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 $F_i = (G[V_i], \mathcal{B}_i)$ 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 $\mathcal{B}_v$ of $\mathcal{B}_i$, 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_j such that v' is in C_j, and let $PM_{i,v'}$ be the set of matches in PM_j 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 $PM_{i,v'}(Q)$. 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_2 can be determined by this optimization technique, while nothing can be done for boundary node PM_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, SA_1) is a false match.

(2) For node SA_2, its only child SD_1 is located in fragment F_5. The subgraph $PM_5(G)$ contains no boundary nodes, and SD belongs to $PM_5(Q)$. Hence, a true match decision is sent back to machine S_4, and this further helps determine that (SA, 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_o in Fig.3. The minimized equivalent pattern graph Q_{m_o} of Q_o is a compact representation of Q_o, 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_o} is much smaller than Q_o. It is easy to see that both the data shipment and computation cost of evaluating Q_{m_o} on G_o are much smaller that those of evaluating Q_o on G_o. □

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^α of edges, and the number l of node labels. Given n, α and l, the generator produces a graph with n nodes, n^α 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$_{vt}$ (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 α (α_q) = 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$_{vt}$. We did not report naiveMatch$_{vt}$ 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/

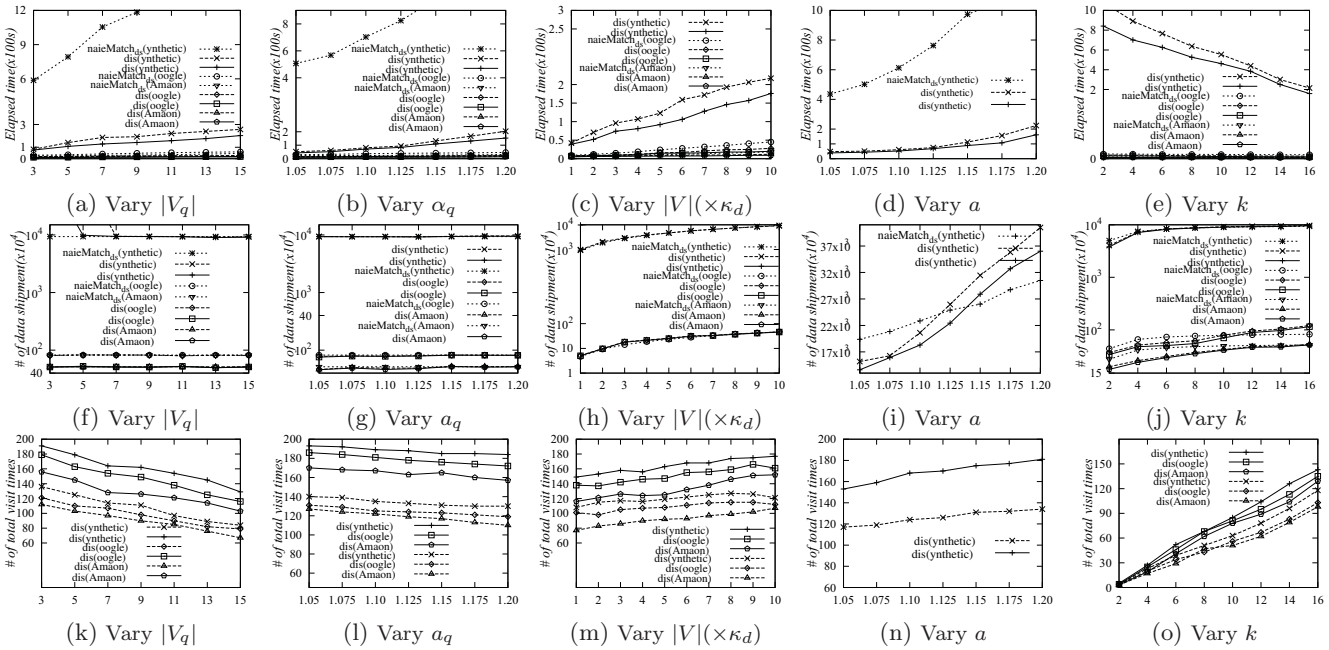

Figure 6: Evaluation on makespan, data shipment and visit times

(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 α_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 α_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 α_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 5×10^4 to 5×10^5, Amazon from 10^3 to 10^4 and synthetic data from 10^7 to 10^8), and the density α 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 κ_d is a constant such that it is 5×10^4, 10^3 and 10^7 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 $|V| = 10^8$ 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 α 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$_{vt}$. We did not report naiveMatch$_{vt}$ 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 α 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$^+$ again trigger similar amount of data shipments as naiveMatch$_{ds}$.

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$_{vt}$ 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 $|V_q|$ or α_q 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 $|V|$, α or k increases. (c) disHHK and disHHK$^+$ have more visit times than naiveMatch$_{ds}$, and disHHK$^+$ 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:

	Google	Amazon	Synthetic
# of total boundary nodes	4287	2418	1343952
# of filtered boundary nodes	1916	1305	446192

It shows that localHHK eliminates a large portion of boundary nodes for disHHK and disHHK$^+$, *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$^+$ are efficient and scale well on large and dense data graphs, and considerably outperform naiveMatch$_{ds}$ (optimal data shipment alone) and naiveMatch$_{vt}$ (optimal visit times alone). (2) Our optimization techniques are effective, reducing the running time by 20% to 25%. (3) We have intensionally sacrificed data shipment and visit times for makespan. However, disHHK and disHHK$^+$ even ship less data than naiveMatch$_{ds}$ when data graphs are large and sparse. Recall that real-life graphs are often large and sparse. disHHK and disHHK$^+$ indeed have more visit times than naiveMatch$_{vs}$, 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$^+$.

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

[1] Data center. *http://wikibon.org/blog/inside-ten-of-the-worlds-largest-data-centers*.
[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.
[3] C. C. Aggarwal and H. Wang. *Managing and Mining Graph Data*. Springer, 2010.
[4] S. Amer-Yahia, M. Benedikt, and P. Bohannon. Challenges in searching online communities. *IEEE Data Eng. Bull.*, 30(2):23–31, 2007.
[5] J. Brynielsson, J. Hogberg, L. Kaati, C. Mårtenson, and P. Svenson. Detecting social positions using simulation. In *ASONAM*, 2010.
[6] D. Bustan and O. Grumberg. Simulation-based minimization. *TOCL*, 4(2), 2003.
[7] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engineering*, 11(4):29–41, 2009.
[8] G. Cong, W. Fan, and A. Kementsietsidis. Distributed query evaluation with performance guarantees. In *SIGMOD*, 2007.
[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.
[10] T. M. Cover and J. A. Thomas. *Elements of information theory (2. ed.)*. Wiley, 2006.
[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[12] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, 2011.
[13] 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.
[14] P.-O. Fjällström. Algorithms for graph partitioning: A survey. *Linkoping Electronic Articles in Computer and Information Science*, 3, 1998.
[15] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS.*, 2006.
[16] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
[17] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.
[18] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *KDD*, 2006.
[19] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
[20] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Capturing topology in graph pattern matching. In *VLDB*, 2012.
[21] 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.
[22] M. Naor and L. J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995.
[23] F. Ranzato and F. Tapparo. An efficient simulation algorithm based on abstract interpretation. *Inf. Comput.*, 208(1):1–22, 2010.
[24] L. G. Terveen and D. W. McDonald. Social matching: A framework and research agenda. In *ACM Trans. Comput.-Hum. Interact.*, pages 401–434, 2005.
[25] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, 2008.
[26] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.
[27] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.
[28] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003.