Distributed Graph Simulation: Impossibility and Possibility

Distributed Graph Simulation: Impossibility and Possibility

Wenfei Fan[1,2] Xin Wang[4] Yinghui Wu[1,3] Dong Deng[5]
[1]University of Edinburgh [2]RCBD and SKLSDE Lab, Beihang University [3]Washington State University [4]Southwest Jiaotong University [5]Tsinghua University
{wenfei@inf, x.wang-36@sms}.ed.ac.uk ywu@eecs.wsu.edu dd11@mails.tsinghua.edu.cn

ABSTRACT

This paper studies fundamental problems for distributed graph simulation. Given a pattern query Q and a graph G that is fragmented and distributed, a graph simulation algorithm $\mathcal{A}$ is to compute the matches $Q(G)$ of Q in G. We say that $\mathcal{A}$ is *parallel scalable* in (a) *response time* if its parallel computational cost is determined by the largest fragment F_m of G and the size $|Q|$ of query Q, and (b) *data shipment* if its total amount of data shipped is determined by $|Q|$ and the number of fragments of G, *independent* of the size of graph G. (1) We prove an *impossibility theorem*: there exists *no* distributed graph simulation algorithm that is parallel scalable in *either* response time *or* data shipment. (2) However, we show that distributed graph simulation is *partition bounded*, *i.e.,* its response time depends only on $|Q|$, $|F_m|$ and the number $|V_f|$ of nodes in G with edges across different fragments; and its data shipment depends on $|Q|$ and the number $|E_f|$ of crossing edges only. We provide the first algorithms with these performance guarantees. (3) We also identify special cases of patterns and graphs when parallel scalability is possible. (4) We experimentally verify the scalability and efficiency of our algorithms.

1. INTRODUCTION

Graph pattern matching is widely used to search and analyze, *e.g.,* social graphs, biological data and transportation networks. Given a *graph pattern* Q and a *data graph* G, it is to compute $Q(G)$, the set of all matches of Q in G. In the real world, graphs G are often "big". For example, Facebook has more than 1 billion users with 140 billion links [2]. Moreover, the graphs are often fragmented and distributed. Indeed, (1) social graphs of Twitter and Facebook are geo-distributed to data centers [34], and (2) to query big graphs, one wants to partition the data and leverage parallel computation, *e.g.,* Pregel [26] and GraphLab [22].

These highlight the need for efficient algorithms $\mathcal{A}$ that, given a pattern Q and a graph G that is fragmented into $\mathcal{F} = (F_1, \ldots, F_n)$ and distributed to sites $(S_1, \ldots, S_n)$, compute $Q(G)$. To query big G, we want $\mathcal{A}$ to be *parallel scalable* in

- *response time*, if its parallel computational cost (response time) is determined only by the largest fragment F_m of G and the size $|Q|$ of pattern Q; and
- *data shipment*, if its total amount of data shipped is decided by $|Q|$ and the number $|\mathcal{F}|$ of fragments in $\mathcal{F}$;

independent of the size $|G|$ of the underlying big graph G; here $|G|$ is measured by the total number of nodes and edges in G; similarly for the sizes of fragment $|F_m|$ and pattern $|Q|$.

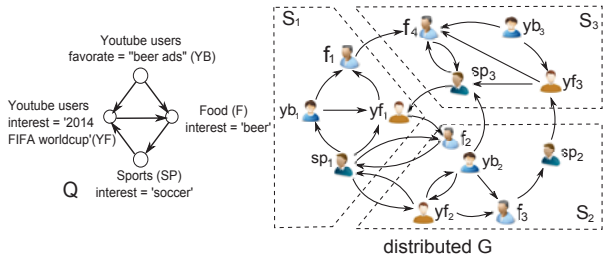


Figure 1: Querying a distributed social network

The need for parallel scalability is evident when querying big distributed graphs: the more processors are available, the smaller the fragments tend to be, and hence, the less response time it takes. That is, it allows us to divide the computation on a big graph G into *parallel computation* on small fragments of G of manageable sizes, *i.e.,* to make big G "small". If it is parallel scalable in data shipment, then network traffic does not substantially increase when G grows.

Example 1: Consider a social graph G consisting of people with different interests: Food (f nodes), Sports (sp), and among Youtube users, beer lovers (yb) and worldcup fans (yf), as depicted in Fig. 1. An edge (A, B) in G indicates a *trusted recommendation* [19] from A to B. For example, edge (f_3, sp_2) indicates that sp_2 trusts the recommendation of f_3 for *e.g.,* beer. Graph G is distributed to sites S_1, S_2 and S_3.

To identify potential customers for a beer brand [21], a company issues a graph pattern Q (Fig. 1). It is to find (1) Youtube users who favor beer ads (YB); (2) Youtube users interested in videos about "2014 FIFA World-cup" (YF), (3) Food lovers and (4) soccer fans (SP). The conditions are that F and YF people trust recommendations from the YB users, and the SP, F and YF folks form a recommendation cycle.

When G is big, it is cost prohibitive to compute $Q(G)$. Graph pattern matching is intractable if it is based on subgraph isomorphism [33], and it takes quadratic time in $|G|$ based on graph simulation [11, 18]. Worse still, data has to be shipped from one site to another as G is distributed.

Proceedings of the VLDB Endowment, Vol. 7, No. 12

With this comes the need for a parallel scalable algorithm $\mathcal{A}$, to allow a high degree of parallelism and to efficiently find potential customers *independent of* $|G|$. □

The practical need raises the following fundamental questions. Is it possible at all to find a distributed algorithm $\mathcal{A}$ that is parallel scalable for graph pattern matching? If not, under what conditions such $\mathcal{A}$ exists? Are there other (weaker) performance bounds that allow $\mathcal{A}$ to scale with $|G|$? While a number of algorithms have been developed for distributed pattern matching (*e.g.,* [10, 15, 29, 30]), and several distributed graph systems are in place [4, 26], to the best of our knowledge, these questions have not been settled.

Contributions. This paper tackles these questions. We focus on graph pattern matching defined with graph simulation [18], as it is commonly used in social community detection [7], biological analysis [23], and wireless and mobile network analyses [16]. While conventional subgraph isomorphism often fails to capture meaningful matches, graph simulation fits into emerging applications with its "many-to-many" matching semantics [7, 11, 18]. Moreover, it is *challenging* since graph simulation is "recursively defined" and has poor data locality [9] (see Section 2 about data locality).

The main contributions of the paper are as follows.

(1) We identify desirable performance guarantees for distributed graph pattern matching algorithms (Section 3). We use *parallel scalability* to characterize that the response time and data shipment are *independent* of the size of graph G.

(2) No matter how desirable, we show that parallel scalability is beyond reach for distributed graph simulation (Section 3). We prove an *impossibility theorem*: there exists *no* algorithm for distributed graph simulation that is parallel scalable in *either* its response time *or* data shipment.

(3) Nonetheless, we identify doable cases for distributed graph simulation with performance guarantees (Section 4). For patterns Q and distributed graphs G, we provide a distributed simulation algorithm that is *partition bounded*. That is, its response time depends only on the largest fragment F_m of G, the size $|Q|$ of Q, and the number $|V_f|$ of nodes with edges across different fragments. Better still, its data shipment is bounded by $O(|E_f||Q|)$, where E_f is the set of all crossing edges. In practice $|V_f|$ and $|E_f|$ are typically *much smaller* than $|G|$, and $|Q|$ is small. Hence both the response time and data shipment of the algorithm are often *independent of* $|G|$, *i.e.,* they are not a function of $|G|$.

(4) When either query Q or graph G is an acyclic direct graph (*i.e.,* a DAG), we show that better bounds exist (Section 5). We develop a distributed simulation algorithm for DAGs that is *parallel scalable in response time* under certain conditions. When G is a tree, we give an algorithm that is parallel scalable in data shipment. To the best of our knowledge, these are the first distributed graph simulation algorithms that have these performance bounds.

The bounds of our algorithms are shown in Table 1 (the last three rows), compared with prior work. They remain intact *no matter how* graphs are partitioned and distributed.

(5) Using real-life and synthetic graphs, we experimentally verify the scalability and efficiency of our algorithms. We find that our algorithms scale well with graphs G: their response time and data shipment are *not a function of* $|G|$. The algorithms are efficient: they take 21 seconds for cyclic queries on graphs with 18 million nodes and edges. Our algorithms substantially outperform previous algorithms for distributed graph simulation: on average they are *3.5 and 21.6 times faster*, and ships *3 and 2 orders of magnitude less data* than those of [25] and [14], respectively. On DAGs, they are 4.7 times faster than the algorithm of [25].

To the best of our knowledge, (1) the results are among the first that tell us what is *doable* and what is *undoable* for distributed graph simulation. (2) Our algorithms possess the lowest known bounds on response time and data shipment. (3) In addition, the algorithms highlight a new approach for distributed query processing, by *combining* partial evaluation [10, 12, 30] and message passing [22, 26] (see details shortly). Taken together with approximation algorithms [27] that minimize $|F_m|$ and $|V_f|$ in graph partitioning, the algorithms are a step toward making distributed graph pattern matching scalable with real-life graphs.

Related Work. We categorize related work as follows.

Distributed graph databases. There have been several graph systems for storing and querying distributed graphs [22, 26, 36]. Microsoft Trinity [36] is a distributed graph storage and (SPARQL) querying system. Facebook TAO [34] is a geographically distributed system that supports simple graph queries (*e.g.,* neighborhood retrieval). Below we discuss two representative systems, Pregel [26] and GraphLab [22].

Pregel [26] is a distributed graph system based on synchronized message passing. It partitions a graph into clusters, and selects a master machine to assign each cluster to a slave machine. A graph algorithm is executed in a series of supersteps, during which slave machines send messages to each other and change their status (voting or halt). The master machine communicates with slaves after each superstep, to guide them for the next step. The algorithm terminates if all the nodes halt. Several graph query algorithms (distance, PageRank, etc.) are supported by Pregel (see [26]).

GraphLab [22] is an asynchronous parallel-computation framework for graphs, optimized for scalable machine learning and data mining algorithms. Given data graph G, a user-defined update function modifies the data attached to the nodes in G, and a sync operation gathers final results. The major difference between Pregel and GraphLab is that the latter decouples the scheduling of computation from message passing, by allowing "caching" information at edges.

These frameworks provide system-level optimizations for *e.g.,* usability and scalability. However, it is hard to assure provable performance bounds in these frameworks, especially for graph pattern matching in arbitrarily partitioned graphs. (1) Message passing of Pregel may serialize operations that can be conducted in parallel, hence incur excessive network traffic. (2) GraphLab reduces, to some extent, unnecessary messages. However, the improvement is only verified empirically, and highly depends on update and partitioning strategy. In fact, we show (Section 3) that the impossibility theorem of this work also holds in both frameworks.

Distributed graph query evaluation. Several algorithms have been developed for querying distributed graphs with performance guarantees, via *partial evaluation* or *message passing*. Methods based on partial evaluation [10, 12, 15, 25, 29, 30] specify a coordinator site S_0 and a set of worker sites. Upon receiving a query Q, S_0 posts Q to workers. Each worker

Query	Datagraph	Type	PT	DS
XPath [10]	XML trees	P	$O(\|Q\|\|F_m\| + \|Q\|\|\mathcal{F}\|)$	$O(\|Q\|\|\mathcal{F}\|)$
regular path [5]	XML trees	P	$O(\|Q\|\|V_f\|\|F_m\| + \|F_m\|\|\mathcal{F}\|)$	$O(\|E_f\|^2)$
regular path [30]	general graphs	P	$O(\|Q\|\|V_f\|\|F_m\| + \|V_f\|^2\|\mathcal{F}\|)$	$O(\|E_f\|^2)$
regular path [29]	general graphs	M	-	$O(\|Q\|^2\|G\|^2)$
regular path [12]	general graphs	P	$O((\|F_m\| + \|V_f\|^2)\|Q\|^2)$	$O(\|Q\|^2\|V_f\|^2)$
bisimulation [6]	general graphs	M	$O(\frac{\|V\|^2+\|V\|\|E\|}{\|\mathcal{F}\|})$ (total)	$O(\|V\|^2)$
simulation [25]	general graphs	M	$O((\|V_q\|+\|V\|)(\|E_q\|+\|E\|))$	$O(\|G\| + 4\|V_f\| + \|\mathcal{F}\|\|Q\|)$
simulation (this work)	general graphs	P&M	$O((\|V_q\|+\|V_m\|)(\|E_q\|+\|E_m\|)\|V_q\|\|V_f\|)$	$O(\|E_f\|\|V_q\|)$
simulation (this work)	DAGs	P&M	$O(d(\|V_q\|+\|V_m\|)(\|E_q\|+\|E_m\|) + \|Q\|\|\mathcal{F}\|)$	$O(\|E_f\|\|V_q\|)$
simulation (this work)	trees	P	$O(\|Q\|\|F_m\| + \|Q\|\|\mathcal{F}\|)$	$O(\|Q\|\|\mathcal{F}\|)$

Table 1: Distributed graph pattern matching: performance bounds

performs *partial evaluation* to compute a partial result *in parallel*. The coordinator S_0 then collects and assembles the partial results to get the complete result. Methods based on message passing [5, 22, 26], on the other hand, exchange intermediate results between two sites. Each site is repeatedly visited until a complete answer is generated.

We summarize the performance bounds of the algorithms in Table 1, for data shipment (DS) and parallel computation time (PT), categorized by partial evaluation (P) and message passing (M). For a fragmentation $\mathcal{F}$ that partitions G, V_f is the set of nodes with edges "crossing" sites, E_f is the set of all crossing edges. For G (resp. query Q and fragment F_m), V and E (resp. V_q and E_q, V_m and E_m) are its set of nodes and edges, respectively. We use d to denote the diameter of Q, *i.e.,* the longest shortest path in Q.

From Table 1 we can see the following. (1) Partial evaluation often guarantees bounds on response time and network traffic; however, redundant local computation may be conducted. (2) Message passing often incurs unbounded data shipment, and is hard to have provable bounds on response time. On the other hand, local evaluation can be minimized with specifically designed routing and scheduling plans.

To the best of our knowledge, our algorithms (Sections 4 and 5) are the first that integrate partial evaluation and message passing for graph pattern matching, with provable performance guarantees on data shipment and response time.

Closer to our work are [14, 25], which also study distributed graph simulation, by scheduling inter-site message passing. In [25], subgraphs from different sites are collected to a single site to form a directly query-able graph, where matches can be determined for the strongly connected components in the query. In [14], a vertex-centric model is developed for distributed simulation, following Pregel.

As shown in Table 1, the response time and data shipment of the algorithm of [25] are functions of *the size of the entire G*. No performance guarantees for data shipment or response time are given in [14]. In contrast, (1) we give algorithms that are parallel scalable when G or Q satisfies certain conditions; (2) we develop a partition bounded algorithm for general Q and G, *i.e.,* its response time and data shipment are not a function of $|G|$; one major difference between our algorithms and [25] is that instead of shipping large chunks of data to a single site, we only ship the truth values among the sites; this significantly reduces the data shipment and query processing time; and (3) we prove the impossibility of parallel scalability, while [14, 25] did not study what is doable and what is undoable.

Hardness of distributed query processing. The hardness has been studied for reducing communication costs by graph partitioning [8, 27] and for the message-passing model [35]. Performance bounds are established for MapReduce algorithms [31] on *e.g.,* network traffic, MapReduce steps, and optimal local computation. However, these bounds are for MapReduce operators in *e.g.,* sorting and aggregation, not for graph pattern matching. This work provides both impossibility results and algorithms with provable performance bounds, particularly for distributed graph simulation.

2. DISTRIBUTED GRAPH SIMULATION

In this section we first review graphs, patterns, data locality and graph simulation [18] (Section 2.1). We then extend graph simulation to distributed graphs (Section 2.2).

2.1 Graphs, Patterns and Graph Simulation

Data graphs. A *data graph* is a node-labeled, directed graph $G = (V, E, L)$, where (1) V is a finite set of data nodes; (2) $E \subseteq V \times V$, where $(v, w) \in E$ denotes a *directed* edge from node v to w; and (3) $L(\cdot)$ is a function such that for each node v in V, $L(v)$ is a label from an alphabet Σ. Intuitively, $L(\cdot)$ specifies *e.g.,* interests, social roles, ratings [20].

To simplify the discussion, we do not explicitly mention edge labels. Nonetheless, our techniques can be readily adapted for edge labels: for each labeled edge e, we can insert a "dummy" node to represent e, carrying e's label.

Pattern graphs. A *pattern query* is a directed graph $Q = (V_q, E_q, f_v)$, where (1) V_q is the set of *query nodes*, (2) E_q is the set of *query edges*, and (3) $f_v(\cdot)$ is a function defined on V_q such that for each node $u \in V_q$, $f_v(u)$ is a label in Σ.

Graph simulation [18]. A graph G *matches* a pattern Q if there exists a binary relation $R \subseteq V_q \times V$ such that

(1) for each query node $u \in V_q$, there exists a node $v \in V$ such that $(u, v) \in R$, referred to as a *match* of u; and

(2) for each pair $(u, v) \in R$, (a) $f_v(u) = L(v)$, and (b) for each query edge (u, u') in E_q, there exists an edge (v, v') in graph G such that $(u', v') \in R$.

It is known that if G matches Q, then there exists a *unique maximum* relation [18], referred to as $Q(G)$. If G does not match Q, $Q(G)$ is the empty set. Moreover, $Q(G)$ can be computed in $O((|V_q| + |V|)(|E_q| + |E|))$ time [11, 18].

We consider two types of pattern queries. (1) A *Boolean pattern* Q returns true on G if G matches Q, and false otherwise. (2) A *data selecting query* Q returns the maximum match $Q(G)$. That is, a Boolean Q simply decides whether G matches Q, while a data selecting Q returns $Q(G)$.

We denote $|V_q| + |E_q|$ as $|Q|$, and $|V| + |E|$ as $|G|$.

Example 2: Consider Q and G of Fig. 1. G matches Q with the unique maximum match, where yb_2, yb_3 match YB, f_2, f_3, f_4 match F, and all yf and sp are the matches of YF and SP, respectively. Here f_1 does not match F since no SP nodes

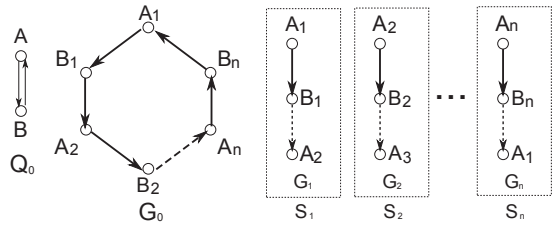


Figure 2: Lack of data locality

trust his recommendation. The answer to Q is true if Q is a Boolean query, and is the relation if Q is data selecting. □

Data locality. A class of graph pattern queries Q is said to have *data locality* if for any graph G and any node v in G, one can decide whether v is a match of a query node u in Q locally, by inspecting only those nodes of G that are within d hops from or to v, where d is determined only by $|Q|$.

For example, graph pattern matching defined in terms of subgraph isomorphism [33] and strong simulation [24] have data locality. However, graph simulation does *not* have it. Data locality makes distributed query evaluation easier since only a bounded number of sites may have to be visited [9].

Example 3: Consider Q_0 and G_0 of Fig. 2. Each node A_i (resp. B_i) has label A (resp. B). One may verify that as a Boolean pattern query, $Q_0(G_0) =$ true, while as a data selecting query, $Q_0(G_0) = \{(A, A_i), (B, B_i) \mid (i \in [1, n])\}$. Note that Q_0 does not have data locality: although it has only 2 edges, to decide if a node v in G_0 is a match of a node in Q_0, those nodes of G_0 even n hops from v may have to be visited and inspected. In contrast, to determine whether a subgraph of G_0 with node v is isomorphic to Q_0, it suffices to visit only the nodes of G_0 within 2 hops of v. □

Graph simulation fits better than strong simulation [24] in, *e.g.,* social community detection. The latter may miss potential matches, *e.g.,* the node yb_2 for YB in Fig. 1. We focus on graph simulation as it finds more potential matches, and it is more challenging due to poor data locality.

2.2 Distributed Graph Pattern Matching

We next present distributed graph simulation.

Distributed data graphs. A *fragmentation* $\mathcal{F}$ of a graph $G = (V, E, L)$ is $(F_1, \ldots, F_n)$, where each *fragment* F_i is specified by $(V_i \cup F_i.O,\ E_i,\ L_i)$ such that

(a) $(V_1, \ldots, V_n)$ is a partition of V,

(b) $F_i.O$ is the set of nodes v' such that there exists an edge $e = (v, v')$ in E, $v \in V_i$ and node v' is in *another fragment*; we refer to v' as a *virtual node* and e as a *crossing edge*; and

(c) $(V_i \cup F_i.O, E_i, L_i)$ is a subgraph of G induced by $V_i \cup F_i.O$.

We assume *w.l.o.g.* that each F_i is stored at site S_i for $i \in [1, n]$. For multiple fragments residing in the same site, they are simply treated as a single fragment.

Here F_i contains "local" nodes in V_i, and virtual nodes in $F_i.O$ from other fragments. The set E_i consists of (a) edges between "local" nodes in V_i and (b) crossing edges from local nodes in V_i to virtual nodes. In a distributed social graph, crossing edges are indicated by either IRIs (universal unique IDs) or semantic labels of the virtual nodes [26, 28].

We also use $F_i.I$ to denote the set of *in-nodes* of F_i, *i.e.,* those nodes $v' \in V_i$ such that there exists an edge (v, v') *incoming* from a node v in another fragment F_j, *i.e.,* v' is a virtual node in F_j. Note that $\bigcup_{i\in[1,n]} F_i.O = \bigcup_{i\in[1,n]} F_i.I$.

symbols	notations
$\mathcal{F}$	Graph fragmentation $(F_1, \ldots, F_n)$
$\lvert\mathcal{F}\rvert$	The number of fragments in $\mathcal{F}$
$F_i.I$	The set of in-nodes in a fragment F_i
$F_i.O$	The set of virtual nodes in a fragment F_i
V_f	$\bigcup_{i\in[1,n]} F_i.O$
E_f	The set of all crossing edges in $\mathcal{F}$

Table 2: Notations: graphs and queries

We will use the following notations:

- $V_f = \bigcup_{i\in[1,n]} F_i.O$ is the set of all virtual nodes in $\mathcal{F}$:
- E_f is the set of all crossing edges in $\mathcal{F}$; and
- $|\mathcal{F}|$ denotes the number of fragments in $\mathcal{F}$.

We summarize the notations in Table 2. Note that $|E_f| \leq |V_f| * d_f$, where d_f is the average out-degree of nodes in V_f.

Distributed graph simulation. Given a pattern query Q and a fragmentation $\mathcal{F}$ of a graph G, a *distributed pattern matching* algorithm $\mathcal{A}$ computes the answer to Q in G, *i.e.,* a truth value if Q is a Boolean pattern query, and the unique, maximum match $Q(G)$ if Q is a data selecting query.

Example 4: As shown in Fig. 1, a fragmentation of G is (F_1, F_2, F_3), where F_1, F_2 and F_3 are stored in sites S_1, S_2 and S_3, respectively. In fragment F_1, $F_1.O$ consists of virtual nodes f_4, f_2 and yf_2, set $F_1.I$ of the in-nodes contains sp_1 and yf_1, and the crossing edges are (f_1, f_4), (yf_1, f_2), (sp_1, yf_2) and (sp_1, f_2); similarly for F_2 and F_3.

To decide that f_3 matches F, for instance, any algorithm has to find a path from f_3 in G, ending with a cycle of *sp*, *yf* and *f* nodes. Such a cycle exists: f_3, sp_2, yf_3, f_4, sp_3, yf_1, f_2, sp_1, yf_2, across all three fragments. To find the cycle, the algorithm has to *ship data* between different sites.

As another example, Figure 2 depicts a fragmentation $\mathcal{F}_0$ of a graph G_0. The fragments are distributed over n sites such that site S_i contains a single edge (A_i, B_i) and a virtual node A_{i+1}. This fragmentation demonstrates the extreme case when V_f consists of all the nodes in G_0. □

3. UNDOABLE AND DOABLE

In this section we first present parallel scalability. We prove that, however, parallel scalability is beyond reach in practice for distributed graph simulation (Section 3.1). In light of this, we propose a weaker notion of partition boundedness, and show that partition boundedness is achievable for distributed graph simulation. We also identify special cases that are parallel scalable (Section 3.2).

3.1 Undoable: Parallel Scalability

A naive algorithm for distributed graph simulation is as follows: given a pattern Q and a graph G that is fragmented and distributed, it ships all the fragments of G to a single site, and uses a *centralized* algorithm to compute the answer to Q. This approach ships data almost as large as $|G|$, and takes at least $O((|V_q| + |V|)(|E_q| + |E|))$ time. The cost is often prohibitive when G is big. It may not even be feasible in distributed applications with limited bandwidth, space and time [17, 35]. To this end, we advocate the following performance guarantees for distributed graph algorithms.

We focus on two metrics for computing $Q(G)$: (a) *response time*, the amount of time to compute $Q(G)$ from the time

when Q is issued; and (b) *data shipment*, the total amount of data shipped between the sites in order to compute $Q(G)$.

Parallel scalability. We say that a distributed graph simulation algorithm $\mathcal{A}$ is *parallel scalable*

- *in response time* if for all patterns Q, graphs G and all fragmentations $\mathcal{F}$ of G, its cost for parallelly computing $Q(G)$ is bounded by a polynomial in the sizes $|Q|$ and $|F_m|$, where F_m is the largest fragment in $\mathcal{F}$; and
- *in data shipment* if it ships at most a polynomial amount of data in $|Q|$ and $|\mathcal{F}|$, where $|\mathcal{F}|$ is the number of fragments (sites) involved in communication;

both *independent of* the size of the entire graph G.

If an algorithm is parallel scalable in response time, then one can partition a big graph and distribute its fragments to different processors, such that the more processors are available, the less response time it takes, *i.e.,* this notion aims to characterize the degree of parallelism. If an algorithm is parallel scalable in data shipment, then it scales with $|G|$ when G grows (note that $|\mathcal{F}|$ is typically much smaller than G).

Impossibility theorems. No matter how desirable, however, we show below that it is impossible to find a parallel scalable algorithm for distributed graph simulation.

Theorem 1: *There exists no algorithm for distributed graph simulation that is parallel scalable in (1) either response time (2) or data shipment, even for Boolean pattern queries.* □

Proof sketch: We prove (1) and (2) by contradiction. For the lack of space we defer the detailed proof to [3].

(1) Assume that there exists a distributed graph simulation algorithm $\mathcal{A}$ that is parallel scalable in response time. Then there exist a Boolean pattern Q_0, a graph G_0 and a fragmentation $\mathcal{F}_0$ of G_0 of the form shown in Fig. 2 (see Examples 3 and 4), such that $\mathcal{A}$ does not correctly decide whether G_0 matches Q_0. Indeed, if $\mathcal{A}$ is parallel scalable, then it takes *constant time* t when processing Q_0 on $\mathcal{F}_0$, since $|Q_0|$ is a constant, and each fragment of $\mathcal{F}_0$ has a constant size. However, $\mathcal{F}_0$ has n fragments, for a "variable" n. To decide whether G_0 matches Q_0, we show that information has to be assembled from m sites and analyzed, for $t < m \leq n$.

(2) Assume that there exists an algorithm $\mathcal{A}$ that is parallel scalable in data shipment. We show that there exist a Boolean pattern Q_1, a graph G_1 and a fragmentation $\mathcal{F}_1$ of G_1, such that $\mathcal{A}$ does not correctly decide whether G_1 matches Q_1. We use the same Q_0 above as Q_1, a variation G_1 of G_0, and an $\mathcal{F}_1$ with *two* fragments, one consisting of all the A nodes of G_1 and the other with all the B nodes. Then only *a constant amount* c of data can be sent by $\mathcal{A}$, since $|Q_0|$ and $|\mathcal{F}_1|$ are constants. However, we show that to correctly decide whether G_1 matches Q_0, data about at least m nodes has to be sent, where $c < m \leq n$, and n is the number of nodes in a fragment of $\mathcal{F}_1$. □

Remarks. The result is generic: it holds on distributed models in which each site makes decisions based on the messages received and local evaluation, *e.g.,* partial evaluation models [10, 12, 30]. It also holds on vertex-centric graph processing systems, *e.g.,* Pregel [26] and GraphLab [22], in which each node makes decision on local computation and message sending. One can verify that the proof above applies to vertex-centric computation, regardless of *e.g.,* how the asynchronous local strategy schedules the messages (GraphLab), or how a synchronized superstep coordinates the shipment of messages (Pregel). See [3] for details.

3.2 Doable: Partition Boundedness

Theorem 1 suggests that we consider weaker performance guarantees for distributed graph simulation.

Partition boundedness. We say that an algorithm $\mathcal{A}$ for distributed graph simulation is *partition bounded*

- *in response time* if its parallel computation cost is a polynomial function in $|Q|$, $|F_m|$ and $|V_f|$ (or $|E_f|$), and
- *in data shipment* if the total data shipped is bounded by a polynomial in $|Q|$ and $|E_f|$ (or $|V_f|$).

That is, $\mathcal{A}$ depends on how G is partitioned, not on its size $|G|$. For a partition $\mathcal{F}$ (thus fixed $|V_f|$ and $|E_f|$), *neither* its response time *nor* data shipment is measured in the size of G. In practice $|V_f|$ and $|E_f|$ are typically much smaller than $|G|$; hence, if $\mathcal{A}$ is partition bounded, it often scales well with big G. In addition, approximation graph partitioning methods are already in place [27] to minimize $|V_f|$ and $|E_f|$, possibly making the sizes of V_f and E_f independent of $|G|$.

Positive results. Despite Theorem 1, we show that it is still possible to find efficient algorithms for distributed graph simulation with performance guarantees.

Theorem 2: *There exists an algorithm for distributed graph simulation that is partition bounded in both response time and data shipment. Over any fragmentation $\mathcal{F}$ of a graph G, it evaluates a pattern query $Q = (V_q, E_q, f_v)$*

- *in $O(|V_f||V_q|(|V_q| + |V_m|)(|E_q| + |E_m|))$ time, and*
- *ships at most $O(|E_f||V_q|)$ amount of data,*

where $F_m = (V_m, E_m, L_m)$ is the largest fragment in $\mathcal{F}$. □

When either Q or G is a directed acyclic graph (*i.e.,* DAG), we have better bounds, and moreover, *parallel scalability* in response time when the number $|\mathcal{F}|$ of fragments is fixed.

Theorem 3: *When either graph G or pattern Q is a DAG, there exists an algorithm that computes $Q(G)$*

- *in $O(d(|V_q| + |V_m|)(|E_q| + |E_m|) + |Q||\mathcal{F}|)$ time, and*
- *ships at most $O(|E_f||V_q|)$ amount of data,*

where d is the diameter of Q, and $\mathcal{F}$ is a fragmentation of G. If $|\mathcal{F}|$ is fixed, it is parallel scalable in response time. □

When G is a tree, parallel scalability is achievable in data shipment, and furthermore, possible in response time when $|\mathcal{F}|$ is fixed. The bounds below are the same as those for evaluating XPath queries on distributed XML trees [10]. That is, we show that the bounds of [10] on XPath extend to *distributed graph simulation* on trees.

Corollary 4: *When G is a tree and each fragment of $\mathcal{F}$ is connected, there exists a parallel scalable algorithm in data shipment. More specifically, it (a) is in $O(|Q||F_m| + |Q||\mathcal{F}|)$ time, and (b) ships at most $O(|Q||\mathcal{F}|)$ amount of data. If $|\mathcal{F}|$ is fixed, it is also parallel scalable in response time.* □

We will prove Theorem 2 in Section 4, and Theorem 3 and Corollary 4 in Section 5, by providing such algorithms.

Remarks. (1) Our performance bounds and techniques do not require any particular fragmentation strategy, while they work better on fragmentations that minimize $|V_f|$ and $|E_f|$.

(2) Table 1 shows that only the algorithms of [10] guarantee parallel scalability in data shipment. Those of [5, 12, 30]

are partition bounded in data shipment, and among these, only [12] is partition bounded in response time. These algorithms are for either XML trees [5] or regular path queries [12, 30]. Distributed graph simulation is *more challenging*, and we are not aware of any prior algorithms for distributed graph simulation that are partition bounded. The algorithms of [6, 25], for instance, require to ship data of $O(|G|)$ size, *i.e.,* the entire graph, and take as much time as the naive algorithm given above in the worst case.

4. PARTITION BOUNDED ALGORITHMS

In this section we prove Theorem 2 by developing an algorithm for distributed graph simulation with the desired bounds. In contrast to conventional distributed algorithms, the algorithm leverages *both* partial evaluation and message passing. (1) As opposed to partial evaluation, it adopts asynchronous message passing to direct partial results among fragments. (2) In contrast to vertex-centric models (*e.g.,* Pregel, GraphLab) where each node has a computing node for local computation, it conducts local evaluation on a *fragment* with effective optimization.

We first present a baseline algorithm in Section 4.1, and then improve it with optimization techniques in Section 4.2.

4.1 Algorithm for Graph Simulation

We start with the baseline algorithm, denoted as dGPM. We first study data selecting queries, and then Boolean ones.

Algorithm dGPM uses partial evaluation to compute partial answers on a fragment at each local site in parallel. Each site then refines its partial answer upon receiving messages from others, and sends updated answers to others, guided by the *dependency* among the sites, *i.e.,* whether a site needs the values of its virtual nodes from other sites. The process repeats until no change happens at any site.

We first present auxiliary structures used by dGPM. Consider $Q = (V_q, E_q, L_q)$, $G = (V, E, L)$, and a fragmentation $\mathcal{F} = (F_1, \ldots, F_n)$ of G, where each F_i is stored at site S_i.

Partial answers. A straightforward way to define partial answer for a site S_i is to induce the subgraph of F_i from all the candidate nodes, assuming that they are all matches [25]. However, this incurs unbounded data shipment. Instead of shipping data of F_i, we send Boolean variables denoting partial results of Q on local fragment F_i, defined as follows.

(a) With each node v in F_i and each pattern node u in Q, we associate a Boolean variable $X_{(u,v)}$ to indicate whether v is a match of u. All such variables for v form a *Boolean vector* $v.\mathsf{rvec}$ of size $|V_q|$, for all pattern nodes u in Q.

(b) A *partial answer* is a set L_i of vectors $v.\mathsf{rvec}$ consisting of all the *in-nodes* v in F_i, such that $v.\mathsf{rvec}[u]$ is defined by a Boolean formula *only* in terms of the Boolean variables of the *virtual nodes* in F_i. We say that v is *unevaluated* for u if the truth value of $X_{(u,v)}$ is not yet known.

Local dependency graphs. A local dependency graph at site S_i keeps track of all the sites with virtual nodes as in-nodes at S_i, More specifically, each site S_i stores a local dependency graph $G_d^i = (V_d^i, E_d^i, A_d^i)$, where

- each node S_j in V_d^i represents a site,
- there is an edge (S_j, S_i) in E_d^i if there is a virtual node v_j in F_j as an in-node in F_i; and

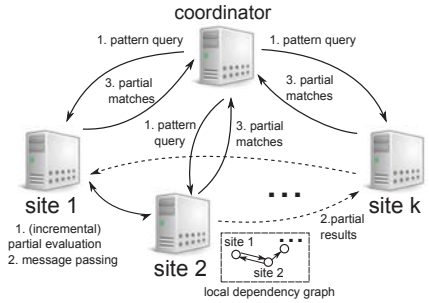


Figure 3: Distributed pattern matching: framework

- a function $A_d^i(\cdot)$ on E_d such that for each edge (S_j, S_i), $A_d^i(S_j, S_i)$ is the set of all virtual nodes v_j in F_j (resp. in-nodes in F_i) as described above.

Such a G_d^i is determined by fragmentation $\mathcal{F}$ only and is small. Each site S_i can compute G_d^i offline *in parallel*, by sharing the identifiers of its virtual and in-nodes [26,28] with other sites, using hashing [26] or indexing techniques [36].

Example 5: Consider Q and G of Fig. 1. Each site S_i keeps a local dependency graph G_d^i. For site S_3, G_d^3 contains edges (S_1, S_3) (annotated with f_4) and (S_2, S_3) (annotated with $\{sp_3, yf_3\}$), as site S_1 has a virtual node f_4 as an in-node in S_3; similarly for S_2. A partial answer at *e.g.,* site S_3 is a set of Boolean vectors, one for each of its in-nodes sp_3, yf_3 and f_4. For, *e.g., sp*$_3$, the truth value of an entry $X_{(\mathsf{SP}, sp_1)}$ in its associated vector indicates if sp_3 is a match for SP. □

Algorithm. Algorithm dGPM computes and refines the partial answers in three phases, as depicted in Fig. 3.

(1) It first conducts partial evaluation (see Section 1). Upon receiving Q at a coordinator site S_c, it ships Q to each site S_i in which fragment F_i in $\mathcal{F}$ resides. Each site computes a *partial answer in parallel*, by invoking procedure lEval (to be given shortly), which keeps track of its in-nodes and virtual nodes that cannot be locally decided as matches.

(2) Each site then follows a *local message passing procedure* lMsg to (a) ship its partial results to other sites guided by its local dependency graph, and (b) receive updated partial results from other sites and use procedure lEval to refine its own partial answer using the input; all sites conduct these *in parallel* and *asynchronously*. If new result appears at some site, the site sends a flag to S_c to indicate the change. The communication repeats until no change happens at any site (indicated by a Boolean variable changed = false at S_c), *i.e.,* no more invalid matches exist and a fixpoint is reached.

(3) Finally, coordinator S_c collects partial matches $Q(F_i)$ from each site, and takes their union as $Q(G)$. If there exists some node u in Q that does not appear in $Q(G)$, S_c returns $\emptyset$, *i.e.,* no match exists. Otherwise, it outputs $Q(G)$.

We next present the two procedures: lEval and lMsg.

Local evaluation. We start with lEval, shown in Fig. 4. The procedure first initializes a list L_i to store the partial results (line 1). It then initializes vector $v.\mathsf{rvec}$ for each node v in F_i and each node u in Q as follows (lines 2-8). For v and u with the same label, if u has no children, then $v.\mathsf{rvec}[u]$ is assigned true (lines 4-5). Otherwise $v.\mathsf{rvec}[u]$ is assigned a Boolean variable $X_{(u,v)}$ (lines 6-7). The variable $X_{(u,v)}$ is assigned false if u and v have distinct labels (line 8).

Procedure IEval/* executed locally at each site S_i, in parallel */

Input: A fragment F_i, and a pattern query Q.
Output: Partial answer to Q in F_i.

```
1.  list L_i := ∅;
2.  for each node v in F_i do
3.    for each node u ∈ V_q do
4.      if L(v) = f_v(u) and u has no child then
5.        v.rvec[u] := true;
6.      else if L(v) = f_v(u) then
7.        v.rvec[u] := X_(u,v);
8.      else v.rvec[u] := false;
9.  compute Q(F_i) by incorporating Boolean variables;
10. for each node v_j ∈ F_i.I do
11.   update Boolean equations in v_j.rvec and L_i;
```

Procedure IMsg/* executed locally at each site S_i, in parallel */

Input: Updated partial answer L_i, and local dependency graph G_d^i.
Output: A set of sites and corresponding messages to be sent.

```
1.  for each in-node v in F_i.I do
2.    for each updated Boolean variable X_(u,v)=false in L_i do
3.      for each edge (S_j, S_i) in G_d^i
        annotated with v (v ∈ A_d^i(S_j, S_i)) do
4.        set L_Sj := L_Sj ∪ {X_(u,v)};
5.  for each site S_j and X_(u,v) ∈ L_Sj do
6.    send X_(u,v) to site S_j;
```

Figure 4: Procedure IEval and IMsg

After these, IEval computes $Q(F_i)$ by invoking a revised graph simulation algorithm of [11,18] (line 9). More specifically, we construct Boolean equations in L_i. Recall that for each node v in G and each u in Q, v matches u if and only if (a) $f_v(u) = L(v)$, and (b) for each child u' of u, there exists a child v' of v, such that v' matches u'. Thus, $X_{(u,v)}$ can be deduced from the vectors of the children of v as follows:

$$X_{(u,v)} = \bigwedge(\bigvee X_{(u_i,v_j)}),$$

for each pair of child u_i of u and child v_j of v with the same label. Therefore, $X_{(u,v)}$ is defined by a Boolean equation in terms of the variables associated with the children of v.

Leveraging this property, IEval computes $Q(F_i)$. To cope with Boolean variables, (1) it always *assumes* the unevaluated virtual nodes as match candidates (*i.e.,*true), and (2) it traverses F_i, instantiates the variables with their truth values whenever possible, and moreover, reduces equations such that for each *in-node* v of F_i, its equations are defined in terms of variables associated with *virtual nodes* only.

After the initial process, upon receiving newly updated variables of virtual nodes from other sites (see procedure IMsg), IEval reevaluates the Boolean equations, and attempts to instantiate variables for unevaluated nodes by evaluating the Boolean equations of in-nodes with the new truth values of virtual nodes. The updated variables are prepared to trigger a local message passing to other sites following IMsg.

Example 6: Consider Q and G of Fig. 1. dGPM first posts Q to each site, where IEval is invoked in parallel to compute Boolean equations. At F_1, *e.g.,* it assigns $X_{(\mathsf{F},f_4)}$, $X_{(\mathsf{F},f_2)}$, $X_{(\mathsf{YF},yf_2)}$ to the virtual nodes, and reduces the Boolean equations for each node using these variables only. (a) It infers *e.g.,* $X_{(\mathsf{YF},yf_1)} = X_{(\mathsf{F},f_2)}$, following query edge (YF, F). (b) For sp_1, it computes sp_1.rvec[SP] as $X_{(\mathsf{YF},yf_2)} \vee X_{(\mathsf{YF},yf_1)}$, following query edge (SP, YF). By $X_{(\mathsf{YF},yf_1)} = X_{(\mathsf{F},f_2)}$ from (a), it reduces sp_1.rvec[SP] as $X_{(\mathsf{YF},yf_2)} \vee X_{(\mathsf{F},f_2)}$.

After the parallel evaluation, the Boolean equations in each L_i are shown in the table below.

fragment	in-node	Boolean equations
F_1	yf_1	$X_{(\mathsf{YF},yf_1)} = X_{(\mathsf{F},f_2)}$
	sp_1	$X_{(\mathsf{SP},sp_1)} = X_{(\mathsf{YF},yf_2)} \vee X_{(\mathsf{F},f_2)}$
F_2	f_2	$X_{(\mathsf{F},f_2)} = X_{(\mathsf{SP},sp_1)}$
	yf_2	$X_{(\mathsf{YF},yf_2)} = X_{(\mathsf{YF},yf_3)}$
F_3	f_4	$X_{(\mathsf{F},f_4)} = X_{(\mathsf{YF},yf_1)}$
	sp_3	$X_{(\mathsf{SP},sp_3)} = X_{(\mathsf{YF},yf_1)}$
	yf_3	$X_{(\mathsf{YF},yf_3)} = X_{(\mathsf{YF},yf_1)}$

For each in-node (*e.g., sp_1 of F_1*), its vector is defined only with the Boolean variables of virtual nodes (*e.g., f_2 and yf_2*). Note that the vectors of some "local" nodes (*e.g., yb_2*.rvec[YB]) are also updated (*e.g.,* to $X_{(\mathsf{YF},yf_3)}$). Although $X_{(\mathsf{YB},yb_2)} = X_{(\mathsf{YF},yf_2)} \wedge X_{(\mathsf{F},f_3)}$, IEval finds that $X_{(\mathsf{YB},yb_2)}$ can be defined by using $X_{(\mathsf{YF},yf_3)}$ only. □

Message passing. We next present procedure IMsg, shown in Fig. 4. Given local dependency graph G_d^i and updated partial answer at site S_i, IMsg dynamically generates messages and determines which sites to send the messages. At each site S_i, after IEval is completed, IMsg (1) collects the set of newly evaluated Boolean variables $X_{(u,v)}$ that are changed to "false", and (2) finds all the sites S_j following edges (S_j, S_i) in G_d^i that are annotated with v (lines 1-4). It then sends the updated truth values of $X_{(u,v)}$ to such S_j (lines 5-6), which trigger the next round of local evaluation.

Example 7: Continuing with Example 6, upon receiving a message of updated Boolean variables, IEval and IMsg are invoked at each site *in parallel* to find local matches based on the truth values of updated variables. For example, IEval is invoked at site S_1 to reevaluate $X_{(\mathsf{YF},yf_1)}$ and $X_{(\mathsf{SP},sp_1)}$. It finds that all the Boolean variables for each in-node at all sites are true, *i.e.,* no variable is updated to false. Hence no message needs to be sent (line 2). The updated vectors before and after the communications are shown below.

F_i	node	1st Round Partial Evaluation	Result
F_1	yb_1	$X_{(\mathsf{YB},yb_1)}$ = false	$X_{(\mathsf{YB},yb_1)}$ = false
	f_1	$X_{(\mathsf{F},f_1)}$ = false	$X_{(\mathsf{F},f_1)}$ = false
	f_4	$X_{(\mathsf{F},f_4)} = X_{(\mathsf{YF},yf_1)}$	$X_{(\mathsf{F},f_4)}$ = true
	f_2	$X_{(\mathsf{F},f_2)} = X_{(\mathsf{SP},sp_1)}$	$X_{(\mathsf{F},f_2)}$ = true
	yf_2	$X_{(\mathsf{YF},yf_2)} = X_{(\mathsf{YF},yf_3)}$	$X_{(\mathsf{YF},yf_2)}$ = true
F_2	f_3	$X_{(\mathsf{F},f_3)} = X_{(\mathsf{YF},yf_3)}$	$X_{(\mathsf{F},f_3)}$ = true
	yb_2	$X_{(\mathsf{YB},yb_2)} = X_{(\mathsf{YF},yf_3)}$	$X_{(\mathsf{YB},yb_2)}$ = true
	sp_2	$X_{(\mathsf{SP},sp_2)} = X_{(\mathsf{YF},yf_3)}$	$X_{(\mathsf{SP},sp_2)}$ = true
	yf_3	$X_{(\mathsf{YF},yf_3)} = X_{(\mathsf{YF},yf_1)}$	$X_{(\mathsf{YF},yf_3)}$ = true
	sp_3	$X_{(\mathsf{SP},sp_3)} = X_{(\mathsf{YF},yf_1)}$	$X_{(\mathsf{SP},sp_3)}$ = true
	sp_1	$X_{(\mathsf{SP},sp_1)} = X_{(\mathsf{YF},yf_2)} \vee X_{(\mathsf{F},f_2)}$	$X_{(\mathsf{SP},sp_1)}$ = true
F_3	yb_3	$X_{(\mathsf{YB},yb_3)} = X_{(\mathsf{YF},yf_1)}$	$X_{(\mathsf{YB},yb_3)}$ = true
	yf_1	$X_{(\mathsf{YF},yf_1)} = X_{(\mathsf{F},f_2)}$	$X_{(\mathsf{YF},yf_1)}$ = true

Finally, all the local matches are sent to S_c, where the complete match relation is assembled. For example, three matches f_1, f_2 and f_3 are identified for node F at S_c. □

Analyses. The correctness of dGPM is warranted as follows. (1) Algorithm dGPM always terminates. Indeed, for any node v in G and node u in Q, once v.rvec[u] is updated from true to false, *it never changes back*; and in each round of communication (Phase (2)), at least one variable v.rvec[u] is updated to false. (2) For any v and u, v matches u iff v.rvec[u] is true. Indeed, dGPM refines v.rvec in the same way as the algorithm of [18], until dGPM terminates.

For performance bounds, one may verify the following.

(1) *Data shipment.* Data shipment is guided by the local dependency graph, which indicates crossing edges (v, v') in

E_f, where v' is both a virtual node in fragment F_i and an in-node in another fragment F_j. The edge is followed only when $v'.\mathsf{rvec}[u]$ is changed to false for some $u \in V_q$, at most $|V_q|$ times. Moreover, each $v'.\mathsf{rvec}[u]$ is changed *at most once*. Hence the *total data shipment* is bounded by $O(|E_f||V_q|)$ in all rounds of communications in the worst case.

(2) *Response time*. In each round of communication, local matching takes at most $t = O((|V_q| + |V_m|)(|E_q| + |E_m|))$ time [11, 18], and there are at most $O(|V_f||V_q|)$ rounds. In the final step, it takes $O(|V_q||\mathcal{F}|)$ time to merge all the matches and check whether every query node has a match from a site. Hence the worst-case response time is in $O((|V_q|+|V_m|)\ (|E_q|+|E_m|)\ |V_q||V_f| + |V_q|\mathcal{F}|)$. In practice, $|\mathcal{F}| \leq |V_f|$, since fragments are typically not isolated, and each fragment yields at least one distinct node in V_f. Hence, the overall time complexity is in $O((|V_q|+|V_m|)\ (|E_q|+|E_m|)\ |V_q||V_f|)$. Moreover, $|Q|$ (*i.e.,* $|V_q|$, $|E_q|$) is typically small, and $|F_m|$ is much smaller than $|G|$ when $|\mathcal{F}|$ is large.

Boolean queries. Algorithm dGPM processes Boolean queries Q by following the same steps (1) and (2) as for data selecting queries. The only difference is that in step (3), S_c simply checks whether each node of Q has a match in any local site. It returns true if so, and false otherwise.

This completes the proof of Theorem 2.

4.2 Optimization Strategies

We next introduce two optimization strategies. The first one reduces unnecessary computation of IEval following the idea of incremental pattern matching [13], upon receiving a message with updated Boolean variables. The second one enables tunable performance of dGPM between data shipment and response time, by allowing a site to send not only evaluated Boolean variables, but also Boolean equations.

Incremental local evaluation. Recall that in Phase (2) of dGPM, when a site S_i receives a message from another site with evaluated values $X_{(u,v)}$ for some virtual node v of S_i, it calls procedure IEval to revise its local matches $Q(F_i)$.

A better idea is to conduct IEval *incrementally*. It only propagates updated truth values, following a "bottom-up" traversal starting from virtual nodes v, and updates the vectors of the "ancestors" of v. When it reaches a node v' with an unchanged vector, it stops the traversal at v'. Finally, if no vector changes in the entire process, IEval sends false to coordinator S_c. Otherwise, it sends true to S_c. It also sends messages with those $X_{(u,v)}$ for all $v \in F_i.I$ that are updated to false, guided by its local dependency graph.

Following [13], one can verify that this incremental version of IEval takes *optimally* $O(|\mathsf{AFF}|)$ time to update all matches, where AFF is the set of changed variables, the "area" that must be visited in response to the changes. This strategy allows us to minimize unnecessary recomputation.

Example 8: Consider Q and G' by removing the edge (f_2, sp_1) from G (Example 6). After partial evaluation, $X_{(\mathsf{F},f_2)}$ is updated to false and sent from S_2 to S_1. Upon receiving $X_{(\mathsf{F},f_2)}$, instead of recomputing all the Boolean formulas, IEval incrementally updates those affected by $X_{(\mathsf{F},f_2)}$ starting from virtual node f_2. It updates $X_{(\mathsf{YF},yf_1)}$ to false, following $X_{(\mathsf{YF},yf_1)} = X_{(\mathsf{F},f_2)}$ (see Example 6). Similarly, $X_{(\mathsf{SP},sp_1)} = X_{(\mathsf{YF},yf_2)} \vee X_{(\mathsf{F},f_2)}$ is reduced to $X_{(\mathsf{SP},sp_1)} = X_{(\mathsf{YF},yf_2)}$. As no new variables can be updated to false, S_1 terminates the local evaluation, and sends the updated $X_{(\mathsf{YF},yf_1)}$ to S_3. □

Tunable message passing strategy. In Phase (2) of dGPM, a site may do nothing but wait for evaluated variables from its children. To reduce the waiting time and hence, improve the overall response time, we introduce a *push operation* that allows a site S_i to send Boolean equations to another site S_j, instead of Boolean variables, such that S_j can "inline" these equations in the equations of its in-nodes, and hence bypass message passing from S_i to S_j.

Push operation. We first extend the local dependency graph G_d^i of S_i by including the edges (S_i, S_k), for all sites S_k having in-nodes as the virtual nodes in S_i. Given G_d^i at site S_i, a push operation does the following. (1) At site S_i, for each in-node v in F_i, it sends the equations in $v.\mathsf{rvec}[u]$ to all the parent sites S_j in G_d^i if $A_d^i(S_j, S_i)$ contains v, *i.e.,* S_j has an unevaluated virtual node as in-node v of S_i. Site S_i also sends all its children sites S_k in G_d^i to S_j that contribute virtual nodes to the evaluation of v. (2) Each parent S_j (resp. child S_k) of S_i then updates its dependency graph by replacing (S_j, S_i) (resp. (S_i, S_k)) with edges (S_j, S_k), for such child sites S_k (resp. parent site S_j) of S_i. Intuitively, this operation outsources part of computation at S_i to S_j, and bypasses the communication via edge (S_j, S_i).

To determine when to perform a push operation, site S_i checks whether a benefit function $B(S_i)$ exceeds a threshold θ. The function $B(S_i)$ is defined as follows:

$$B(S_i) = \frac{|F_i.O'|}{m * |F_i.I'|}$$

where $F_i.I'$ (resp. $F_i.O'$) denotes the number of unevaluated in-nodes (resp. virtual nodes) at S_i, and m denotes the total size of the equations (messages) to be sent. Intuitively, (1) the more unevaluated virtual nodes and the less unevaluated in-nodes at S_i, the longer a parent S_j has to wait for messages from S_i, and hence, the better if S_i ships its local computation to S_j, bypassing S_i; and (2) the less amount of data requires to be sent, the better a push operation is. If $B(S_i) \geq \theta$, procedure IMsg triggers a push operation at S_i.

Remarks. A push operation ships more data in exchange for better waiting time. To strike a balance, we use m in $B(\cdot)$ to suppress the overhead of shipment. While waiting time is the bottleneck in response time (as observed in our experiments; see [3]), $B(\cdot)$ can be adjusted (*e.g.,* to be positively correlated with m) to balance local evaluation time. That is, IMsg outsources more computation via push operations for larger m. Other performance metrics (*e.g.,* site visit times, workload and processing capacity) can also be integrated into $B(\cdot)$ to improve the performance of dGPM.

Our experimental study shows that these two optimization strategies substantially improve the performance. Indeed, dGPM extended with these strategies (also denoted by dGPM) is *20 times* faster than its counterpart without them (denoted as $\mathsf{dGPM}_{\mathsf{NOpt}}$) on average (see Section 6).

5. PARALLEL SCALABLE ALGORITHMS

We next prove Theorem 3 and Corollary 4 by giving distributed graph simulation algorithms for DAGs and trees in Sections 5.1 and 5.2, respectively, with the desired bounds.

5.1 DAG Patterns and Graphs

We start with an algorithm for DAG Q, denoted as dGPM_d. Algorithm dGPM_d reduces the number of messages sent by *scheduling* the shipment of the updated Boolean variables, following the *(topological) ranks* of query nodes in Q.

The rank $r(u)$ of a node u in a DAG Q is defined as follows: (a) $r(u) = 0$ if u has no child; (b) otherwise, $r(u) = \max(r(u')) + 1$ for each child u' of u. Note that $0 \leq r(u) \leq d$, where d is the *diameter* of Q, *i.e.,* the length of the longest shortest path between two nodes in Q. Moreover, $d \leq |E_q|$.

Recall that for each node v in G and u in Q, $X_{(u,v)}$ depends *only* on $X_{(u',v')}$ for children u' and v' of u and v, respectively. Thus, $X_{(u,v)}$ is determined only by $X_{(u',v')}$ if $r(u) \geq r(u')$ when Q is a DAG. This suggests that we ship variables $X_{(u,v)}$ among sites guided by the rank $r(u)$ of u, and send them in a single batch, such that each message to site S_i allows S_i to find the matches for the query nodes that have the same rank $r(u)$. Observe that since $r(u) \leq d$, each site sends at most d batches of messages. This further indicates less number of partial evaluation over all the sites.

Example 9: Consider a DAG query Q'' and a graph G'' depicted in Fig. 5. One may verify that the rank $r(\mathsf{FB}) = 0$ as it has no child in Q''. Similarly, $r(\mathsf{YB}_2) = 1$, $r(\mathsf{SP}) = 2$, $r(\mathsf{YF}) = r(\mathsf{F}) = 3$, and $r(\mathsf{YB}_1) = 4$. Here YB_1 and YB_2 are two distinct query nodes with the same label YB.

Note that G'' does not match Q''. When algorithm dGPM of Section 4 is used to check whether G'' matches Q'', in total 12 messages have to be sent: 2 messages from F_4 to F_7 and F_8, 4 messages from F_7 and F_8 to F_5 and F_6, and 6 messages from F_5 and F_6 to F_4. For example, two messages for variables $X_{(\mathsf{SP},sp_4)}$ and $X_{(\mathsf{SP},sp_5)}$ have to be sent from F_7 to F_5. This further triggers two rounds of (incremental) partial evaluation on F_5. However, the updated variables $X_{(\mathsf{SP},sp_4)}$ and $X_{(\mathsf{SP},sp_5)}$ can be "merged" and sent as a single message from F_7 to F_5, following the rank of SP. Note that this also reduces one round of partial evaluation at F_5. □

Algorithm $\mathsf{dGPM_d}$. Motivated by this observation, we develop $\mathsf{dGPM_d}$, which has the same three phases as in dGPM, but uses a different message passing strategy $\mathsf{IMsg_d}$.

(1) In the first phase, upon receiving a query Q from the coordinator S_c, each site invokes IEval as in dGPM. It then assigns to each Boolean variable $X_{(u,v)}$ the topological order $r(u)$ of the query node u, for each virtual node and in-node v.

(2) For message passing in Phase (2), each site S_i keeps track of the topological ranks of nodes in Q being processed, and lists L_r of updated variables collected from each site. Each list consists of variables $X_{(u,v)}$ with the same rank $r(u)$ for u in Q. Instead of shipping a Boolean variable $X_{(u,v)}$ once evaluated, $\mathsf{IMsg_d}$ *waits* until all the variables in L_r are evaluated, following the ascending order of r. It then sends the evaluated variables in L_r, in a *single* batch, to the parent sites guided by G_d^i, and waits for the next batch of variables (*i.e.,* with rank $r + 1$) to be evaluated. This reduces the numbers of messages without increasing the overall response time.

(3) This refinement and message passing process repeats until no Boolean variable is unevaluated, *i.e.,* query modes of all ranks are checked (no data needs to be shipped when $r = d$). It then informs coordinator S_c for termination.

Example 10: Given Q'' and G'' in Fig. 5, $\mathsf{dGPM_d}$ finds that Q'' cannot match G'' by shipping at most 6 messages as follows. As no variable is associated with FB ($r = 0$), no data shipment is incurred. It then starts with $r = 1$. Following the local dependency graph G_d^4 at site S_4, it ships only $X_{(\mathsf{YB}_2,yb_4)}$ to F_7 and F_8, as two messages (locally evaluated as false). After the local evaluation, the variables associated with SP nodes in F_7 and F_8 are all false, where the local rank $r = 2$ at both sites. Since no more variables are unevaluated for query nodes with topological rank 2 at this time, S_7 and S_8 send these updated values as two messages to F_5 and F_6, for $r = 2$. Sites S_5 and S_6 collect all the variables of query nodes ranked at 3, *i.e.,*YF and F, and send them to S_4, again in two messages. Now at S_4, $X_{(\mathsf{YB}_1,yb_4)}$ is evaluated false. In total 6 messages are sent, as opposed to 12 by $\mathsf{dGPM_t}$. □

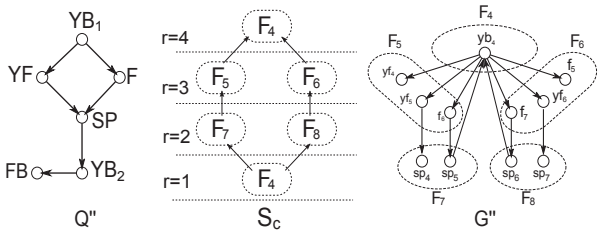


Figure 5: Scheduling data shipment

Analyses. Algorithm $\mathsf{dGPM_d}$ computes $Q(G)$ in $O(d(|V_q|+|V_m|)\,(|E_q|+|E_m|)+|Q||\mathcal{F}|)$ time. To see this, note that (a) it performs d rounds of parallel partial evaluation, (b) each round of evaluation takes $O((|V_q|+|V_m|)\,(|E_q|+|E_m|))$ time, and (c) it takes $O(|Q||\mathcal{F}|)$ time for the coordinator S_c to merge and assembles $Q(G)$. Further, the bound on the total data shipped by dGPM carries over to $\mathsf{dGPM_d}$.

DAG G. Algorithm $\mathsf{dGPM_d}$ also works on acyclic G. To see this, it suffices to consider the following cases. (a) When Q is cyclic, G does not match Q. Indeed, at least one query node in a cycle of Q cannot find a match in G, by the definition of graph simulation. (b) When both Q and G are DAGs, $\mathsf{dGPM_d}$ applies. Hence given a DAG G, all we need to do is to check whether Q is also a DAG (in linear time by using, *e.g.,* Tarjan's algorithm [32]), and if so, apply $\mathsf{dGPM_d}$ to Q and G.

These tell us that for DAGs, $\mathsf{dGPM_d}$ is partition bounded in data shipment, and it is parallel scalable in response time if $|\mathcal{F}|$ is fixed. This completes the proof of Theorem 3.

Remark. Algorithm $\mathsf{dGPM_d}$ sends a bounded number of messages, hence with low communication cost in applications where site-to-site communication is the bottleneck [35]. Moreover, this reduces the total number of partial evaluation at the sites, which further improves the response time.

5.2 Data Graphs as Trees

When G is a distributed tree (with each fragment a connected subtree of G), there exists an algorithm that is parallel scalable in data shipment. We present such an algorithm, denoted as $\mathsf{dGPM_t}$, for data selecting queries.

Algorithm $\mathsf{dGPM_t}$. The algorithm uses *two rounds* of communications between coordinator S_c and each site as follows.

(1) Algorithm $\mathsf{dGPM_t}$ posts Q to all sites S_i. Each site S_i invokes procedure IEval to compute the partial answer L_i in parallel, as in Phase (1) of dGPM.

(2) Instead of sending messages following its dependency graph, each site ships the partial answer L_i and the Boolean vector of its "root" to coordinator S_c. Algorithm $\mathsf{dGPM_t}$ puts all L_i's together as a Boolean equation system [10]. It solves the equations and instantiates all Boolean variables in $O(|Q||\mathcal{F}|)$ time by iteratively unifying variables in the equations, following a "bottom-up" computation induced from

the tree fragments, where the variables associated with virtual nodes are connected to the variables of in-node they define. This completes the first round of communication.

(3) The instantiated Boolean variables are sent back to each site, where IEval is invoked again to complete the matching process. After this, each site sends its local matches to S_c, which are assembled at S_c to get answer $Q(G)$, as in dGPM.

Analysis. Observe the following. (1) Each site is visited at most twice by $\mathsf{dGPM_t}$. (2) IEval computes the partial answer at each fragment F_i in $O(|Q||F_i|)$ time. Hence, the total parallel computational cost of $\mathsf{dGPM_t}$ is in $O(|Q||F_m|)$. The total response time, including the time for evaluating Boolean equations, is in $O(|Q||F_m| + |Q||\mathcal{F}|)$. (3) Each fragment has at most a *single* in-node. Hence, $\mathsf{dGPM_t}$ ships at most a single Boolean vector of size $O(|Q|)$ for each fragment, and the total data shipment is in $O(|Q||\mathcal{F}|)$. Note that the linear bound on Boolean equations does not hold when G is a DAG or a cyclic graph, *i.e.,* the idea only works for trees.

Algorithm $\mathsf{dGPM_t}$ extends the idea of partial evaluation of XPath queries on fragmented XML trees [10] to *graph simulation* on distributed trees, as well as its performance bounds. This completes the proof of Corollary 4.

6. EXPERIMENTAL EVALUATION

We next present an experimental study of our distributed algorithms. Using real-life and synthetic graphs, we conducted three sets of experiments to evaluate the efficiency and data shipment of algorithms (1) dGPM, for general pattern queries and data graphs; (2) $\mathsf{dGPM_d}$, for DAG queries or graphs; and (3) the scalability of dGPM over large synthetic graphs. More experimental results are reported in [3].

Experimental setting. We used two real-life graphs.

(1) Real-world graphs. (a) *Yahoo* (`http://webscope.sandbox.yahoo.com/catalog.php?datatype=g`), which has 3M nodes and 15M edges. Each node denotes a Web page with attributes such as domain. An edge from x to y indicates that x links to y. Its size is 1.6GB. (b) *Citation* (`http://www.arnetminer.org/citation/`) has 1.4M nodes and 3M edges, in which nodes represent papers with attributes such as title, authors, year and venue, and edges denote citations. It is a DAG of 628MB.

(2) Synthetic data. We designed a generator to produce synthetic graphs $G = (V, E, L)$, controlled by the numbers of nodes $|V|$ and edges $|E|$, where L is taken from a set Σ of 15 labels. We use $(|V|, |E|)$ to denote the size of G.

In all our tests we used data selecting patterns.

(2) Graph fragmentation. We randomly partitioned G into a set $\mathcal{F}$ of fragments, controlled by $\mathsf{size}(\mathcal{F})$, the average size of the fragments. Unless stated otherwise, the size $|F_m|$ of the largest fragment is $\mathsf{size}(\mathcal{F}) = |G|/|\mathcal{F}|$. To control $|V_f|$ (resp. $|E_f|$), we iteratively "swapped" two nodes in different fragments that maximally reduced $|V_f|$ (resp. $|E_f|$) following [27], until the ratio $|V_f|/|V|$ (resp. $|E_f|/|E|$) reached a threshold. We represent the size of V_f by *the ratio* $|V_f|/|V|$.

(3) Algorithms. We implemented the following algorithms, all in Java: (1) dGPM (Section 4.1); (2) $\mathsf{dGPM_d}$ (Section 5); (3) Match, which first ships all fragments to a site, and then computes $Q(G)$ using a centralized graph simulation algorithm (see Section 3.1); (4) algorithm disHHK of [25]; and (5) $\mathsf{dGPM_{NOpt}}$, a version of dGPM without using incremental evaluation or push operations (Section 4.2), to evaluate the effectiveness of our optimization strategy.

We also developed a message-based algorithm dMes, to simulate the vertex-centric model of Pregel [14, 26]. Upon receiving Q from a coordinator S_c, each site S_i, as a worker, does the following (as a superstep [14]) for each *virtual node* in fragment F_i. (1) It requests the Boolean values from other sites for the variables of its virtual nodes. (2) It performs local evaluation to update all its local variables. (3) If no change happens, it sends a flag to S_c to vote for termination. It collects the matches from all the sites if at a superstep, all the sites vote to "terminate". For a fair comparison, we do not assume message passing for local evaluation.

Machines. We deployed these algorithms on Amazon EC2 General Purpose instances [1]. Each site stored a fragment. Each experiment was run 5 times and the average is reported here. We report the response time (PT) and data shipment (DS) of the algorithms. As $\mathsf{dGPM_{NOpt}}$ has the same data shipment as dGPM, we do not show DS for $\mathsf{dGPM_{NOpt}}$. We do not show DS for Match as it always ships the entire G. We fixed threshold θ for push operations at 0.2.

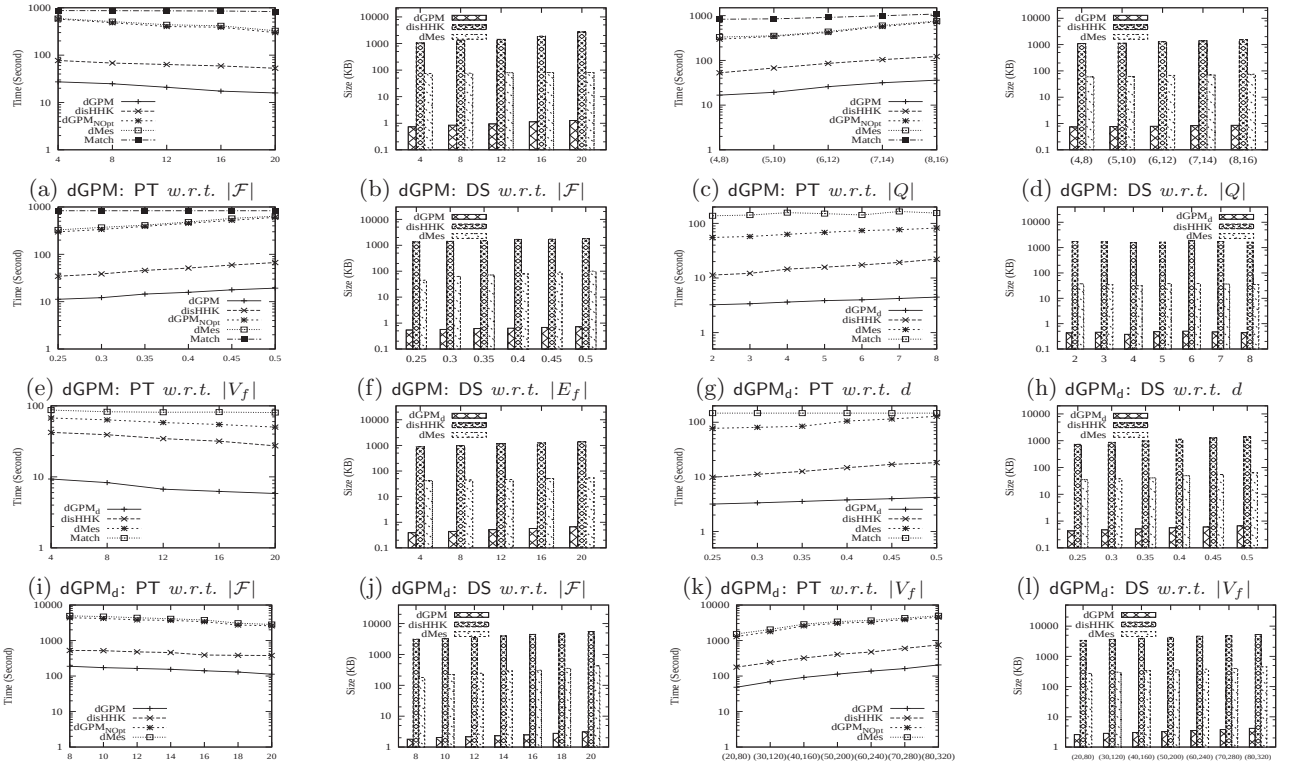
Experimental results. We next report our findings.

Exp-1: Performance of dGPM. We first evaluated the performance of algorithm dGPM for general Q and G, compared with disHHK, Match, dMes and $\mathsf{dGPM_{NOpt}}$, using *Yahoo*. We identified 20 cyclic patterns with conditions such as "domain = '.uk' ", and report the average here. We use *logarithmic scale* for the y-axis in the figures for PT and DS.

Varying $|\mathcal{F}|$. Fixing $|G| = (3M, 15M)$, $|Q| = (5, 10)$ and $|V_f| = 25\%$, we varied the number $|\mathcal{F}|$ of sites from 4 to 20. Figures 6(a) and 6(b) report the PT and DS, respectively. The results verify that dGPM allows a high degree of parallelism: the more processors are available (*i.e.,* the larger $|\mathcal{F}|$ is), the less time dGPM takes. In contrast, Match is indifferent to $|\mathcal{F}|$. Compared to disHHK, dGPM is *3.5 times faster*, and ships *3 orders of magnitude* less data, when $|\mathcal{F}|$ is 20. The improvement over disHHK is *more significant* when $|\mathcal{F}|$ increases: it is 2.8 times faster (resp. ships 0.07% of the data by disHHK) when $|\mathcal{F}| = 4$, and 3.32 times faster when $|\mathcal{F}| = 16$ (resp. ships 0.05% of data). On average, dGPM is *21.6 times faster* than dMes, while the latter ships *80 times* more data, since dMes incurs redundant message passing. We observed slightly increased DS as larger $|\mathcal{F}|$ induces slightly larger $|V_f|$ (resp. $|E_f|$), despite of the adjustment using [27].

Varying $|Q|$. Fixing $|\mathcal{F}| = 8$, $|G| = (3M, 15M)$ and $|V_f| = 25\%$, we varied query size $|Q|$ from $(4, 8)$ to $(8, 16)$. As shown in Figures 6(c) and 6(d), when $|Q|$ gets larger, so do the response time of all these algorithms (the logarithmic scale makes the increase of Match less obvious), and data shipment of all but Match, as expected. Compared to disHHK and dMes, dGPM is 3.6 and 20 times faster, and ships at most 0.044% and 1.5% of their DS, respectively, when $|Q| = (8, 16)$. Moreover, the data shipment of dGPM is much less sensitive to the change of $|Q|$ than the other two.

Varying $|V_f|$. Fixing $|\mathcal{F}| = 8$, $|G| = (3M, 15M)$ and $|Q| = (5, 10)$, we varied $|V_f|$ (resp. $|E_f|$) from 25% to 50%. As shown in Figures 6(e) and 6(f), (1) when $|V_f|$ (resp. $|E_f|$) increases, dGPM takes more time (from 11 to 19 seconds) and ships more data (from 0.54K to 0.97K), as ex-



(a) dGPM: PT *w.r.t.* $|\mathcal{F}|$ (b) dGPM: DS *w.r.t.* $|\mathcal{F}|$ (c) dGPM: PT *w.r.t.* $|Q|$ (d) dGPM: DS *w.r.t.* $|Q|$

(e) dGPM: PT *w.r.t.* $|V_f|$ (f) dGPM: DS *w.r.t.* $|E_f|$ (g) $\mathsf{dGPM_d}$: PT *w.r.t.* d (h) $\mathsf{dGPM_d}$: DS *w.r.t.* d

(i) $\mathsf{dGPM_d}$: PT *w.r.t.* $|\mathcal{F}|$ (j) $\mathsf{dGPM_d}$: DS *w.r.t.* $|\mathcal{F}|$ (k) $\mathsf{dGPM_d}$: PT *w.r.t.* $|V_f|$ (l) $\mathsf{dGPM_d}$: DS *w.r.t.* $|V_f|$

(m) PT *w.r.t.* $|\mathcal{F}|$ (synthetic) (n) DS *w.r.t.* $|\mathcal{F}|$ (synthetic) (o) PT *w.r.t.* $|G|$ (synthetic) (p) DS *w.r.t.* $|G|$ (synthetic)

Figure 6: Performance evaluation

pected. Nonetheless, (2) in all cases dGPM is more efficient than Match and disHHK and moreover, ships less data.

These results also verify *the effectiveness* of our *optimization strategies* (Section 4.2). Indeed, dGPM is *20.3 times faster* than $\mathsf{dGPM_{NOpt}}$ on average. The improvement is more significant over larger fragments: dGPM is 20 times faster than $\mathsf{dGPM_{NOpt}}$ when $|F_m|$ is $(0.15M, 0.75M)$, and is 21.5 times faster when $|F_m|$ increases to $(0.75M, 3.75M)$.

Exp-2: Performance of $\mathsf{dGPM_d}$. The second set of experiments evaluated algorithm $\mathsf{dGPM_d}$, when G is a DAG. We compared $\mathsf{dGPM_d}$ with Match, disHHK and dMes, using *Citation*, which is a DAG. We generated 8 sets of DAG queries $\mathcal{Q}_1, \ldots, \mathcal{Q}_8$ such that all the queries in $\mathcal{Q}_i$ have $d = i + 1$, where d is the diameter of pattern query Q (Section 5).

Varying d. Fixing $|\mathcal{F}| = 8$, $|G| = (1.4\text{M}, 3\text{M})$, $|Q| = (9, 13)$ and $|E_f| = 25\%$, we varied diameter d from 2 to 8. As shown in Figures 6(g) and 6(h), $\mathsf{dGPM_d}$ takes more time when d increases, but its data shipment does not increase. This is consistent with the analysis of Section 5. Despite this, $\mathsf{dGPM_d}$ is much faster and ships much less data than the other three. When $d = 4$, for example, $\mathsf{dGPM_d}$ takes 3.6 seconds on average, 44, 4.1 and 18 times faster than Match, disHHK and dMes, respectively; it ships only 0.024% and 1.06% of data by disHHK and dMes, respectively.

The impact of $|Q|$ on $\mathsf{dGPM_d}$ is consistent with Figures 6(c) and 6(d) for dGPM, and thus is not shown here.

Varying $|\mathcal{F}|$. Using the same $|G|$, $|Q|$ and $|V_f|$ as above and fixing $d = 4$, we varied $|\mathcal{F}|$ from 4 to 20. The results given in Figures 6(i) and 6(d) show that given more processors, $\mathsf{dGPM_d}$ takes less time (from 7.3 seconds with $|\mathcal{F}| = 4$ to 4 seconds for $|\mathcal{F}| = 20$). When $|\mathcal{F}| = 20$, $\mathsf{dGPM_d}$ is 4.7, 12.5 and 15.8 times faster than disHHK, dMes and Match, and ships 2, 3 and 6 orders of magnitude less data, respectively.

Varying $|V_f|$. Fixing $|\mathcal{F}| = 8$, $|Q| = (9, 13)$, $d = 4$ and $|G| = (1.4\text{M}, 3\text{M})$, we varied $|V_f|$ from 25% to 50%. As shown in Fig. 6(k), the PT of $\mathsf{dGPM_d}$ is *insensitive* to $|V_f|$. This experimentally verifies Theorem 3. Contrast this with the 81% increase of the PT of dGPM caused by the same change of $|V_f|$ (Fig. 6(e)). As shown in Fig. 6(l), $\mathsf{dGPM_d}$ ships more data with larger $|V_f|$ (hence larger $|E_f|$), but disHHK and dMes incurs 2144 and 87 times more DS on average, respectively.

Exp-3: Synthetic graphs. Using larger scale synthetic graphs, we evaluated the scalability of dGPM compared with disHHK, dMes and $\mathsf{dGPM_{NOpt}}$, by varying $|\mathcal{F}|$ and $|G|$. The performance of Match is not reported, as it is not capable to cope with large $|G|$ due to memory limit using a single site.

Varying $|\mathcal{F}|$. Fixing $|G| = (30\text{M}, 120\text{M})$, $|Q| = (5, 10)$ and $|V_f| = 20\%$, we varied the number $|\mathcal{F}|$ of sites from 8 to 20. Figures 6(m) and 6(n) verify that dGPM allows a high degree of parallelism: the more processors are available (indicated by $|\mathcal{F}|$), the less time dGPM takes. Compared to disHHK (resp. dMes), dGPM is 3.4 (resp. 23) times faster, and ships 3 (resp. 2) orders of magnitude less data, when $|\mathcal{F}|$ is 20.

Varying $|G|$. Fixing $|\mathcal{F}| = 20$, $|Q| = (5, 10)$, and $|V_f| = 20\%$, we varied $|G|$ from (20M, 80M) to (80M, 320M), *i.e.,* $|F_m|$ from (1M, 4M) to (4M, 16M). As shown in Figures 6(o) and 6(p), the larger $|F_m|$ is, the longer dGPM takes, as expected. While disHHK and dMes are not directly related to $|F_m|$, their response time and data shipment are functions of $|G|$,

and increase when $|G|$ gets larger. Observe that dGPM is 24.7, 3.6 and 27.5 times faster than $\mathsf{dGPM}_{\mathsf{NOpt}}$, disHHK and dMes. It ships at most 0.077% and 0.9% of data shipped by disHHK and dMes on average, respectively.

Summary. We find the following. (1) Our algorithms scale well with large G: their response time and data shipment are *not a function* of $|G|$. (2) They allows a high degree of parallelism: their response time is significantly reduced when more processors are used. For example, dGPM is twice faster when $|\mathcal{F}|$ is increased from 4 to 20. Compared to Match, disHHK and dMes, it is *55.4, 3.5* and *21.6 times faster*, and ships *6, 3* and *2 orders of magnitude less data*, respectively, when $|\mathcal{F}| = 20$. The improvement over other algorithms is even bigger when more processors are used. (3) The algorithms are efficient, *e.g.,* dGPM takes less than 21 seconds when $|G| = (3\text{M}, 15\text{M})$, $|Q| = (5, 10)$ and $|\mathcal{F}| = 12$, and ships only 0.94K data. When Q or G is a DAG, $\mathsf{dGPM}_{\mathsf{d}}$ is 15.8, 4.7 and 12.5 times faster than Match, disHHK and dMes on average, respectively, with orders of magnitude less data shipment. (4) Our optimization strategies are effective, and make dGPM *20 times faster*.

7. CONCLUSION

We have studied what is doable and what is undoable for distributed graph simulation. We have shown that it is *impossible* to find distributed simulation algorithms that are parallel scalable in response time or data shipment. Nonetheless, we have shown that distributed simulation is *partition bounded*, by providing algorithms whose response time and data shipment are *not* a function in the size of graph G. We have also verified, analytically and experimentally, that our algorithms scale well with big G.

One topic for future work is to study parallel scalability and partition boundedness for other graph queries, *e.g.,* graph pattern matching with subgraph isomorphism [33] and strong simulation [24]. Another topic is to give a full treatment of the model advocated in this work by combining partial evaluation and message passing, comparing them with, *e.g.,* MapReduce and GraphLab [22]. In addition, to effectively query real-life graphs, one wants to combine distributed processing with, *e.g.,* graph compression, view-based query processing and top-k query answering.

Acknowledgment. Fan and Wu are supported in part by 973 Programs 2014CB340302 and 2012CB316200, NSFC 61133002, Guangdong Innovative Research Team Program 2011D005, Shenzhen Peacock Program 1105100030834361, and EPSRC EP/J015377/1.

8. REFERENCES

[1] Amazon. *http://aws.amazon.com/ec2/*.
[2] Facebook statistics. *http://www.facebook.com/press/info.php?statistics*.
[3] Full version. *http://www.cs.ucsb.edu/~yinghui/full.pdf*.
[4] Trinity. *research.microsoft.com/en-us/projects/trinity/*.
[5] V. L. Anh and A. Kiss. Efficient processing regular queries in shared-nothing parallel database systems using tree- and structural indexes. In *ADBIS Research Communic*, 2007.
[6] S. Blom and S. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *STTT*, 7(1), 2005.
[7] J. Brynielsson, J. Högberg, L. Kaati, C. Martenson, and P. Svenson. Detecting social positions using simulation. In *ASONAM*, 2010.
[8] A. Buluç and K. Madduri. Graph partitioning for scalable distributed graph computations. In *Graph Partitioning and Graph Clustering*, pages 83–102, 2012.
[9] F. Chung, R. Graham, R. Bhagwan, S. Savage, and G. M. Voelker. Maximizing data locality in distributed systems. *JCSS*, 72(8):1309–1316, 2006.
[10] G. Cong, W. Fan, and A. Kementsietsidis. Distributed query evaluation with performance guarantees. In *SIGMOD*, 2007.
[11] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 3(1), 2010.
[12] W. Fan, X. Wang, and Y. Wu. Performance guarantees for distributed reachability queries. *PVLDB*, 5(11), 2012.
[13] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *TODS*, 38(3), 2013.
[14] A. Fard, M. U. Nisar, L. Ramaswamy, J. A. Miller, and M. Saltz. A distributed vertex-centric approach for pattern matching in massive graphs. In *Big Data*, 2013.
[15] S. Flesca, F. Furfaro, and A. Pugliese. A framework for the partial evaluation of SPARQL queries. In *SUM*, 2008.
[16] J. C. Godskesen and S. Nanz. Mobility models and behavioural equivalence for wireless networks. In *Coordination Models and Languages*, pages 106–122, 2009.
[17] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39, 2008.
[18] M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
[19] M. Jamali. A distributed method for trust-aware recommendation in social networks. *CoRR*, 2010.
[20] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *KDD*, 2006.
[21] J. Leskovec, A. Singh, and J. Kleinberg. Patterns of influence in a recommendation network. In *Advances in Knowledge Discovery and Data Mining*, pages 380–389. 2006.
[22] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB*, 2012.
[23] J. J. Luczkovich, S. P. Borgatti, J. C. Johnson, and M. G. Everett. Defining and measuring trophic role similarity in food webs using regular equivalence. *Journal of Theoretical Biology*, 220(3):303–321, 2003.
[24] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Capturing topology in graph pattern matching. *PVLDB*, 5(4), 2011.
[25] S. Ma, Y. Cao, J. Huai, and T. Wo. Distributed graph pattern matching. In *WWW*, pages 949–958, 2012.
[26] 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 *SIGMOD*, 2010.
[27] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. Technical report, Swedish Institute of Computer Science, 2013.
[28] M. Rowe and O. Group. Interlinking distributed social graphs. In *WWW*, 2009.
[29] M. Shoaran and A. Thomo. Fault-tolerant computation of distributed regular path queries. *TCS*, 410(1):62–77, 2009.
[30] D. Suciu. Distributed query evaluation on semistructured data. *TODS*, 27(1):1–62, 2002.
[31] Y. Tao, W. Lin, and X. Xiao. Minimal MapReduce algorithms. *SIGMOD*, 2013.
[32] R. E. Tarjan. Depth-first search and linear graph algorithms. *SICOMP*, 1(2):146–160, 1972.
[33] J. R. Ullmann. An algorithm for subgraph isomorphism. *JACM*, 23(1):31–42, 1976.
[34] V. Venkataramani et al. TAO: how facebook serves the social graph. In *SIGMOD*, 2012.
[35] D. P. Woodruff and Q. Zhang. When distributed computation is communication expensive. In *DISC*, 2013.
[36] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *VLDB*, 2013.