not require a shuffle.
13      PCollection<KV<<NodeId, Node>> new_set =
14        LocalMinima(graph, NodePriority);
15      independent_set.push_back(new_set.Keys());
16      // (2) Compute node ids that the nodes in new_set
17      // and their neighbors (no shuffle).
18      PCollection<NodeId> to_remove =
19          IdsOfNodesAndNeighbors(new_set);
20      // (3) Mark which nodes should be removed. This
21      // requires joining graph with node ids in
22      // to_remove (1 shuffle).
23      PCollection<KV<NodeId, pair<Node, bool>>>
24          marked_graph =
25              MarkNodesToRemove(graph, to_remove);
26      // (4) Each marked node x emits <x, y> and
27      // <y, x> for each neighbor y. This computes all
28      // edges to be deleted (no shuffle).
29      PCollection<KV<NodeId, NodeId>> edges_to_delete =
30          FindDeletedEdges(marked_graph);
31      // (5) Update the graph by removing marked nodes
32      // and their incident edges. Requires joining the
33      // graph with edges_to_remove (1 shuffle).
34      graph = RemoveNodesAndEdges(
35          marked_graph, edges_to_remove);
36    }
37    // Flatten vector<PCollection<NodeId>> to
38    // a PCollection<NodeId>
39    return Flatten(independent_set);
40  }
```

**Figure 2:** Pseudocode for the rootset-based MPC MIS algorithm.

proceeds in phases. In each phase, it finds all vertices whose priority is lower than the priority of all their neighbors. All such vertices are added to the MIS, and after that they are removed from the graph together with their neighbors. After $O(\log n)$ phases, all vertices are removed from the graph [35]. The pseudocode of our MPC implementation is given in Figure 2. We note that although a $O(\sqrt{\log n})$-round MPC algorithm is also known [38], the complexity of the algorithm makes it likely impractical. Finally, we experimentally determined that switching to an in-memory algorithm once the number of edges in the graphs decreases below 5×10^7 achieves a good tradeoff between the cost of processing the graph on a single machine, and the cost of a new phase.

We note that we also considered an MPC implementation of the AMPC algorithm as a potential baseline, in which each step of querying the key-value store was mapped to a shuffle. We observed that this algorithm requires over 1000 shuffles even for the Orkut and Friendster graphs, and is over 50x slower than the rootset-based algorithm, and thus we chose to use the rootset-based algorithm as our MPC baseline.

Round-Complexity and Communication. What is the empirical benefit of reducing the round-complexity? For one, the implementations of our AMPC algorithms perform fewer *shuffles*. A **shuffle** is a phase which takes a set of emitted key-value pairs and groups pairs with the same key on the same machine [27] and is the only way a Flume-C++ worker can exchange large amounts of data with other workers. At

Table 3: Number of shuffles (costly rounds) used by our AMPC and MPC implementations for real-world graph datasets.

Algorithm	OK	TW	FS	CW	HL
AMPC Maximal Independent Set	1	1	1	1	1
AMPC Maximal Matching	1	1	1	1	1
AMPC Minimum Spanning Forest	5	5	5	5	5
MPC Maximal Independent Set	8	10	10	12	14
MPC Maximal Matching	8	12	12	14	16
MPC Minimum Spanning Forest	33	54	57	84	–
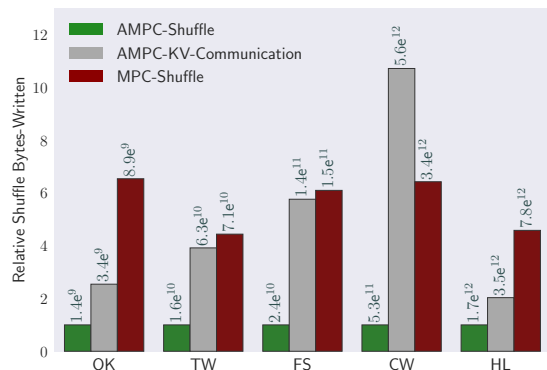
Figure 3: Normalized bytes shuffled for the AMPC and MPC MIS algorithms, and the normalized bytes of communication with the key-value store for the AMPC MIS algorithm. Each bar is annotated with the actual number of bytes on top of the bar.

a high level, a single Flume-C++ shuffle corresponds to a single round in the MPC and AMPC models. We point out that empirically, most of the computation time in the MPC algorithms studied in this paper is spent on shuffles. In Table 3 we report the number of shuffles (or MPC/AMPC rounds) used by our AMPC MIS algorithm and our MPC MIS algorithm. The AMPC algorithm uses a single shuffle to construct a directed graph (step (1) in Figure 1). The MPC algorithm requires two shuffles per phase of the algorithm, and takes between 8–14 shuffles in total.

Although the graph size shrinks between each phase, each shuffle still transmits a significant amount of data. To illustrate the impact of performing fewer shuffles in the AMPC algorithm, in Figure 3 we plot the total number bytes written during all shuffles in the algorithm for our AMPC and MPC algorithm, as well as the total amount of communication performed by the AMPC algorithm to the key-value store. In all cases, the AMPC algorithm shuffles significantly fewer bytes, since the single shuffle it performs writes bytes only proportional to the input graph size. Note that the total communication to the key-value store in the AMPC algorithm is typically less than the total bytes shuffled by the MPC algorithm, with the exception of the ClueWeb graph. Despite the fact that the AMPC algorithm sometimes uses more total bytes of communication, it is always faster as we discuss later, since the key-value store communication is done over a relatively high-throughput network.

Optimizations in the AMPC Algorithm. Next, we consider two optimizations that can be applied to the AMPC algorithm to improve its performance.

The first is a ***multithreading*** optimization that is broadly applicable to any AMPC algorithm that performs lookups to a key-value store. Since in our programming model, these lookups are performed synchronously, we use multiple threads to enable each worker to concurrently process many instances
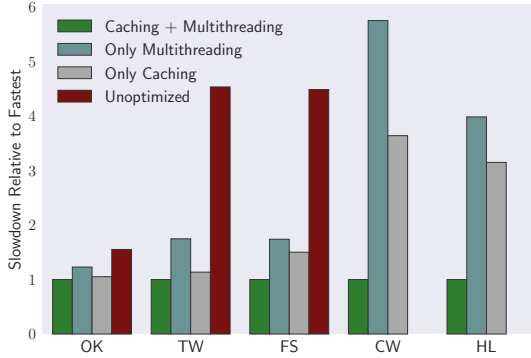
Figure 4: Effect of the caching and multithreading optimizations for our AMPC Maximal Independent Set implementation. The AMPC MIS algorithm without optimizations did not finish within 4 hours for both the CW and HL graphs.

Figure 5: Normalized running times for AMPC and MPC Maximal Independent Set implementations. Each bar is annotated with the parallel running time on top of the bar.

of a DoFn. This optimization enables a thread processing a task that is waiting to receive a result from the key-value store to be swapped out for another thread. Since RDMA lookups to the key-value store are in general an order of magnitude slower than lookups to DRAM, enabling this optimization should improve the running time of the part of the algorithm that performs lookups to the key-value store.

The second is a ***caching*** optimization that we apply to the MIS and Maximal Matching algorithms in this paper. The idea is essentially to save the result of a query determining whether a given vertex (or edge) is preserved in the MIS. Theoretically, since each machine only performs $O(n^\epsilon)$ communication, the results of all queries it recursively answers can be saved on this machine in the model. In practice, we implement the caching optimization using an array indexed over the vertices that is shared between all threads operating on a machine. In our MIS algorithm, this table stores a three-valued state reporting whether the status of this vertex is either *Unknown*, *InMIS* or *NotInMIS*.

Figure 4 shows the impact of the caching and multithreading optimizations for our AMPC MIS implementation on the overall running time. We note that only the running time of Step (3) in our algorithm in Figure 1 is affected by these optimizations (we tried enabling multithreading for the other steps, but the results were always slower). Caching decreases the running time of this step by reducing the number of bytes that must be communicated with the key-value store. We observe that both optimizations always provide speedups over a baseline, and that the fastest times are obtained when both optimizations are applied. Only using multithreading enables a 1.26–2.59x speedup over the unoptimized algorithm, and only using caching enables a 1.47–3.99x speedup over the unoptimized algorithm. Enabling caching decreases the number of bytes transmitted to the key-value store by between 1.96–12.2x. In what follows, when we refer to our AMPC MIS algorithm we mean the variant that uses both caching and multithreading unless mentioned otherwise.

Running Time and Speedup. Figure 5 plots the normalized running times of the AMPC and MPC MIS algorithms, and breaks down the running time of the AMPC algorithm into the three steps indicated in Figure 1. For smaller graphs, the time spend shuffling to construct the directed graph is between 2.06–3.24x more than the time spent in the IsInMIS search procedure. For larger graphs, the cost of the search procedure is between 1.38–1.43x more costly than building
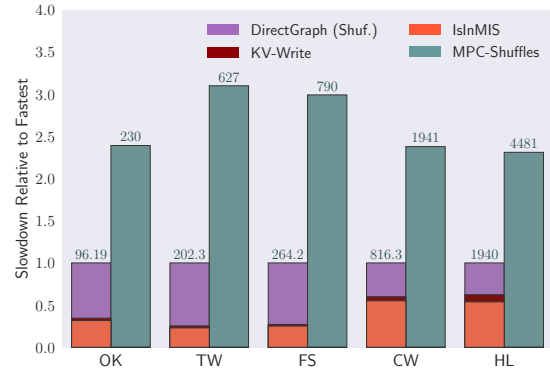
the directed graph due to a greater volume of queries (see Section 5.7). Writing the directed graph to the key-value store takes a small fraction of the time (at most 8%).

We observed that the optimized AMPC algorithm is always faster than the MPC algorithm, and achieves between 2.31–3.18x speedup over the MPC algorithm. The largest speedup is for the ClueWeb graph. The slow performance of the MPC algorithm on this graph is due to skew in the join since the ClueWeb graph has many high degree vertices with degrees larger than 10 million, and as large as 75.6 million. Our speedups over the MPC algorithm can be attributed to both performing fewer shuffles, and shuffling fewer bytes of data overall (see Figure 3). We also obtain this speedup due to performing a modest amount of total communication to the key-value store, which we discuss more in Section 5.7.

5.4 Maximal Matching

AMPC Implementation and Optimizations Similarly to MIS, we implement algorithms finding the lexicographically first matching for a random ordering of edges. Our AMPC implementation is the constant round algorithm that we designed in Section 4, which we refer to as ***AMPC-MaximalMatching***. The algorithm is implemented similarly to the AMPC MIS algorithm shown in Figure 1.

The main differences are that (i) the graph stored in the key-value store does not direct the edges, but instead sorts the edges based on random priorities assigned to each edge and (ii) instead of applying an query process independently on each edge, we iteratively query edges incident to each vertex u in order of increasing priority. Finally, instead of caching results *per-edge*, we observed that it suffices to maintain a value *per-vertex*, since if an edge incident to vertex u with priority P is not in the matching, this indicates that all edges incident to u with priority less than P are also not in the matching. Specifically, the cache stores for each vertex a NodeId, and an enum indicating whether this id is the highest priority neighbor that is finished, the matched neighbor, or whether this vertex has not been searched yet.

MPC Implementation. We implement a rootset-based maximal matching algorithm, which is very similar to our MIS algorithm in the MPC setting. Similarly to MIS, in each round, this algorithm adds to the matching all edges whose priority is smaller than the priority of all its adjacent edges and removes matched edges together with their endpoints. We implement this algorithm in Flume C++ almost identically to the MIS algorithm shown in Figure 2. Once the graph contains at most s edges, where s is a tunable
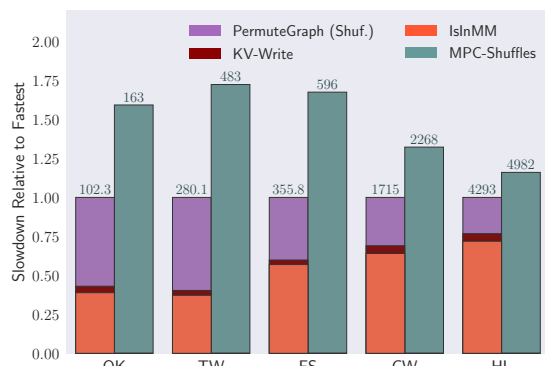
Figure 6: Normalized running times for AMPC and MPC Maximal Matching implementations. Each bar is annotated with the parallel running time on top of the bar.

Figure 7: Breakdown of running time for AMPC and MPC implementations of Minimum Spanning Forest.

parameter, it is sent to a single machine, which finds the remaining edges of the matching. In our experiments, we experimentally determined that setting $s = 5 \times 10^7$ gives a good tradeoff between the overhead of spawning a new round of the algorithm, and the time taken to process the graph on a single machine. Similarly to the MIS algorithm, this algorithm takes $O(\log n)$ rounds with high probability.

Round-Complexity. In Table 3 we report the number of shuffles used by our AMPC MM algorithm and our MPC MM algorithms. Like our AMPC MIS algorithm, our AMPC MM algorithm only performs a single shuffle to construct the edge-permuted graph. The remaining round is a cheap map step over the original graph. Our MPC MM algorithm requires two shuffles per phase of the algorithm. We observe that the algorithm performs a similar number of phases (and thus shuffles) compared to the MPC MIS algorithm.

Optimizations. Next, we considered the effect of optimizations on our AMPC MM algorithm. We found that multithreading was always beneficial, and did not evaluate its effect in more detail. We were able to evaluate the algorithm without using caching on OK, TW, and FS, but the algorithm without caching took longer than 4 hours on CW and HL. For these graphs, enabling caching reduced the total number of bytes read from the key-value store by between 2.65–8.81x and improved the running times by between 1.42–1.95x over the algorithm using just multithreading.

Running Time. Figure 6 reports the results of our comparison on all of our real-world graph inputs, and reports a breakdown of the different phases of our AMPC MM algorithm. The breakdown is grouped into the three steps of the algorithm—building the edge-sorted graph, writing it to the key-value store, and running searches using the IsInMM procedure—similar to the breakdown for our MIS algorithm in Section 5.3. We observe that copying the graph takes somewhat longer than the MIS algorithm, which is due to the fact that the all edges are present in this graph. The breakdown otherwise exhibits a similar trend to the MIS ones. Compared to the MPC algorithms, the AMPC algorithm is always faster, and achieves between 1.16–1.72x speedup over the MPC algorithm. The main reason for the lower speedup compared to that of the MIS algorithm is the larger cost of the search procedure, and the larger amount of data shuffled to build the graph written to the key-value store. The cost of the IsInMM procedure increases with the
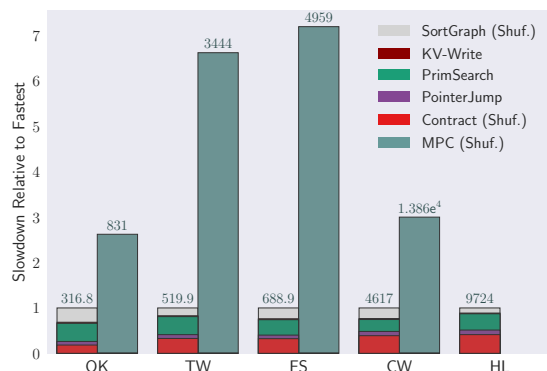
graph size, which is due to the increased number of queries made to the key-value store (discussed more in Section 5.7).

5.5 Minimum Spanning Forest

AMPC Implementation. Our AMPC algorithm is based on the constant round algorithm that we designed in Section 3. We empirically found that implementing a single search procedure on the graph without ternarization is sufficient to shrink it to a very small size. Conceptually our implementation has three main parts:

The first part sorts the edges incident to each vertex by their weights. This graph is then written to the key-value store. The algorithm then applies Prim's algorithm at each vertex v using the key-value store to fetch newly visited vertices. The search runs until either the vertex hits a threshold in terms of the total amount of edges it examines, or if it visits a vertex with higher priority. Each search from a vertex u emits a tuple containing itself and the NODEID of every lower priority vertex it visits, which is used to contract the graph in the next step. It also emits all of the MSF edges that it observes during the search.

The second part groups the tuples from the previous part by the visited vertex u, and combines them to select the visitor with the *highest priority* among all visitors to u. The algorithm then writes this map from NODEIDs to visitor NODEIDs to the key-value store, and applies *pointer jumping* to contract the directed trees induced by the visited relationships. The final output of this part is a contraction mapping sending NODEIDs to NODEIDs (tree roots).

The final part contracts the graph based on the mapping obtained by the previous step, and applies an in-memory MSF algorithm on the contracted graph. The graph-contraction step is implemented using two shuffles in Flume.

MPC Implementation. We implement the classic Borůvka's algorithm which finds a minimum spanning forest in $O(\log n)$ rounds in the MPC model. In each phase of the algorithm, every vertex randomly colors itself either *red* or *blue*. Each blue vertex computes the minimum weight edge incident to it, and if this neighbor is red, then the vertex contracts to the neighbor, otherwise the vertex does not contract. The contraction routine is the same one used in our AMPC algorithm above. The algorithm iterates these phases until the number of edges in the graph goes below 5×10^7, at which point it applies an in-memory MSF algorithm.

Round-Complexity. Table 3 reports the number of shuffles used by the AMPC and MPC MSF algorithms. The AMPC algorithm requires 5 shuffles, 1 for each of {graph construction,

combining on visited vertices, pointer-jump construction}
and three shuffles to contract the graph. The MPC algorithm
performs three shuffles per phase of the algorithm (each
phase contracts the graph); empirically, Borůvka's algorithm
takes between 11–28 phases. The number of phases is much
higher than in the MPC MIS or MM algorithms since each
phase of Borůvka only shrinks the number of vertices (not
the number of edges) in the graph by a constant factor in
expectation, and thus many phases are required.

Running Time. Figure 7 shows the normalized running
times for the AMPC and MPC MSF algorithms. We note
that we were unable to obtain a result for the MPC algorithm
within 4 hours for the HL graph. We breakdown the running
time of our AMPC algorithm into five phases, namely the
time to: sort the graph (SortGraph); perform writes to
the key-value store (KV-Write); perform Prim's searches
(PrimSearch); pointer-jump (PointerJump); and lastly the
MPC routine used to contract the graph (Contract).

We observe that unlike in our MIS and MM algorithms,
the largest fraction of the time is spent on graph contraction
(on the HL graph, where the search time and contraction
time are the closest, 15% more time is spent contracting).
Performing pointer-jumping consistently takes about 10%
of the overall time. Our implementation of pointer-jumping
simply repeatedly queries the parent of a vertex until it hits
a tree root. Although the worst-case depth of this algorithm
could be as much as $O(n)$, in practice, the trees constructed
by the algorithm are very shallow (we observed a maximum
query length of 33 over all graphs).

Compared to the MPC algorithm, our AMPC algorithm
is always faster, and achieves between 2.6–7.19x speedup.
The running times for both of our implementations of this
problem are significantly slower than for the other problems
studied in this paper, primarily because of the costly graph
contraction procedure used in both algorithms. We note
that we tried to optimize this graph contraction step using
lookups to a key-value store, but were unable to obtain any
significant speedup over the MPC routine.

5.6 1-vs-2-Cycle

AMPC and MPC Implementation The 1-vs-2-Cycle prob-
lem is to distinguish whether an input graph consists of a
single cycle on n vertices, or 2 cycles on $n/2$ vertices each.
This is a canonical problem widely believed to be hard in
the MPC model, i.e., the 1-vs-2-Cycle conjecture states that
solving it requires $\Omega(\log n)$ rounds in the MPC model. At the
same time, the problem admits a very simple algorithm that
requires only $O(1)$ rounds in the AMPC model [19]. While
the problem is of purely theoretical interest, comparing MPC
and AMPC algorithms for the problem reveals the potential
speedups from using the AMPC model.

The $O(1)$ round AMPC algorithm for this problem is based
on sampling vertices with probability $O(n^{-\epsilon/2})$ and searching
outward from each vertex until another sampled vertex is
hit. Then, the graph is contracted to a graph on the sampled
vertices, with edges between adjacent samples that discover
each other. We refer the reader to Behnezhad *et al.* [19] for
the full details of the algorithm, and analysis.

We implemented this sampling-based algorithm in Flume
C++ and refer to it as AMPC-*1-vs-2-Cycle*, and evaluated
its performance on a family of $2 \times k$-cycle graphs. Our imple-
mentation performs a single round of the search procedure,
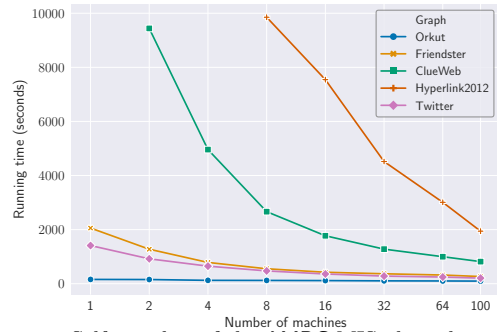
Figure 8: Self-speedup of the AMPC MIS algorithm when run
on between 1–100 machines.
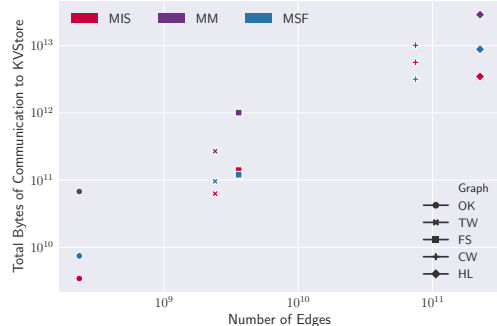
Figure 9: Total bytes of communication to the key-value store
by AMPC algorithms.

sampling vertices with probability 1/1024, and solves the sub-
sequent contracted graph on a single machine. We compare
with a general-purpose MPC connectivity algorithm (*CC-
LocalContraction*) based on performing local-contractions,
which prior work found to be the fastest MPC connectivity
implementation across a wide range of graphs [48].

Results On average, AMPC-1-vs-2-Cycle achieves between
3.40–9.87x speedup over CC-LocalContraction, with the
speedups increasing as the number of vertices increases. The
main savings (and improvement in the running time) comes
from fewer shuffles, and fewer bytes shuffled. The AMPC
algorithm requires a single shuffle used to write the graph to
the key-value store. The MPC algorithm reduces the length
of the cycle by roughly a factor of 2.59–3x in each iteration
of the algorithm (2.69x on average). Each iteration contracts
the graph, which requires 3 shuffles. The MPC algorithm
uses 4–9 iterations across all cycle inputs (12–27 shuffles).

5.7 Discussion

Scaling. We evaluated the scaling (or self-speedup) or our
AMPC MIS algorithm to check that our AMPC algorithms
obtain speedups when varying the number of machines. We
were unable to obtain 1-machine times for the ClueWeb graph
and 1, 2, or 4-machine times for the Hyperlink2012 graph
within 4 hours. For the smaller graphs, the 100-machine time
is between 1.64–7.76x faster than the 1-machine time. The
speedups are better for larger graphs, since there is more
work to do relative to the overhead of spawning rounds and
shuffles. For ClueWeb, the 100-machine time obtains a 11.5x
speedup over the 2-machine time, and for Hyperlink2012 the
100-machine obtains a 5x speedup over the 8-machine time.
We believe that one of the reasons we do not obtain linear
speedup may be due to saturating the network bandwidth
when querying the key-value store, which we discuss next.

Table 4: Normalized running times of 1-vs-2-Cycle (2-Cyc.) and MIS algorithms. The AMPC algorithms are the first two rows in each group and communicate with the key-value store using either RDMA or TCP/IP. The MPC implementation is the last row in each group. $2e^k$ refer to the 2×10^k graphs.

Algorithm	OK	TW	FS	CW	HL	$2e^8$	$2e^9$	$2e^{10}$
2-Cyc. (RDMA)	–	–	–	–	–	1	1	1
2-Cyc. (TCP/IP)	–	–	–	–	–	1.74	3.75	5.90
MPC 2-Cyc.	–	–	–	–	–	3.40	6.70	9.87
MIS (RDMA)	1	1	1	1	1	–	–	–
MIS (TCP/IP)	1.85	1.63	1.50	1.68	1.71	–	–	–
MPC MIS	2.39	3.04	2.98	2.37	2.30	–	–	–

Key-Value Store Communication. Figure 9 plots the the number of edges in our real-world inputs on the x-axis and the total bytes of data communicated to the key-value store on the y-axis. We observe that for all of the problems there is a consistent linear trend in terms of the total amount of communication with respect to the number of edges. Although the total number of bytes communicated is one upper bound on the total amount of memory used by our algorithms, we point out that in practice, the total memory usage is closer to the graph size (which is at most 2TB for the Hyperlink2012 graph). Finally, we observed that in all cases, the maximum throughput we obtain on our network in our experiments is about 80Gb/sec (which amounts to just under 1 Gb/sec per machine). It seems likely that the time spent on the search phase of each algorithm could be accelerated by exploiting high throughput networks, and this is an interesting direction to investigate in future work.

How much of the speedup over MPC algorithms in our experiments is due to using RDMA-based communication? To study this question we replaced all of the key-value store communication within the AMPC algorithms with RPCs sent over TCP/IP. Table 4 shows the results of this evaluation for the 1-vs-2-Cycle and MIS problems. We see that the impact of using RDMA is more significant for 1-vs-2-cycle, since the lower latency of RDMA enables searches around the cycle to terminate more quickly. The slowdown for the MIS problem is more modest, with the TCP/IP based algorithm only being 1.67x slower than the RDMA-based algorithm on average. These results suggest that although RDMA-based communication is important in achieving good running times, it can safely be replaced by RPCs sent over TCP/IP, and the resulting algorithms will still outperform fast MPC baselines.

AMPC vs MPC Algorithms. One of the main messages of our paper is that AMPC algorithms can deliver better performance than state-of-the-art MPC algorithms, without sacrificing fault-tolerance properties. There are a number of reasons for this performance improvement. First, AMPC algorithms use significantly fewer rounds compared to fast MPC baselines as shown in Table 3. Second, AMPC algorithms also shuffle significantly less data per round, which results in fewer disk writes (e.g., see Figure 3). Finally, using RDMA as a communication mechanism results in excellent performance as the algorithm can effectively replace the shuffle-based communication mechanism which uses costly disk writes with a fast, low-latency protocol. We see the AMPC approach as an interesting middle-ground between systems that communicate through persistent storage (like MapReduce, Hadoop and Flume-C++) and are thus robust to preemptions, and systems that run fully in memory, which deliver better performance at the cost of not tolerating preemptions well and requiring high priority (reserved) resources.

Applicability. We believe the AMPC model is a promising platform for problems which can be expressed using graph exploration, such as connected components, sub-structure problems, and problems involving random walks.

Connected Components One specific challenge stemming from this paper is to obtain a result for connectivity that achieves speedups over the state-of-the-art MPC algorithm [48]. We tried to apply our MSF algorithm over a graph with random edge weights, but were not able to obtain significant speedups over this MPC result due to the high cost of graph contraction on the first step (contracting the initial graph takes about 2/3 of the overall running time).

Sub-structure Extraction. The k-core, k-truss, and related problems have garnered a large amount of interest due to their importance in community detection [14, 63, 64, 65, 68, 73]. It would be interesting to study whether we can solve these problems $O(1)$ rounds in the AMPC model.

Random-walk and Embedding. The AMPC model can potentially help accelerate random-walk based problems, such as PageRank and Personalized PageRank [13, 39, 57] since it efficiently supports random access. Graph embeddings are another impactful data-mining area where expressing random-walk algorithms, such as DeepWalk [58], LINE [66], and NetSMF [59] in AMPC could allow scaling to massive graphs in a reliable, fault-tolerant way.

6 Related Work

Massively Parallel Algorithms. There has been a huge body of work on developing efficient, low round-complexity distributed graph algorithms in the Massively Parallel Computation model [3, 5, 6, 9, 15, 17, 18, 22, 30, 36, 46, 49, 60], including many other papers. In this paper, our focus is on recently proposed Adaptive Massively Parallel Computation model [19], which was also recently studied from a lower-bound perspective by Charikar *et al.* [28].

Distributed Graph Processing. Motivated by the need to process very large graphs, there have been many distributed graph processing frameworks developed in the literature (e.g., [40, 50, 51] among many others). We refer the reader to [52, 69] for surveys of existing work in this area.

7 Conclusion and Open Problems

In this paper we presented new graph algorithms with constant round-complexity in the Adaptive Massively Parallel Computation model. Our theoretical results for these problems address three open problems posed in [19], essentially settling the round complexity of connectivity, minimum spanning forest and matching in this model. Our practical evaluation of these algorithms shows that AMPC algorithms can deliver better performance than corresponding MPC algorithms, without sacrificing fault-tolerance properties.

For future research, can we improve the query complexity of these problems to to linear even for sparse graphs, when $m = O(n)$? It would also be interesting to understand whether the model can be used to solve other fundamental large-scale problems faster, such as string matching, and whether it can be applied to distance problems on graphs.

Acknowledgements

8 References

[1] *AWS Pricing*, accessed September 6, 2020.
https://aws.amazon.com/emr/pricing/.

[2] *Google Cloud Pricing*, accessed September 6, 2020.
https://cloud.google.com/compute/all-pricing.

[3] K. J. Ahn and S. Guha. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 202–211, 2015.

[4] T. Akidau, S. Chernyak, and R. Lax. *Streaming systems: the what, where, when, and how of large-scale data processing.* O'Reilly Media, Inc., 2018.

[5] A. Andoni, A. Nikolov, K. Onak, and G. Yaroslavtsev. Parallel algorithms for geometric graph problems. In *ACM Symposium on Theory of Computing (STOC)*, pages 574–583, 2014.

[6] A. Andoni, Z. Song, C. Stein, Z. Wang, and P. Zhong. Parallel graph connectivity in log diameter rounds. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 674–685, 2018.

[7] Apache Software Foundation. Beam.
https://beam.apache.org.

[8] Apache Software Foundation. Hadoop.
https://hadoop.apache.org.

[9] S. Assadi, M. Bateni, A. Bernstein, V. S. Mirrokni, and C. Stein. Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1616–1635, 2019.

[10] S. Assadi, M. Bateni, and V. Mirrokni. Distributed weighted matching via randomized composable coresets. In *International Conference on Machine Learning (ICML)*, pages 333–343, 2019.

[11] I. T. Association. Infiniband.

[12] I. T. Association. Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.2 Annex a16: RDMA over Converged Ethernet (RoCE), 2010.

[13] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized PageRank. 4(3):173–184, 2010.

[14] V. Batagelj and M. Zaversnik. An $O(m)$ algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[15] M. Bateni, S. Behnezhad, M. Derakhshan, M. Hajiaghayi, R. Kiveris, S. Lattanzi, and V. S. Mirrokni. Affinity clustering: Hierarchical clustering at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 6867–6877, 2017.

[16] P. Beame, P. Koutris, and D. Suciu. Communication Steps for Parallel Query Processing. *J. ACM*, 64(6):40:1–40:58, 2017.

[17] S. Behnezhad, M. Derakhshan, H. Esfandiari, E. Tan, and H. Yami. Brief announcement: Graph matching in massive datasets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 133–136.

[18] S. Behnezhad, M. Derakhshan, and M. Hajiaghayi. Brief announcement: Semi-MapReduce meets congested clique. *CoRR*, abs/1802.10297, 2018.

[19] S. Behnezhad, L. Dhulipala, H. Esfandiari, J. Łącki, V. Mirrokni, and W. Schudy. Massively parallel computation via remote memory access. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 59–68, 2019.

[20] S. Behnezhad, L. Dhulipala, H. Esfandiari, J. Łącki, V. Mirrokni, and W. Schudy. Parallel graph algorithms in constant adaptive rounds: Theory meets practice. *CoRR*, abs/1807.10727, 2020.

[21] S. Behnezhad, L. Dhulipala, H. Esfandiari, J. Łącki, and V. S. Mirrokni. Near-optimal massively parallel graph connectivity. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1615–1636, 2019.

[22] S. Behnezhad, M. Hajiaghayi, and D. G. Harris. Exponentially faster massively parallel maximal matching. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1637–1649, 2019.

[23] S. Behnezhad, M. Hajiaghayi, and D. G. Harris. Exponentially faster massively parallel maximal matching. *CoRR*, abs/1901.03744, 2019.

[24] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 308–317, 2012.

[25] G. E. Blelloch and B. M. Maggs. Algorithms and theory of computation handbook. chapter Parallel Algorithms. Chapman & Hall/CRC, 2010.

[26] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *International World Wide Web Conference (WWW)*, pages 595–601, 2004.

[27] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan. FlumeJava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2010.

[28] M. Charikar, W. Ma, and L.-Y. Tan. Unconditional lower bounds for adaptive massively parallel computation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 141–151, 2020.

[29] I. Corporation. Intel Omni-Path Architecture.
https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-driving-exascale-computing.html.

[30] A. Czumaj, J. Łącki, A. Madry, S. Mitrovic, K. Onak, and P. Sankowski. Round compression for parallel matching algorithms. In *ACM Symposium on Theory of Computing (STOC)*, pages 471–484, 2018.

[31] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[32] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 393–404, 2018.

[33] A. Dragojevic, D. Narayanan, and M. Castro. RDMA reads: To use or not to use? *IEEE Data Eng. Bull.*, 40(1):3–14, 2017.

[34] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *USENIX*

Symposium on Networked Systems Design and Implementation (NSDI), pages 401–414, 2014.

[35] M. Fischer and A. Noever. Tight analysis of parallel randomized greedy MIS. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2152–2160, 2018.

[36] M. Ghaffari, T. Gouleakis, C. Konrad, S. Mitrovic, and R. Rubinfeld. Improved massively parallel computation algorithms for MIS, matching, and vertex cover. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–138, 2018.

[37] M. Ghaffari, F. Kuhn, and J. Uitto. Conditional hardness results for massively parallel computation from distributed lower bounds. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1650–1663, 2019.

[38] M. Ghaffari and J. Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1636–1653, 2019.

[39] D. F. Gleich. PageRank beyond the web. *SIAM Review*, 57(3):321–363, 2015.

[40] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

[41] M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, searching, and simulation in the mapreduce framework. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 374–383, 2011.

[42] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be general and fast. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–16, 2019.

[43] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.

[44] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

[45] R. Kiveris, S. Lattanzi, V. S. Mirrokni, V. Rastogi, and S. Vassilvitskii. Connected components in MapReduce and beyond. In *ACM Symposium on Cloud Computing (SoCC)*, pages 18:1–18:13, 2014.

[46] P. Koutris, S. Salihoglu, D. Suciu, et al. Algorithmic aspects of parallel data processing. *Foundations and Trends® in Databases*, 8(4):239–370, 2018.

[47] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *International World Wide Web Conference (WWW)*, pages 591–600, 2010.

[48] J. Łącki, V. S. Mirrokni, and M. Wlodarczyk. Connected components at scale via local contractions. *CoRR*, abs/1807.10727, 2018.

[49] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 85–94, 2011.

[50] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 340–349, 2010.

[51] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM International Conference on Management of Data (SIGMOD)*, pages 135–146, 2010.

[52] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2), Oct. 2015.

[53] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. The graph structure in the web–analyzed on different aggregation levels. *The Journal of Web Science*, 1(1):33–47, 2015.

[54] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX Annual Technical Conference (ATC)*, pages 103–114, 2013.

[55] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 439–455, 2013.

[56] D. Nanongkai and M. Scquizzato. Equivalence classes and conditional hardness in massively parallel computations. *CoRR*, abs/2001.02191, 2020.

[57] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[58] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 701–710. ACM, 2014.

[59] J. Qiu, Y. Dong, H. Ma, J. Li, C. Wang, K. Wang, and J. Tang. Netsmf: Large-scale network embedding as sparse matrix factorization. In *International World Wide Web Conference (WWW)*, pages 1509–1520. ACM, 2019.

[60] T. Roughgarden, S. Vassilvitskii, and J. R. Wang. Shuffles and circuits: (on lower bounds for modern parallel computation). In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 1–12, 2016.

[61] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *PVLDB*, 11(4):420–431, 2017.

[62] S. Sakr, F. M. Orakzai, I. Abdelaziz, and Z. Khayyat. *Large-Scale Graph Processing Using Apache Giraph*. Springer, 2016.

[63] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek. Incremental k-core decomposition: algorithms and evaluation. *The VLDB Journal*, 25(3):425–447, 2016.

[64] A. E. Sarıýüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and U. V. Çatalyürek. Streaming algorithms for k-core decomposition. *PVLDB*, 6(6), Apr. 2013.

[65] A. E. Sariyüce, C. Seshadhri, and A. Pinar. Local algorithms for hierarchical dense subgraph discovery. *PVLDB*, 12(1):43–56, 2018.

[66] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. Line: Large-scale information network embedding. In *International World Wide Web Conference (WWW)*, pages 1067–1077, 2015.

[67] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: the next generation. In *European Conference on Computer Systems (EuroSys)*, pages 1–14, 2020.

[68] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. X. Yu. I/O efficient core graph decomposition: Application to degeneracy ordering. *IEEE Trans. Knowl. Data Eng.*, 31(1):75–90, 2019.

[69] D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1-2):1–195, 2017.

[70] Y. Yoshida, M. Yamamoto, and H. Ito. An improved constant-time approximation algorithm for maximum matchings. In M. Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 225–234. ACM, 2009.

[71] R. Zadeh and S. Ben-David. A uniqueness theorem for clustering. In J. A. Bilmes and A. Y. Ng, editors, *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 639–646, 2009.

[72] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.

[73] Y. Zhang and J. X. Yu. Unboundedness and efficiency of truss maintenance in evolving graphs. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *ACM International Conference on Management of Data (SIGMOD)*, pages 1024–1041, 2019.