Distributed Graph Pattern Matching

Shuai Ma  Yang Cao  Jinpeng Huai  Tianyu Wo
State Key Laboratory of Software Development Environment
Beihang University, Beijing 100191, China
{mashuai@act, caoyang@act, huaijp, woty@act}@buaa.edu.cn

ABSTRACT
Graph simulation has been adopted for pattern matching to reduce the complexity and capture the need of novel applications. With the rapid development of the Web and social networks, data is typically distributed over multiple machines. Hence a natural question raised is how to evaluate graph simulation on distributed data. To our knowledge, no such distributed algorithms are in place yet. This paper settles this question by providing evaluation algorithms and optimizations for graph simulation in a distributed setting. (1) We study the impacts of components and data locality on the evaluation of graph simulation. (2) We give an analysis of a large class of distributed algorithms, captured by a message-passing model, for graph simulation. We also identify three complexity measures: visit times, makespan and data shipment, for analyzing the distributed algorithms, and show that these measures are essentially controversial with each other. (3) We propose distributed algorithms and optimization techniques that exploit the properties of graph simulation and the analyses of distributed algorithms. (4) We experimentally verify the effectiveness and efficiency of these algorithms, using both real-life and synthetic data.

Categories and Subject Descriptors
H.2.8 [Database Management]: Database applications—graph data, data mining

Keywords
Graph querying, graph simulation, distributed algorithms

1. INTRODUCTION
Graph pattern matching is being increasingly used in various applications, e.g., software plagiarism detection, protein interaction networks, social networks and intelligence analysis [18, 25, 26]. Graph matching is typically defined in terms of subgraph isomorphism (see, e.g., [15] for a survey). Hence the problem is NP-complete [27]. Furthermore, subgraph isomorphism is often too restrictive to catch sensible matches in emerging applications such as social networks [5, 13].

To reduce the complexity and capture the need of novel applications, graph simulation [16] has been adopted for pattern matching [5, 13]. It is less restrictive than subgraph isomorphism, and can be determined in quadratic time [16]. We say that a graph G matches a pattern Q, via graph simulation.

Figure 1: Pattern query and fragmented data graph simulation, if there exists a binary relation $M \subseteq V_q \times V$, where $V_q$ and $V$ are the two sets of nodes in $Q$ and $G$, respectively, such that (1) for each $(u, v) \in M$, u and v have the same label; and (2) for each node $u \in Q$, there exists $v \in G$ such that $(u, v) \in M$, and (b) for each edge $(u, v) \in Q$, there exists an edge $(v, v') \in G$ having $(u', v') \in M$. Graph simulation (and its extensions) play a critical role for the analysis of social positions/roles in social networks [5, 13, 12].

When evaluating a query on a large dataset, one wants to partition and distribute the data to multiple machines, so that the query can be efficiently evaluated in parallel, as advocated by, e.g., MapReduce [11] and Pregel [21]. Moreover, it is common to find distributed real-life datasets, such as Facebook, Yahoo Flickr, Twitter, and Apple AppStore, that are stored at data centers, which typically host a cluster of tens to thousands of machines [1]. Hence, a natural question raised is how to evaluate graph pattern matching, in terms of graph simulation, on distributed data. To our knowledge, no such distributed algorithms are in place yet.

Example 1: Consider a real-life example for social matching, taken from [24], that recruits people from a social system to set up a new software product. Ideally such a team consists of members with the following roles: (1) project manager (PM), (2) business analyst (BA), (3) software architect (SA), (4) user interface designer (UD), (5) software developer (SD), and (6) software tester (ST). All candidates are stored in a social system in the form of data graph $G_1$ as shown in Fig. 1, where (1) a node is a person labeled with her expertise (with subscription to distinguish one from another), and (2) an edge $(A, B)$ indicates the relationship that $B$ worked well under the supervision of $A$ in previous projects. The relationships of team members are particularly important as the success of the software product heavily relies on their collaboration.

To identify the proper candidates from $G_1$, pattern query $Q_1$ is designed that requires (1) all SA, BA and UD worked well under the same PM, (2) UD worked well under BA such that the user interface could clearly reflect the idea of BA, (3) SD worked well under SA and ST, and (4) ST further worked well under SD as they are mostly related in the development.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
ACM 978-1-4503-1229-5/12/04.
Data graph $G_1$ is partitioned into five fragments: $F_1, \ldots, F_5$ (separated by dotted cycles), and it is distributed over a cluster of five machines, one fragment on each machine. Observe the following. (1) When graph simulation is used, a sensible match containing all nodes in $F_3, F_4$ and $F_5$ are identified. In contrast, when subgraph isomorphism is adopted, it imposes too strict constraints such that no matches can be found. (2) Furthermore, to build a well-organized team, one has to query over all these fragments over different machines to avoid missing any potential candidates.

This highlights the need for developing distributed algorithms for graph simulation. There are a few cheap solutions. (1) A naive one simply collects all parts of a data graph into one machine, and calls a centralized algorithm, e.g., [16]. (2) Another one is to build a MapReduce [11] or Pregel [21] system, and to delegate most of the work to the system. There are, however, several problems for these two solutions. Obviously the first solution does not make use of the distributed facilities at all. Graph simulation needs recursive computations, as illustrated by its recursive definition given before. MapReduce is typically not fit for this kind of graph algorithms, which needs a series of chained MapReduce invocations [7, 21]. Pregel utilizes a message-passing model, which is typically fit for graph algorithms. However, if we simply delegate the computation tasks to Pregel, it may involve too many rounds of computations. As one knows, a good solution for graph simulation must exploit the nature of graph simulation itself.

As observed in [21], graph algorithms often exhibit poor data locality and hence, may incur prohibitive overhead on network traffic. One may verify that to make $G_1$ match $Q_1$, it essentially requires the complete information of the subgraph consisting of fragments $F_3, F_4$ and $F_5$ (even the entire data graph in the worst case). That is, graph simulation has poor data locality (more sophisticated analyses are available in Section 3). This further brings challenge for developing distributed algorithms for graph simulation.

Contributions & Roadmap. To this end, we develop algorithms and optimization techniques for evaluating graph simulation in a distributed setting.

Our contributions can be summarized as follows.

(1) We study fundamental properties of graph simulation (Section 3). We show that connected components in data graph can be treated separately, and the final matches are simply the union of those matches for all single components. We also show that graph simulation has poor data locality. Nevertheless, we identify cases when data locality could be exploited to facilitate the evaluation of graph simulation.

(2) We give an analysis of a large class of distributed algorithms for graph simulation (Section 4). These distributed algorithms are captured by a message-passing model, which is flexible enough to express a broad class of algorithms [19], and is typically fit for the evaluation of graph algorithms [21]. We also identify three complexity measures: visit times, makespan and data shipment, for the analyses of this class of distributed algorithms, and show that these measures are controversial with each other. As efficiency (makespan) remains the dominant factor, we make a decision to sacrifice visit times and data shipment for makespan when designing distributed algorithms.

(3) We propose distributed algorithms which exploit the properties of graph simulation and the analyses of distributed algorithms (Section 5). The algorithms guarantee the following. (a) The total computation cost at all machines is comparable to what is needed by the best-known centralized algorithm [16]. Moreover, the number of rounds of computation is bounded by a constant 4. (b) The total data shipment is at most $|G| + 4|E| + |Q||G| + (k-1)|Q|$, where $|G|$ and $|Q|$ are the sizes of data graph and pattern graph, respectively, and $|E|$ is the total number of nodes with edges across different fragments (boundary nodes), and $k$ is the total number of machines. (c) Each machine except the coordinator is visited at most $g + 2$ times, where $g$ is the maximum number of machines at which a connected component resides. The coordinator is visited with $2(k - 1)$ extra times due to the need for scheduling data shipment and assembling the final result. We also develop effective optimization techniques.

(4) Using both real-life data (Google and Amazon) and synthetic data, we conduct an extensive experimental study (Section 6). We find that our distributed algorithms for graph simulation scale well with large data graphs (e.g., with 10^9 nodes). We also find that our optimization techniques are effective, reducing 1/5 of running time in average.

Related work. There has been a host of work on graph pattern matching, via subgraph isomorphism (e.g., [18, 25, 26]; see [3, 15] for surveys) and via graph simulation [16] and its extensions [13, 12]. Nevertheless, none of these investigates the problem in a distributed fashion.

Distributed query processing has been studied for relational data [17] and XML [8]. There has also been recent work on distributed graph processing to manage large-scale graphs [11, 21]. However, to the best of our knowledge, no previous work has studied distributed computation of graph simulation [16] and its extensions [13, 12]. Close to this work is strong simulation [20], in which the locality property of strong simulation allows us to develop a simple yet effective algorithm to find matches in distributed graphs. In contrast, we show that graph simulation has poor locality, and hence the simple algorithm does not work here.

Message-passing model has been recently adopted for various famous distributed systems, e.g., Pregel [21]. Our distributed algorithms follow this model, which is flexible enough to express a broad class of algorithms [19], and is typically fit for the evaluation of graph algorithms [21].

2. PRELIMINARIES

In this section, we first present basic graph notations. We then introduce the problems of graph pattern matching and its distributed counterpart, in terms of graph simulation.

2.1 Basic Graph Notations

We specify both pattern graphs and data graphs as follows. Let $\Sigma$ be a (possibly infinite) set of labels.

Graphs. A node-labeled directed graph (or simply a graph) is defined as $G(V, E, \lambda)$, where (1) $V$ is a finite set of nodes; (2) $E \subseteq V \times V$ is a finite set of edges, in which $(u, u')$ denotes an edge from nodes $u$ to $u'$; and (3) $\lambda$ is a labeling function that maps each node $u$ in $V$ to a label $\lambda(u)$ in $\Sigma$. The size of $G$, denoted as $|G|$, is the total number of its nodes and edges, i.e., $|V| + |E|$. We also denote $G$ as $(V, E)$ when it is clear from the context.

Intuitively, the function $\lambda$ specifies node attributes, e.g., keywords, blogs, comments, ratings, names, emails, companies [4]; and the label set $\Sigma$ denotes all such attributes.
Input: Pattern graph $Q(V_q, E_q)$ and data graph $G(V, E)$.
Output: The maximum match $M$ in $G$ for $Q$.
1. for each node $u \in V_q$ do $\text{sim}(u) := \{w \mid w \in V, l_G(u) = l_G(u)\}$
2. while there exist $u \in Q, w \in \text{sim}(u)$ and $v \in \text{post}_G(u)$ such that $\text{post}_G(v) \cap \text{sim}(u) = \emptyset$ do
3. $\text{sim}(u) := \text{sim}(u) \setminus \{w\}$
4. $M := \{(v, w) \mid v \in V, w \in \text{sim}(v)\}$
5. return $M$.

Figure 2: Centralized algorithm HHK

Subgraphs. Graph $H(V_s, E_s, hH)$ is a subgraph of graph $G(V, E, h_G)$ if (1) for each node $u \in V_s$, $u \in V$ and $h_H(u) = l_G(u)$, and (2) for each edge $e \in E_s$, $e \in E$. That is, subgraph $H$ only contains a subset of nodes and a subset of edges of $G$. We also denote subgraph $H$ as $G[V_s]$ if $E_s$ is exactly the edges that appear in $G$ over $V_s$.

Descendants. We say that node $v$ is a descendant of node $u$ in $G$ if $v$ is reachable from $u$, i.e., there is a directed path from $u$ to $v$. We use $\text{desc}(G, u)$ to denote the subgraph that contains the set of all descendants of $u$ in $G$, including $u$ itself, and the set of edges in $G$ on those descendants exactly.

2.2 Graph Pattern Matching

We next review the notion of graph simulation [16]. Consider a pattern graph $Q(V_q, E_q)$ and a data graph $G(V, E)$.

A binary relation $R \subseteq V \times V$ is said to be a match if (1) for each $(u, v) \in R$, $u$ and $v$ have the same label, i.e., $l_Q(u) = l_G(v)$; and (2) for each edge $(u, u') \in E_q$, there exists an edge $(v, v')$ in $E$ such that $(u', v') \in R$.

Note that an empty binary relation is a match.

Graph $G$ matches pattern $Q$ via graph simulation, denoted by $Q \prec G$, if there exists a total match relation $M$, i.e., for each $u \in V_q$, there exists $v \in V$ such that $(u, v) \in M$.

Intuitively, simulation preserves the labels and the child relationship of a graph pattern in its match. Simulation was proposed for the analyses of programs [16], and studied for schema extraction from semi-structured data [2]. Simulation and its extensions were recently introduced for social networks [5] and for graph pattern matching [13, 12].

Graph pattern matching. The problem is to find, given any pattern graph $Q$ and data graph $G$, the maximum graph in $G$ for $Q$ if $P \prec G$. It was known that the following result holds [16, 13], by which the problem is well defined.

Proposition 1: Given any pattern graph $Q$ and data graph $G$, there exists a unique maximum match of $Q$ for $G$ no matter whether $Q \prec G$ or not [16, 13].

Algorithm HHK. We next present an algorithm for graph simulation in [16], denoted by HHK and shown in Fig. 2.

For each node $u \in Q$, the set $\text{sim}(u)$ contains candidate nodes in $G$, initially all nodes in $G$ with the same label as $u$ (line 1). By the definition, if $(u, v) \in E_q$ ($v \in \text{post}_G(u)$, successors of $u$), but there exist no nodes $w \in \text{sim}(v)$ such that $(w, w') \in E$ ($w' \in \text{post}_G(v)$), then $w$ cannot be matched to $v$, and hence is removed from $\text{sim}(u)$. This process is repeated until there are no more changes (lines 2–3). Finally, the maximum match is assembled and returned (lines 4–5).

Example 2: Consider pattern graph $Q_1$ and data graph $G_1$ shown in Fig. 1. The maximum match computed by HHK is $\{(PM_3, PA_4), (SA_1, SA_2), (BA_3, BA_4), (UD_1, UD_2), (SD, SD_1), (SD, SD_2), (ST_1, ST_2), (ST, ST_1)\}$. Note that here $PM_1$ cannot match $PM$ since no child of $PM_1$ is labeled with $UD$, and it is similar for the other false matches.

Remark. (1) Graph simulation is computable in quadratic time [16]. Algorithm HHK does not run in quadratic time, but it is simple and easy to understand. Its further refinement leads to a quadratic algorithm [16], the best available algorithm as long as time complexity is concerned [23].

(2) Algorithm HHK correctly computes the maximum match $M$ in data graph $G$ for pattern graph $Q$, and $Q \prec G$ iff for each node $v \in Q$, there exists a node $u \in G$ with $(v, u) \in M$. Hence we focus on computing the maximum match.

2.3 Distributed Graph Pattern Matching

We now introduce graph fragmentation, followed by the problem of distributed graph pattern matching.

We say $\{G[V_1], \ldots, G[V_k]\}$ is a partition of graph $G(V, E)$ if (1) $\bigcup_{i=1}^k V_i = V$ and (2) for any $i \neq j \in [1, k]$, $V_i \cap V_j = \emptyset$.

We also say node $u$ in subgraph $G[V_i]$ ($1 \leq i \leq k$) is a boundary node if there exists an edge $(u, v)$ in $G$ from $u$ to $v$ in $G[V_i]$ such that $i \neq j$ and $1 \leq j \leq k$. To maintain the completeness of $G$, for each boundary node $u$ in a partition $G[V_i]$, we maintain locally a set $B_u$ of labeled nodes $v : j$ such that there exists an edge from $u$ to $v$ in subgraph $G[V_i]$.

Fragmented graph. A fragmented graph $F$ of data graph $G$ is denoted as $(F_1, \ldots, F_5)$, where (1) for each $i \in [1, k]$, $F_i = (G[V_i], B_i)$ is a fragment of $G$ placed at a separate machine $S_i$, (2) $(G[V_1], \ldots, G[V_k])$ is a partition of $G$, and (3) $B_i (i \in [1, k])$ is the union of the sets $B_{i,u}$ of labeled nodes of all boundary nodes $u$ in subgraph $G[V_i]$.

Example 3: Consider the fragmented data graph $G_1$ in Fig. 1 that consists of five fragments $F_1, \ldots, F_5$. Formally, $F_1 = (G[V_1], \{B_{PM_1}, B_{SA_1}\}), F_2 = (G[V_2], \emptyset), F_3 = (G[V_3], \{B_{PM_1}\}), F_4 = (G[V_4], \{B_{SA_1}\}), F_5 = (G[V_5], \emptyset)$, where (1) $V_1 = \{PM_1, BA_1\}$, $V_2 = \{SA_1, ST_1\}$, $V_3 = \{PM_2, BA_2, UD_1\}$, $V_4 = \{SA_2\}$, and $V_5 = \{SD_1, SD_2, SDH, ST_1\}$, respectively; and (2) $B_{PM_1} = \{BA_1 : 2\}, B_{SA_1} = \{SD_1 : 2\}, B_{PM_2} = \{SA_2 : 4\}$ and $B_{SA_1} = \{SD : 5\}$, respectively.

Remark. (1) The fragmented graph has the same number of nodes and edges as the original graph except that the children of boundary nodes are labeled with IDs of the fragments in which they are located. (2) Graph partition is NP-complete in general [14], and is not the focus of this work. Here we allow arbitrary fragmentation of data graphs.

Distributed graph pattern matching. We now define the graph pattern matching problem in a distributed setting.

Given pattern graph $Q$ and fragmented graph $F = (F_1, \ldots, F_5)$ of data graph $G$, in which each fragment $F_i = (G[V_i], B_i)$ ($i \in [1, k]$) is placed at a separate machine $S_i$, the distributed graph pattern matching problem is to find the maximum match in $G$ for $Q$, via graph simulation.

3. PROPERTY OF GRAPH SIMULATION

In this section, we study fundamental properties of graph simulation, which help us design distributed algorithms. We consider pattern graph $Q(V_q, E_q)$ and data graph $G(V, E)$.

3.1 Connected Components

We first show the impacts of connected components (CCs) on the evaluation of graph simulation.

Proposition 2: Let pattern $Q$ consist of $h$ CCs $Q_1, \ldots, Q_h$. For any data graph $G$, if $M_i$ is the maximum match in $G$ for $Q_i$, then $\bigcup_{i=1}^h M_i$ is the maximum match in $G$ for $Q$. □
By Proposition 2, we assume w.l.o.g. that pattern graphs are always connected in the sequel.

**Proposition 3:** Let data graph \( G \) consist of \( h \) CCs \( G_1, \ldots, G_h \). For any pattern \( Q \), if \( M_i \) is the maximum match in \( G_i \) for \( Q \), then \( \bigcup_{i=1}^{h} M_i \) is the maximum match in \( G \) for \( Q \). □

Proposition 3 implies that when \( h \) CCs \( G_i \ (i \in [1, h]) \) are always connected in the sequel, we can simply compute the maximum match \( M \) in \( G_i \) for \( Q \) locally. However, this strategy does not work when \( G \) is across multiple fragments at different machines. We need a better solution.

To do this, we first introduce the following notions.

Consider a binary relation \( R \subseteq V_h \times V \). We use \( R(G) \) to denote the subgraph \( H(V_h, E_h) \) of \( G \), in which (1) \( v \in V_h \) iff there exists \( u \in V_h \) with \((u, v) \in R \), and (2) \((v, v') \in E_h \) iff (i) \((v, v') \in E \) and (ii) there exist \( u, u' \in V_h \) with \((u, v) \in R \), \((u', v') \in R \) and \((u, u') \in E \). We also use \( R(Q) \) to denote the subgraph \( Q[V_h] \) of \( Q \) in which \( u \in V_h \) iff there exists \( v \in V \) with \((u, v) \in R \). Intuitively, \( R(Q) \) and \( R(G) \) are the subgraphs of \( Q \) and \( G \), respectively, that play a role in \( R \).

**Theorem 4:** Consider any binary relation \( R \subseteq V_h \times V \) on pattern graph \( Q(V_h, E_h) \) and data graph \( G(V, E) \) that contains the maximum match \( M \) in \( G \) for \( Q \). If \( M_i \) is the maximum match in \( R(G) \) for \( Q \), then \( \bigcup_{i=1}^{h} M_i \) is exactly \( M \), where \( R(G) \) consists of \( h \) CCs \( R(G_1), \ldots, R(G_h) \). □

By Theorem 4, we can utilize subgraph \( R(G) \), instead of the entire \( G \), to compute the maximum match \( M \) if \( R \) is guaranteed to contain \( M \). Note that even if \( G \) is connected, \( R(G) \) might be highly disconnected, by removing useless nodes and edges from \( G \). This enhances the possibility of treating each CC separately for evaluating graph simulation.

**Remark.** Note that finding all pairwise disconnected components is linear-time equivalent to finding strongly connected components, which is in linear time \([9]\).

### 3.2 Data Locality

We then study the impacts of data locality on graph simulation. For distributed algorithms, one way to maximize parallelization is to explore “what can be computed locally” \([22]\). However, as observed in \([21]\), graph algorithms often exhibit poor locality and hence, may incur prohibitive overhead on network traffic. This is indeed rather challenging for graph simulation, illustrated by an example below.

**Example 4:** Consider pattern \( Q_1 \) and data graph \( G_1 \) of Fig. 1. Let \( G_1 \) be the CC of \( G_1 \) containing node \( PM_2 \). Then to decide whether \( Q_1 \sim G_1 \), we have to ship all subgraphs of \( G_1 \) to a single site to re-assemble \( G_1 \). Indeed, (1) the match graph of \( Q_1 \) and \( G_1 \) is the entire \( G_1 \); and (2) removing any node or edge from \( G_1 \) makes \( Q_1 \not\sim G_1 \). This tells us that graph simulation has poor data locality. □

**Theorem 5:** For any binary relation \( R \subseteq V_h \times V \) on pattern graph \( Q(V_h, E_h) \) and data graph \( G(V, E) \) that contains the maximum match \( M \) in \( G \) for \( Q \),

(1) \((u, v) \in R \) if and only if \((u, v) \in M \),

(2) \((v, v') \in R \) for all \((v, v') \in M \),

and

(1) there exists a cycle in \( R(Q, u) \), and

(2) there exists a cycle in \( R(G, v) \) as well. □

By Theorem 5, whether node \( v \) in \( G \) can be mapped to node \( u \) in \( Q \) can be reduced to the sub-problem of checking whether it belongs to the maximum match in \( desc[R(G, v)] \) for \( desc(R(Q, u)) \), in which \( desc[R(G, v)] \) and \( desc(R(Q, u)) \) are connected subgraphs of \( G \) and \( Q \), respectively. Moreover, a cycle in \( Q \) must match a cycle in \( G \). Indeed, the poor data locality of simulation is caused by the cycles in \( Q \).

**Data locality.** We next formally define data locality that may avoid unnecessary data shipment when evaluating pattern query \( Q \) on data graph \( G \), via graph simulation.

We say that a node \( v \) in data graph \( G \) can be determined locally if checking whether \( v \) matches any node \( u \) in \( Q \) involves only the nodes \( v' \) in \( G \) that have distance \( dist(v, v') \) bounded by a constant factor determined by \( Q \) only. Similarly, we say that data graph \( Q \) can be determined locally if all nodes in \( G \) can be determined locally.

**Theorem 6:** Checking whether node \( v \) in data graph \( G \) matches node \( u \) in pattern graph \( Q \) can be determined locally iff subgraph \( desc(Q, u) \) is a DAG. □

**Example 5:** Consider again pattern graph \( Q_1 \) and data graph \( G_1 \) of Fig. 1. Observe that the cycle \( SD_{ST}/SD \) in \( Q_1 \) matches the cycle \( SD_{ST}/SD_{ST} \) in \( G_1 \). This makes \( G_1 \) impossible to be determined locally since \( h \) can be arbitrarily large and cannot be bounded by \( Q_1 \).

On the contrary, nodes \( BA \) and \( UD \) in \( G_1 \) can be determined locally since the involved nodes in \( G_1 \) have a distance bounded by the longest shortest distance in \( Q_1 \). □

Now let us consider a fragmented graph \( F = (F_1, \ldots, F_h) \) of data graph \( G \), in which each fragment \( F_i = (G[V_i], E_i) \) is placed at a separate machine \( S_i \).

**Corollary 7:** A fragment \( F_i \) can be determined locally if all its boundary nodes can be determined locally. □

**Summary.** (1) We can treat each connect component in a data graph separately when evaluating graph simulation. (2) Theorem 6 tells us how to check whether a node in data graph \( G \) can be determined locally or not. (3) Corollary 7 tells us that the key for distributed pattern matching is to determine the matches of boundary nodes.

### 4. ANALYSES OF DISTRIBUTED ALGORITHMS

In this section, we investigate the complexities of a large class of distributed algorithms for graph simulation, which guides us the design of distributed algorithms. We consider pattern graph \( Q(V_h, E_h) \) and fragmented graph \( F = (F_1, \ldots, F_h) \) of data graph \( G(V, E) \), where each fragment \( F_i = (G[V_i], E_i) \) is placed at a separate machine \( S_i \).

#### 4.1 Model of Computation

We first present the computational model for a large class of distributed algorithms for graph simulation.

We consider distributed algorithms in a pure message-passing (sharing nothing) model. The model consists of a cluster of identical machines, in which one machine can directly send arbitrary number of messages to another one, and those machines co-work with each other by local computations and message-passing. Note that this model is flexible enough to express a broad class of algorithms \([19]\), and is typically fit for the evaluation of graph algorithms \([21]\).

The class of distributed algorithms that we consider work in the following fashion. A user initiates a pattern query \( Q \) at an arbitrary machine, referred to as the coordinator, in the a cluster of \( k \) identical machines. Then query \( Q \) is
(possibly) broadcasted to all the rest machines in the cluster, after which those machines cooperate with each other, through message passing, to compute the maximum match $M$ in data graph $G$ for $Q$, via graph simulation. Finally, the maximum match $M$ is collected and presented to the user at the coordinator. Here the fragmented graph of $G$ is placed at $k$ machines, one fragment at each machine, i.e., the number of fragments is exactly the number of machines. Note that this should not be treated as a restriction as an arbitrary number of fragments can be merged into a single fragment.

### Complexity measures

There are a variety of complexity measures on the performance of distributed algorithms [19], closely related to their computation models. We consider three measures: (1) visit times, the maximum visiting times of a machine in the cluster which indicates the complexity of interactions, (2) makespan, the time cost measuring the completion time, from the time when the query is initialed to the time when the maximum match $M$ is completely assembled, and (3) data shipment, the size of the total messages among distinct machines in the cluster during the computation. Intuitively, one wants to minimize visit times, makespan and data shipment in the same time. As will be seen shortly, these three measures, however, are controversial with each other, which advocates a well-balanced strategy when designing distributed algorithms.

### Specifications

To help analyze the complexity of the class of distributed algorithms, we need to further specify the following: (1) the local information available at each machine, (2) the messages exchanged among machines, and (3) the local computations executed on single machines.

(1) The local information at each machine $S_i$ ($i \in [1, k]$) consists of (a) pattern graph $Q$, (b) subgraph $G_{s,i}$ of data graph $G$ and (c) a marked binary relation $R_i \subseteq V_i \times V$. The pattern graph $Q$ is broadcasted to all machines, and kept unchanged during the computation. The subgraph $G_{s,i}$ is initially the local fragment $F_i = (G[V_i], B_i)$, and is updated once receiving messages containing subgraphs from other machines. For each node pair $(u, v) \in R_i$, it is marked as true, false or unknown, denoting that $(u, v)$ is in the maximum match $M$, not in $M$, or undetermined. Relation $R_i$ can be updated by either messages or local computations.

(2) When machine $S_i$ sends a message to another machine $S_j$, the message only consists of the local information $(Q, G_{s,j}, R_j)$ available at machine $S_j$. Here we do not allow information coding [10] since it is orthogonal to the analysis of the data shipment of distributed algorithms.

(3) At each machine $S_i$, local algorithms that given the local information $(Q, G_{s,i}, R_i)$, compute an updated $R_i$ by utilizing the definition of graph simulation. We require that the local algorithms execute only local computations without involving message-passing during the computation, and they run in time of a polynomial of $|Q|$ and $|G_{s,i}|$. Note that this is to help analyze the makespan problem [28] of distributed algorithms, and should not be treated as a restriction at all.

**Remark.** Under the model and specifications, we can express a large class of distributed algorithms for graph simulation, including all the ones that come into our mind.

### 4.2 Complexities of Distributed Algorithms

We next present our findings on the class of distributed algorithms for evaluating graph simulation queries, which satisfy the computation model and specifications given above.

---

**Proposition 8:** The optimal data shipment of the class of distributed algorithms is $|G| - 1$, and the bound is tight.

Surprisingly, a simple distributed algorithm, referred to as naiveMatch$_{ds}$, can achieve the optimal data shipment. Given pattern graph $Q$ and fragmented data graph $G$, the algorithm simply collects all fragments of the data graph to the coordinator, and then it calls a standard centralized algorithm of graph simulation, e.g., HHK [16], to compute the maximum match in $G$ for $Q$. Note that the total data shipment in the process is bounded by $|G| - 1$ since the subgraph at the coordinator contains at least one node.

**Proposition 9:** The optimal visit times of the class of distributed algorithms are 1, and the bound is tight.

Again a simple distributed algorithm, referred to as naiveMatch$_{ds}$, achieves the optimal visit times. Given pattern graph $Q$ and fragmented data graph $G$, the algorithm visits $k$ machines sequentially, where the last visited one is the coordinator. In the process, each time when the algorithm visits a machine, it collects the local fragment, and sends all the fragments collected so far to the next machine. Finally, all fragments of $G$ reside at the coordinator, and it calls a centralized algorithm of graph simulation, e.g., HHK [16], to compute the maximum match in $G$ for $Q$. Note that each machine is visited one and only once in the process.

While there are efficient optimal algorithms for data shipment and visit times separately, the problem of finding a minimum makespan is much harder, as shown below.

**Proposition 10:** The minimum makespan problem of the class of distributed algorithms is NP-complete [28].

We next present our second set of findings that these three complexity measures are controversial with each other. We illustrate this with the following example.

**Example 6:** Consider pattern graph $Q_o$ and the fragmented data graph $G_o$ in Fig. 3, in which each fragment is a CC without any boundary nodes. Here nodes in $Q_o$ are only allowed to match nodes in $G_o$ with the same labels by ignoring their subscripts. We also assume w.l.o.g. that $F_1 = (G[V_1], \emptyset)$ lies at the coordinator.

One can verify the following:

(1) For algorithm naiveMatch$_{ds}$, the (optimal) data shipment is $|G| - 1 = 8h$, while the visit times are $h$.

(2) For algorithm naiveMatch$_{ds}$, the (optimal) visit times are $1$, while the data shipment is $4h(h + 1)$.

Observe that all the computational workload in these algorithms is mostly laid at a single machine, the coordinator. Essentially, no computing power is used in parallel at all.

(3) Ideally, a distributed algorithm would work as follows.

(i) After receiving $Q$ on each machine, a local algorithm is called to compute the local match in the fragment.

**Figure 3:** Example for complexity analyses
Input: Pattern graph $Q$ and fragmented data graph $F = (F_1, \ldots, F_k)$ with $F_i = (G[V_i], B_i)$ placed at machine $S_i$ ($i \in [1, k]$).
Output: The maximum match $M$ in $G$ for $Q$.

Coordinator $S_Q$.
1. send pattern graph $Q$ to all $k$ participating machines;
2. call $dSchedule$ to find an assignment of those CCs;
3. send the assignment to all participating machines;
4. send $M_{h,s}$ from all $k$ machines do
   1. $M := M \cup M_{h,s}$;
   2. return $M$.

Machine $S_i$;
1. $PM_i := \text{localHHK}(Q, F_i)$;
2. let $C_{h,i}$ be the CCs in $PM_i$ containing nodes in $B_i$;
3. let $M_i$ be the matches in $PM_i$ containing nodes in $C_{h,i}$;
4. send $M_i$ and the sizes and boundary nodes of CCs in $C_{h,i}$ to $S_Q$;
5. receive the assignment of data shipment do
   1. $M_{h,i} := \text{refineHHK}(Q, C_i)$;
   2. send $M_{h,i}$ to coordinator $S_Q$;

![Figure 4: Distributed algorithm disHHK](image)

(ii) Here since each fragment is a CC, all local match results are simply sent to and assembled at the coordinator. Recall Proposition 3 in Section 3.

In this way, the computation is maximally parallelized, and the makespan is minimized, accordingly. However, one may verify that the data shipment is $30h$, and the visit times are $h$. In contrast, the optimal data shipment and visit times are $8h$ and 1, respectively.

Summary. We find that data shipment, visit times and makespan of a large class of distributed algorithms for graph simulation are controversial with each other. As efficiency remains the dominant factor, there needs a well-balanced strategy between makespan and the other two measures.

5. DISTRIBUTED EVALUATION

In this section, we present the distributed algorithms that exploit the properties of graph simulation (Section 3) and the analyses of distributed algorithms (Section 4). We consider pattern graph $Q(V_q, E_q)$ and fragmented graph $F = (F_1, \ldots, F_k)$ of data graph $G(V, E)$, where each fragment $F_i = (G[V_i], B_i)$ ($i \in [1, k]$) is placed at a separate machine $S_i$.

The distributed algorithm, referred to as disHHK, follows the computation model and specifications given in Section 4. It is initiated at the coordinator $S_Q$ where the query $Q$ is issued, and it consists of five stages (shown in Fig. 4).

Stage 1: Coordinator $S_Q$ simply broadcasts pattern query $Q$ to all the $k$ participating machines.

Stage 2: The main objective is to (partially) evaluate $Q$ in each fragment at local machines in parallel (case (3)). As a by-product, the local evaluation filters out useless nodes and edges in the fragment, and hence reduces the data shipment. It also breaks a large fragment into smaller CCs, which further reduces the sizes of CCs across different machines.

Stage 3: The objective is to ship those CCs across different machines to single machines (case (1) and case (4)). This involves two important, but controversial, issues: minimizing data shipment and makespan. We provide a solution that (a) both minimizes the makespan with performance guarantees and (b) minimizes the data shipment with heuristics in the same time.

Stage 4: The objective is to compute the maximum matches in those CCs originally across different machines in parallel, by making use of those partial matches on these components computed locally at Stage 1 (case (5)).

Stage 5: Finally, the match results on all machines are sent to and assembled at the coordinator (case (5) and case (2)).

Correctness. The correctness of disHHK can be easily verified by the analyses of graph simulation in Section 3.

Proposition 11: Given any pattern graph $Q$ and fragmented data graph $G$, algorithm disHHK computes the maximum match in $G$ for $Q$. \qed

Performance. The algorithm guarantees the following.

(1) The total computation cost is comparable to the one of the best-known centralized algorithm [16]. And it invokes four rounds of message-passing and local evaluation only.

(2) The total data shipment is at most $|G| + 4|B| + |Q||G| + (k - 1)|Q|$, where $|B|$ is the total number of boundary nodes.

(3) Each machine except coordinator $S_Q$ is visited at most $g + 2$ times, where $g$ is the maximum number of machines at which a CC resides at the end of Stage 2. Coordinator $S_Q$ is visited with $2(k - 1)$ extra times since it needs to schedule the data shipment and assemble the final result.

Remark. (1) We have decided to sacrifice the visit times and data shipment for the benefits of the makespan, a decision based on the analyses of Section 4.

(2) As one may notice, most stages are run in parallel except for case (1) at Stage 3. As will be seen soon, its computation cost is really low, and would not cause a bottleneck.

In what follows, we describe each stage in more detail.

5.1 Local Evaluation of Partial Match

As shown by the analyses of Section 3, special care needs to be paid on boundary nodes. To do this, we introduce a notion of partial match relation. We consider a fragment $F_i = (G[V_i], B_i)$ at machine $S_i$ ($i \in [1, k]$).

Partial match. A binary relation $R \subseteq V_q \times V_i$ is said to be a partial match if (1) for each $(u, v) \in R$, $u$ and $v$ have the same label; and (2) for each edge $(u, u')$ in $E_q$, (a) there exists a node $v' \in B_i$ in $B$, having the same label as $u'$ if $v$ is a boundary node, or (b) there exists an edge $(v, v')$ in $G[V_i]$ such that $(u', v') \in R$, otherwise.

The difference between a partial match and a match given in Section 2 is the latter deals with boundary nodes. When there are no boundary nodes involved, they are equivalent. Note that $(u, v)$ in a partial match might not appear in the maximum match in $G$ for $Q$ since the matches on boundary nodes are checked partially only with their directed neighbors. We illustrate this with an example below.

Example 7: Consider pattern graph $Q_1$ and data graph $G_1$ in Example 1. Pair $SA_i, SA_j$ is in the maximum partial match $PM_1$ in fragment $F_1$ for $Q$. However, it does not belong to the maximum match $M$ in $G$ for $Q$. \qed

Algorithm localHHK. To compute the partial match, we propose algorithm localHHK, a revision of algorithm HHK that further deals with boundary nodes. Given pattern graph $Q$ and fragment $F_i = (G[V_i], B_i)$ ($i \in [1, k]$), it computes the maximum partial match in the fragment for $Q$. Due to space limitations, its detail is omitted here.
It is easy to get the following result, along the same lines as Proposition 1 and algorithm HHK in Section 2.2.

**Corollary 12:** For any pattern graph $Q$ and fragment $F_i$, (1) there is a unique maximum partial match; and (2) algorithm localHHK computes the maximum partial match. □

We next illustrate with an example how local evaluation also filters out useless nodes and edges in data graphs, which reduces data shipment and computations.

**Example 8:** Consider again pattern graph $Q_1$ and fragments $F_1$ and $F_2$ in data graph $G_1$ in Example 1. Recall that the maximum partial matches $PM_1 = \{(SA_1, SA_1)\}$ and $PM_2 = \emptyset$. Hence, instead of the connected component consisting of fragments $F_1$ and $F_2$, only node $SA_1$ in $F_1$ will be considered in the following stages. Thus a plenty of unnecessary nodes and edges are filtered out at this stage. □

### 5.2 Scheduling Data Shipment

After the local evaluation of partial match is done, we have a partial match $PM_i$ on each machine $S_i$ ($i \in [1, k]$). Let $PM$ be $\bigcup_{i=1}^{k} PM_i$, and $M$ be the maximum match in $G$ for $Q$. It is easy to verify that $M \subseteq PM$.

Now let us consider subgraph $PM(G)$ consisting of nodes and edges of $G$ that play a role in $PM$, as defined in Section 3.1. Theorem 4 tells us that those matches in $PM$ involved with those CCs residing at single machines must belong to the maximum match $M$. Due to the poor data quality of graph simulation, those CCs of $PM$ residing at multiple machines need to be gathered into single machines.

A challenging task here is how to schedule the data shipment such that both the data shipment and the makespan are minimized. Note that the visit times have been fixed in the algorithm, and hence are not involved.

To do this, we introduce the following problem.

**The scheduling problem.** Consider subgraph $PM_i(G)$ at machine $S_i$. Each CC of $PM_i(G)$ is identified by its boundary nodes, its size (e.g., the number of nodes and edges) and its location $S_i$. These information of all the CCs involved with boundary nodes are sent to the coordinator, at which those CCs of $PM(G)$ across different machines are merged. Hence, we have a set $\{C_1, \ldots, C_l\}$ of CCs of $PM(G)$. For each $j \in [1, l]$, $C_j$ is associated with $k + 1$ costs: (1) for each $i \in [1, k]$, the data shipment cost $C_j.d_i$ if $C_j$ is shipped to $S_i$; and (2) the computation cost $C_j.c$. Recall that the computation cost of $C_j$ is a polynomial of $|Q|$ and $|C_j|$.

Formally, the **scheduling problem** is defined as follows. Given $l$ CCs $C_1, \ldots, C_l$, and an integer $k$, find an assignment of the CC to $k$ identical machines, so that both the makespan and the total data shipment are minimized.

**Approximation hardness.** To minimize the makespan, the key is to distribute those connected components evenly on those $k$ machines. As the minimum makespan problem is $\text{NP}$-complete (Proposition 10), we focus on approximate solutions here. We first look for chances that minimize both the makespan and the data shipment.

We say that the scheduling problem is approximatable within $(\alpha, \beta)$ if there exists a PTM algorithm such that given any instance of the problem, the algorithm produces a scheduling solution such that the data shipment is bounded by $\alpha$ times of the optimal data shipment and the makespan is bounded by $\beta$ times of the optimal makespan.

**Theorem 13:** The scheduling problem is not approximatable within $(\xi, \max(k-1, 2))$ for any $\xi > 1$. □

**Approximation algorithm.** This motivates us to look for solutions that have performance guarantees for the makespan only. We propose an approximation algorithm $\text{dSchedule}$ that has a heuristic for minimizing the data shipment, and a performance guarantee for the makespan.

**Theorem 14:** Algorithm $\text{dSchedule}$ produces an assignment of the scheduling problem such that the makespan is within a factor $(2 - 1/k)$ of the optimal one. □

We now present the details of $\text{dSchedule}$, shown in Fig. 5. Given connected components $C_1, \ldots, C_l$ and an integer $k$ as inputs, the algorithm finds an assignment for these CCs. It first computes the average computation $avg_c$ of all CCs (line 1). Given a CC $C_j$, the optimal machine $S_j$, to which $C_j$ is assigned is the one on which the largest part of $C_j$ resides, i.e., $C_j.d_{\text{max}}$ is the largest among $\{C_{j, 1}, \ldots, C_{j, k}\}$. If the total load of $C_j$ and its optimal assigned machine together is equal or less than the average computation cost $avg_c$, then $C_j$ is assigned to its optimal machine (lines 2-4). For the rest CCs $C_j$, schedule $C_j$ to the machine that has the least amount of load (lines 5-6).

**Remark.** (1) A heuristic is used to minimize the data shipment, by shipping CCs to their corresponding optimal machines w.r.t. the data shipment (lines 1-4).

(2) A greedy approach (lines 5-6) is adopted to guarantee the performance of the makespan, along the same lines as the one for the standard makespan problem [28].

(3) The algorithm runs in $O(kl)$, and is very efficient. Hence, its evaluation could not cause a bottleneck.

**Example 9:** Consider again pattern graph $Q_1$ and data graph $G_1$ in Example 1. Algorithm $\text{dSchedule}$ finds an assignment for the two connected components $C_1$ consisting of a single node $SA_1$, and $C_2$ consisting of the entire fragments $F_3, F_4$ and $F_5$. In this case, $C_1$ is simply assigned to $S_1$, and $C_2$ is assigned to $S_3$. That is, these CCs are assigned to the machines with optimal data shipment. □

### 5.3 Refining Partial Match

Similar to algorithm localHHK, algorithm $\text{refineHHK}$ is also a revision of algorithm HHK. The key differences between $\text{refineHHK}$ and HHK lie in that (1) partial matches evaluated at stage 1 are treated as initial candidate matches, and (2) we process the matches of boundary nodes first, in which way the total computation cost of $\text{refineHHK}$ and localHHK is comparative to the one of HHK.

### 5.4 Optimization Techniques

We next present optimization techniques for algorithm $\text{dSchedule}$, by means of data locality and query minimization.
Determining boundary matches using data locality. For each partial match \( PM_i \) (\( i \in [1, k] \)) in fragment \( F_i \) for \( Q \), we further determine whether those matches in \( PM_i \) with boundary nodes, computed by localHHK, belong to the maximum match \( M \) in the entire data graph \( G \) for \( Q \). It is based on an application of Theorems 5 and 6 in Section 3. This reduces both computations and data shipments. Consider the matches \((u, v)\) for a boundary node \( v \) in a partial match \( PM_i \) in fragment \( F_i = (G[V_i], B_i) \) for \( Q \).

To determine whether \((u, v)\) belongs to the maximum match \( M \) in \( G \) for \( Q \), it suffices to determine whether for each child \( u' \) of \( u \) in \( Q \), there is a child \( v' \) of \( v \) such that \((u', v')\) is in \( M \). For each child node \( j : v' \) in \( B_v \) of \( B_i \), if \( v' \) matches a child \( u' \) of \( u \) in \( PM_i \), match \((v', u')\) is further checked by lazy evaluation at machine \( S_j \) as follows.

Let \( C_{v'} \) be the connected component of \( PM_i \) such that \( v' \) is in \( C_j \), and let \( PM_{v', v'} \) be the set of matches in \( PM \) that involve nodes in \( C_j \). We have two cases to consider:

**Case 1:** when there are no boundary nodes in \( G_j \).

For this case, we simply check whether node \( u' \) belongs to subgraph \( PM_{v', v'}(Q) \). If the answer is 'yes', then match \((v', u')\) is a true match, i.e., \((v', u')\) belongs to the maximum match \( M \) in \( G \) for \( Q \). Otherwise, it is a false match. The correctness of this approach is guaranteed by Theorem 5.

**Case 2:** when there are boundary nodes in \( C_j \), but subgraph \( desc(Q, u') \) of pattern graph \( Q \) is a DAG.

For this case, we need to check whether all nodes in \( desc(Q, u') \), including \( u' \) itself, matches no boundary nodes in \( C_j \). If the answer is 'yes', then match \((v', u')\) is a true match, i.e., \((v', u')\) belongs to the maximum match \( M \) in \( G \) for \( Q \). Otherwise, it is an unknown match. The correctness of this approach is guaranteed by Theorems 5 and 6.

We next illustrate the benefits of this optimization technique with the following example.

**Example 10:** Consider pattern graph \( Q_1 \) and data graph \( G_1 \) in Fig. 1, and the partial match results in Example 7. One can verify that (1) boundary nodes \( SA_1 \) and \( SA_3 \) can be determined by this optimization technique, while nothing can be done for boundary node \( PD_2 \).

(1) For node \( SA_1 \), its only child \( SD_1 \) is located in fragment \( F_2 \). The partial match \( PM_2 \) is empty. Hence, a false match decision is sent back to machine \( S_1 \), and this further helps determine that \((SA_1, SA_1)\) is a false match.

(2) For node \( SA_2 \), its only child \( SD_1 \) is located in fragment \( F_3 \). The subgraph \( PM_3(G) \) contains no boundary nodes, and \( SD \) belongs to \( PM_3(Q) \). Hence, a true match decision is sent back to machine \( S_4 \), and this further helps determine that \((SA_1, SA_2)\) is a true match.

After these are done, fragment \( F_3 \) is the only part of \( G \) that needs to be further evaluated. To check the matches in \( F_3 \), we simply ship fragment \( F_4 \) to machine \( S_3 \), instead of shipping \( F_3 \) and \( F_4 \) to machine \( S_5 \) as shown in Example 9. That is, our approach could potentially save a large amount of unnecessary data shipments and computations.

**Remark.** (1) This approach exploits the partial match results at other machines, and the checking is simple and efficient. (2) Only a small amount of data shipment is incurred. The only involved data shipment is the triggers of the lazy evaluation and the decisions (true, false, or unknown). Note that the evaluation is done at machine \( S_j \), not \( S_i \). This is why the data shipment incurred is small.

Minimizing pattern graphs. Given pattern graph \( Q \), we compute a minimized pattern graph \( Q_m \) such that for any data graph \( G \), \( G \) matches \( Q \) iff \( G \) matches \( Q_m \), via graph simulation. The algorithm runs in quadratic time, and is taken from \([6]\). Note that \( Q \) is typically small.

We next illustrate the benefits of minimizing pattern graphs with an example below.

**Example 11:** Consider pattern graph \( Q_6 \) in Fig. 3. The minimized equivalent pattern graph \( Q_m \) of \( Q_6 \) is a compact representation of \( Q_m \), by merging (1) nodes \( A_1, A_2 \), (2) nodes \( C_1, C_2 \), and (3) nodes \( D_1, D_2, D_3, D_4 \). It only consists of four nodes and four edges. Hence, \( Q_m \) is much smaller than \( Q_6 \). It is easy to see that both the data shipment and computation cost of evaluating \( Q_m \) on \( G \) are much smaller that of evaluating \( Q_6 \) on \( G \).

We have implemented a version of disHHK that supports these optimizations, referred to as disHHK*. As will be seen in Section 6, disHHK* significantly outperforms disHHK.

6. EXPERIMENTAL STUDY

We next present an experimental study of our algorithms disHHK and disHHK*. Using both real-life and synthetic data, we conducted four sets of tests to evaluate: (1) the makespan, (2) the data shipment, (3) the visit times of our algorithms, and (4) the effectiveness of algorithm localHHK.

**Experimental setting.** We use the following datasets.

**Real-life data.** We used two real-life datasets\(^1\). (a) Google records a Web graph with 875,713 nodes and 5,105,039 edges where nodes are URLs and an edge from URLs \( x \) to \( y \) indicates that there exists a hyperlink from \( x \) to \( y \). (b) Amazon contains a product co-purchasing network with 548,552 nodes and 1,788,725 edges in which nodes are products and an edge from products \( x \) to \( y \) represents that people buy \( y \) with high probability when they buy \( x \).

**Synthetic graph generator.** We adopted the graph-tool library\(^2\) to produce both pattern and data graphs. It is controlled by three parameters: the number \( n \) of nodes, the number \( n^e \) of edges, and the number \( l \) of node labels. Given \( n \), \( a \) and \( l \), the generator produces a graph with \( n \) nodes, \( n^e \) edges, and the nodes are labeled from a set of \( l \) labels.

**Algorithms.** We implemented the following algorithms, all in Python: (1) algorithms disHHK and disHHK*, and (2) optimal algorithms naiveMatch\(_{ds} \) and naiveMatch\(_{ds} \) (Section 4).

The experiments were run on a cluster of 16 machines, all with 2 Intel Xeon E5620 CPUs and 64GB memory, that are connected by kilomega network. Each test was repeated over 5 times and the average is reported here.

**Experimental results.** In all the experiments, we fixed \( l = 200 \), \( k = 16 \), and set \( \alpha (\alpha_0) = 1.20 \) by default. All datasets are partitioned with a hashing function hash(ID) mod \( k \), and distributed over all participating machines. This partition approach has been commonly used in large-scale data process systems, such as MapReduce [11] and Pregel [21].

**Exp-1: Makespan.** In the first set of tests, we evaluated the performance of disHHK, disHHK*, naiveMatch\(_{ds} \), and naiveMatch\(_{ds} \). We did not report naiveMatch\(_{ds} \) here as it is always much slower than naiveMatch\(_{ds} \).

\(^1\) http://snap.stanford.edu/data/index.html
\(^2\) http://projects.skewed.de/graph-tool/
(1) To evaluate the impacts of pattern graphs $Q$, we fixed data graphs $G$, e.g., Google with 875,713 nodes, Amazon with 548,552 nodes and synthetic data with $10^8$ nodes, while varying (a) the number $|V_q|$ of nodes in $Q$ from 3 to 15 and (b) the density $\alpha_q$ of $Q$ from 1.05 to 1.20, respectively. The results are reported in Figures 6(a) and 6(b), respectively.

One can find the following. (a) All these algorithms scale well with $V_q$ and $\alpha_q$ on large data graphs, except naiveMatch$_{ds}$. While it took disHHK and disHHK$^+$ less than 300s in all cases, it took naiveMatch$_{ds}$ over 500s in all cases, and was much slower than disHHK and disHHK$^+$. (b) disHHK$^+$ is faster than disHHK. Indeed, the running time of disHHK$^+$ is consistently about $[3/4, 4/5]$ of the time taken by disHHK, a significant reduction.

(c) Finally, the elapsed time of all algorithms increases when $|V_q|$ or $\alpha_q$ increases.

(2) To evaluate the impacts of data graphs $G$, we fixed pattern graphs $Q$ with $|V_q| = 9$, while varying the number $|V|$ of nodes of $G$ (Google from $5 \times 10^5$ to $5 \times 10^7$, Amazon from $10^3$ to $10^7$ and synthetic data from $10^2$ to $10^7$), and the density $\alpha$ of $G$ on synthetic data from 1.05 to 1.20, respectively. The results are reported in Figures 6(c) and 6(d), respectively, where $\kappa$ is a constant such that it is $5 \times 10^4, 10^4$ and $10^5$ for Google, Amazon and synthetic data, respectively.

One can find the following. (a) All algorithms scale well on the large data graphs except naiveMatch$_{ds}$, which took over 300s even on the smallest synthetic graphs with $10^7$ nodes. Hence we did not report its running time on synthetic data graphs in Figure 6(c). (b) disHHK$^+$ is consistently faster than disHHK, e.g., it took disHHK 217s on synthetic data graphs with $|V| = 10^5$ while it was only 176s for disHHK$^+$. Indeed, the running time of disHHK$^+$ is consistently about $[3/4, 4/5]$ of the time taken by disHHK, the same as the case above when varying pattern graphs. (c) Finally, the elapsed time of all algorithms increases when $|V|$ or $\alpha$ increases.

(3) To evaluate the impacts of the number $k$ of participating machines, we fixed both data graphs $G$ (with the same setting for data graphs as (1)) and pattern graphs $Q$ (with the setting for pattern graphs as (2)), while varying $k$ from 2 to 16. The results are shown in Figure 6(e).

We find the following. (a) The elapsed time of all algorithms decreases when $k$ increases. (b) The elapsed time of disHHK and disHHK$^+$ decreases faster than the one of naiveMatch$_{ds}$ when $k$ increases. The elapsed time of disHHK on synthetic data was reduced from 1081s with $k = 2$ to 215s with $k = 16$, while the one of naiveMatch$_{ds}$ was only reduced from 1603s with $k = 2$ to 1521s with $k = 16$. And (c) the running time of disHHK$^+$ is consistently about $[3/4, 4/5]$ of the time taken by disHHK, the same as the cases when varying pattern or data graphs.

**Exp-2: Data shipments.** In the second set of tests, using the same setting as Exp-1, we evaluated the total data shipments of disHHK, disHHK$^+$, naiveMatch$_{ds}$ and naiveMatch$_{ct}$. We did not report naiveMatch$_{ct}$ here since it always triggered much more data shipments than naiveMatch$_{ds}$.

(1) We tested the impacts of $Q$ using the same setting as Exp-1(1). The results are reported in Figs. 6(f) and 6(g).

(a) The total data shipments of all algorithms are not sensitive to the size of pattern graphs, (b) although naiveMatch$_{ds}$ achieves the theoretical optimal data shipment, disHHK and disHHK$^+$ trigger similar amount of data shipments as naiveMatch$_{ds}$, and (c) they even trigger less data shipments than naiveMatch$_{ds}$ on large and sparse data graphs, e.g., when $|V_q| > 11$ on synthetic data or $\alpha_q \leq 1.125$ on Amazon.

(2) We tested the impacts of $G$ using the same setting as Exp-1(2). The results are reported in Figs. 6(h) and 6(i).

(a) It is obvious that the total data shipments of all algorithms increase while $|V|$ or $\alpha$ increases. (b) disHHK and disHHK$^+$ shipped less data than naiveMatch$_{ds}$ on large and sparse data graphs, e.g., when $\alpha \leq 1.125$ on synthetic data. And (c) disHHK$^+$ always shipped less data than disHHK.

(3) We tested the impacts of $k$ using the same setting as Exp-1(3). The results are reported in Figure 6(j).

(a) The total data shipments of all algorithms increase when $k$ increases. This is obvious since there are more boundary nodes when $k$ increases when fixing data graphs. And (b)
disHHK and disHHK\textsuperscript{+} again trigger similar amount of data shipments as naiveMatch\textsubscript{ds}.

Exp-3: Visit times. In the third set of tests, using the same setting as Exp-1, we evaluated total visits times of all algorithms. We did not report naiveMatch\textsubscript{st} here as it is always equal to the number \( k \) of participating machines. The impacts of \( Q, G \) and \( k \) are reported in Figs. 6(k) and 6(l), Figs. 6(m) and 6(n), and Figure 6(o), respectively. (a) The total visit times of all algorithms decrease when \(|V_p|\) or \( a_k \) increases. This is because when the size of pattern graphs increases, there are less matches in data graphs. (b) The visit times of all algorithms obviously increase when \(|V|\), \( \alpha \) or \( k \) increases. (c) disHHK and disHHK\textsuperscript{+} have more visit times than naiveMatch\textsubscript{ds}, and disHHK\textsuperscript{+} has \([30\%, 53\%]\) more visit times than disHHK, as expected. Indeed, this is a price that has to be paid in exchange for efficiency.

Exp-4: Effectiveness of localHHK. Adopting the same setting as Exp-1, we also evaluated the effectiveness of localHHK, measured by the number of boundary nodes that are filtered out by localHHK. The results are shown below:

<table>
<thead>
<tr>
<th># of total boundary nodes</th>
<th>Google</th>
<th>Amazon</th>
<th>Synthetic</th>
</tr>
</thead>
<tbody>
<tr>
<td># of filtered boundary nodes</td>
<td>4287</td>
<td>2418</td>
<td>1343952</td>
</tr>
<tr>
<td></td>
<td>1916</td>
<td>1305</td>
<td>446192</td>
</tr>
</tbody>
</table>

It shows that localHHK eliminates a large portion of boundary nodes for disHHK and disHHK\textsuperscript{+}, e.g., it cuts off \(44\%\), \(54\%\) and \(33\%\) boundary nodes on Google, Amazon and synthetic data, respectively. This means that localHHK indeed plays a considerable role in our algorithms.

Summary. From these experiments, we find the following. (1) disHHK and disHHK\textsuperscript{+} are efficient and scale well on large and dense data graphs, and considerably outperform naiveMatch\textsubscript{ds} (optimal data shipment alone) and naiveMatch\textsubscript{st} (optimal visit times alone). (2) Our optimization techniques are effective, reducing the running time by 20\% to 25\%. (3) We have intentionally sacrificed data shipment and visit times for makespan. However, disHHK and disHHK\textsuperscript{+} even ship less data than naiveMatch\textsubscript{ds} when data graphs are large and sparse. Recall that real-life graphs are often large and sparse. disHHK and disHHK\textsuperscript{+} indeed have more visit times than naiveMatch\textsubscript{st}, a price that has to be paid in exchange for improving efficiency and minimizing data shipments. (4) localHHK effectively filters out \([33\%, 54\%]\) unnecessary boundary nodes for disHHK and disHHK\textsuperscript{+}.

7. CONCLUSION
We have proposed evaluation algorithms for graph simulation in a distributed setting. To our knowledge, we are among the first to settle this problem. We have also verified, both analytically and experimentally, the effectiveness of our algorithms and optimization techniques.

Several topics are targeted for future work. First, we are to extend our algorithms to deal with skewed graph partitions. Second, we are experimentally verifying our algorithms using MapReduce and Pregel platforms [11, 21]. Finally, we are to explore indexing techniques and distributed incremental methods to speed up the computation, in response to the dynamic changes of real-life data graphs.

Acknowledgments. Shuai is supported in part by NGFR 973 grant 2011CB302602, NSFC grant 60903149, the Fundamental Research Funds for the Universities, and the Young Faculty Program of MSRA. Tianyu is a contact author.

8. REFERENCES