2015 IEEE Symposium on Security and Privacy

GraphSC: Parallel Secure Computation Made Easy

Kartik Nayak*, Xiao Shaun Wang*, Stratis Ioannidis†, Udi Weinsberg‡, Nina Taft° and Elaine Shi*
*University of Maryland, †Yahoo!, ‡Facebook and °Google

***Abstract*—We propose introducing modern parallel programming paradigms to secure computation, enabling their secure execution on large datasets. To address this challenge, we present GraphSC, a framework that (i) provides a programming paradigm that allows non-cryptography experts to write secure code; (ii) brings parallelism to such secure implementations; and (iii) meets the needs for obliviousness, thereby not leaking any private information. Using GraphSC, developers can efficiently implement an oblivious version of graph-based algorithms (including sophisticated data mining and machine learning algorithms) that execute in parallel with minimal communication overhead. Importantly, our secure version of graph-based algorithms incurs a small logarithmic overhead in comparison with the non-secure parallel version. We build GraphSC and demonstrate, using several algorithms as examples, that secure computation can be brought into the realm of practicality for big data analysis. Our secure matrix factorization implementation can process 1 million ratings in 13 hours, which is a multiple order-of-magnitude improvement over the only other existing attempt, which requires 3 hours to process 16K ratings.**

I. Introduction

Through their interactions with many web services, and numerous apps, users leave behind a dizzying array of data across the web ecosystem. The privacy threats due to the creation and spread of personal data are by now well known. The proliferation of data across the ecosystem is so complex and daunting to users, that encrypting data at all times appears as an attractive approach to privacy. However, this hinders all benefits derived from mining user data, both by online companies and the society at large (e.g., through opinion statistics, ad campaigns, road traffic and disease monitoring, etc). Secure computation allows two or more parties to evaluate any desirable polynomial-time function over their private data, while revealing only the answer and nothing else about each party's data. Although it was first proposed about three decades ago [1], it is only in the last few years that the research community has made enormous progress at improving the efficiency of secure computation [2]–[6]. As such, secure computation offers a better alternative, as it enables data mining while simultaneously protecting user privacy.

The need to analyze data on a massive scale has led to modern architectures that support parallelism, as well as higher level programming abstractions to take advantage of the underlying architecture. Examples include MapReduce [7], Pregel [8], GraphLab [9], and Spark [10]. These provide software developers interfaces handling inputs and parallel data-flow in a relatively intuitive and expressive way. These programming paradigms are also extremely powerful, encompassing a broad class of machine learning, data mining and graph algorithms. Even though these paradigms enable developers to efficiently write and execute complex parallel tasks on very large datasets, they do not support secure computation. Our goal is to bring secure computation to such frameworks in a way that does not require programmers to have cryptographic expertise.

The benefits of integrating secure computation into such frameworks are numerous. The potential to carry out data analysis tasks while simultaneously not leaking private data could change the privacy landscape. Consider a few examples. A very common use of MapReduce is to compute histograms that summarize data. This has been done for all kinds of data, such as counting word frequencies in documents, summarizing online browsing behavior, online medicine purchases, YouTube viewing behavior, and so on, to name just a few. Another common use of the graph parallelization models (e.g., GraphLab) is to compute influence in a social graph through, for example, the PageRank algorithm. Today, joint influence over multiple social graphs belonging to different companies (such as Facebook and LinkedIn), cannot be computed because companies do not share such data. For this to be feasible, the companies need to be able to perform an oblivious secure computation on their joint graph in a highly efficient way that supports their massive datasets and completes in a reasonable time. A privacy requirement for such an application is to ensure that the graph structure, and any associated data, is not leaked; the performance requirements for scalability and efficiency demand the application to be highly parallelizable. A third example application is recommender systems based on the matrix factorization (MF) algorithm. It was shown in [3] that it is possible to carry out secure MF, enabling users to receive recommendations without ever revealing records of past behavior (e.g., movies watched or rated) in the clear to the recommender system. But this previous work did not gracefully incorporate parallelism to scale to millions of records.

This paper addresses the following key question: *can we build an efficient secure computation framework that uses familiar parallelization programming paradigms?* By creating such a framework, we can bring secure computation to the practical realm for modern massive datasets. Furthermore, we can make it accessible to a wide audience of developers that are already familiar with modern parallel programming paradigms, and are not necessarily cryptography experts.

One naïve approach to obtain high parallelization is the following: (a) programmers write programs using a programming language specifically designed for (sequential) secure computation such as the SCVM source language [2] or the ObliVM source language [11]; (b) apply an existing program-to-circuits compiler[1]; and (c) exploit parallelism that occurs at the circuit level – in particular, all the gates within the same layer (circuit depth) can be evaluated in parallel. Henceforth,

†‡°This work was done when the authors were working at Technicolor, Los Altos.

[1]RAM-model compilers such as SCVM [2] and ObliVM [11] effectively compile a program to a sequence of circuits as well. In particular, dynamic memory accesses are compiled into ORAM circuits.

DOI 10.1109/SP.2015.30

we use the term circuit-level parallelism to refer to this baseline approach.

While intuitive, this baseline approach is far from ideal. The circuit derived by a sequential program-to-circuits compiler can also be sequential in nature, and many opportunities to extract parallelism may remain undiscovered. We know from experience, in the insecure environment, that generally trying to produce parallel algorithms requires careful attention. Two approaches have been intensely pursued (for the case of non-secure computation): (a) *Design of parallel algorithms:* an entire cottage industry has focused on designing parallel versions of specific algorithms that seek to express computation tasks with shallow depth and without significantly increasing the total amount of work in comparison with the sequential setting; and (b) *Programming abstractions for parallel computation:* the alternative to finding point solutions for particular algorithms, is to develop programming frameworks that help programmers to easily extract and express parallelism. The frameworks mentioned above fall into this category. These two approaches can also be followed for solutions in secure computation; examples of point solutions include [3], [4]. In this work, we follow the second approach to enable parallel oblivious versions for a range of data mining algorithms.

There are two fundamental challenges to solve our problem. The first is the need to provide a solution that is data oblivious, in order to prevent any information leakage and to prevent unnecessary circuit explosion. The second is that of migrating secure computation models to the parallel environment in an efficient way. Because our solution focuses on graph-based parallel algorithms, we need to ensure that the graph structure itself is not revealed.

In this paper, we focus on 2-party computation in the semi-honest model. Our two parties could be two non-colluding cloud providers (such as Google and Amazon) where both parties have parallel computing architectures (multiple machines with multiple cores). In this case, the data is outsourced to the cloud providers, and within each cloud the secret data could be distributed across multiple machines. In a second scenario, a single cloud provider splits up the data to achieve resilience against insider attacks or APT threats. To realize these, we make the following novel contributions.

A. Our Contributions

We design and implement a parallel secure computation framework called GraphSC. With GraphSC, developers can write programs using programming abstractions similar to Pregel and GraphLab [8], [9], [12]. GraphSC executes the program with a parallel secure computation backend. Adopting this programming abstraction allows GraphSC to naturally support a broad class of data mining algorithms.

New parallel oblivious algorithms. To the best of our knowledge, our work is the *first to design non-trivial parallel oblivious algorithms that outperform generic Oblivious Parallel RAM* [13]. The feasibility of the latter was recently demonstrated by Boyle *et al.* [13]; however, their constructions are of a theoretical nature, with computational costs that would be prohibitive in a practical implementation. Analogously, in the sequential literature, a line of research focuses on designing efficient oblivious algorithms that outperform generic ORAM [14]–[17]. Many of these works focus on specific functionalities of interest. However, such a one-at-a-time approach is unlikely to gain traction in practice, since real-life programmers likely do not possess the expertise to design customized oblivious algorithms for each task at hand; moreover, they *should not* be entrusted to carry out cryptographic design tasks.

While we focus on designing efficient parallel oblivious algorithms, *we take a departure from such a one-at-a-time design approach. Specifically, we design parallel oblivious algorithms for GraphSC's programming abstractions, which in turn captures a broad class of interesting data mining and machine learning tasks*. We will demonstrate this capability for four such algorithms. Moreover, our parallel oblivious algorithms can also be immediately made accessible to non-expert programmers. Our parallel oblivious algorithms achieve logarithmic overhead in comparison with the high poly-logarithmic overhead of generic OPRAM [13]. In particular, *for a graph containing $|\mathsf{E}|$ edges and $|\mathsf{V}|$ vertices, GraphSC just has an overhead of $O(\log |\mathsf{V}|)$ when compared with the parallel insecure version*.

System implementation. ObliVM-GC (http://www.oblivm.com) is a programming language that allows a programmer to write a program that can be compiled into a garbled circuit, so that the programmer need not worry about the underlying cryptographic framework. In this paper, we architect and implement GraphSC, a parallel secure computation framework that supports graph-parallel programming abstractions resembling GraphLab [9]. Such graph-parallel abstractions are expressive and easy-to-program, and have been a popular approach for developing parallel data mining and machine learning algorithms. GraphSC is suitable for both multi-core and cluster-based computing architectures. The source code of GraphSC is available at http://www.oblivm.com.

Evaluation. To evaluate the performance of our design, we implement four classic data analysis algorithms: (1) a histogram function assuming an underlying MapReduce paradigm; (2) PageRank for large graphs; and two versions of matrix factorization, namely, (3) MF using gradient descent, and (4) MF using alternating least squares (ALS). We study numerous metrics, such as how the time scales with input size, with an increasing number of processors, as well as communication costs and accuracy. We deploy our experiments in a realistic setting, both on a controlled testbed and on Amazon Web Services (AWS). We show that we can achieve practical speeds for our 4 example algorithms, and that the performance scales gracefully with input size and the number of processors. We achieve these gains with minimal communication overhead, and an insignificant impact on accuracy. For example, we were able to run matrix factorization on a real-world dataset consisting of 1 million ratings in less than 13 hours on a small 7-machine lab cluster. As far as we know, this is the first application of a complicated secure computation algorithm on large real-world dataset; previous work [3] managed to complete a similar task on only 17K ratings, with no ability to scale beyond a single machine. This demonstrates that our work can bring secure computation into the realm of practical large-scale parallel applications.

The rest of the paper is structured as follows. Following

the related work, in Section II we present GraphSC, our framework for parallel computation on large-scale graphs. In Section III we detail how GraphSC can support parallel *data oblivious* algorithms. Then, in Section IV, we discuss how such parallel oblivious algorithms can be converted into parallel *secure* algorithms. Section V discusses the implementation of GraphSC and detailed evaluation of its performance on several real-world applications. We conclude the paper in Section VI.

B. Model and Terminology

Our main deployment scenario is the following parallel secure two-party computation setting. Consider a client that wishes to outsource computation to two non-colluding, semi-honest cloud providers. Since we adopt Yao's Garbled Circuits [18], one cloud provider acts as the garbler, and the other acts as the evaluator. Each cloud provider can have multiple processors performing the garbling or evaluation.

We adopt the standard security notion of semi-honest model secure computation. The two clouds do not see the client's private data during the course of computation. We assume that the size information $|\mathsf{V}| + |\mathsf{E}|$ is public, where $|\mathsf{V}|$ is the total number of vertices and $|\mathsf{E}|$ is the total number of edges. Not only can the client hide the data from the two cloud providers, it can also hide the computation outcome – simply by masking the computation outcome with a one-time random secret known only to the client.

To keep terminology simple, our main algorithms in Section III-D refers to *parallel oblivious algorithms* – assuming a model where multiple processors have a shared random-access memory. It turns out that once we derive parallel oblivious algorithms, it is easy to translate them into parallel secure computation protocols. Section IV and Figure 5 later in the paper will elaborate on the details of our models and terminology.

C. Related Work

Secure computation has been studied for decades, starting from theory [18]–[22] to implementations [2], [3], [5], [6], [23]–[29].

Parallel secure computation frameworks. Most existing implementations are sequential. However, parallel secure computation has naturally attracted attention due to the wide adoption of multi-core processors and cloud-based compute clusters. Note that in Yao's Garbled Circuits [18], the garbler's garbling operations are trivially parallelizable: garbling is input data independent, and essentially involves evaluating four AES encryptions or hash functions per AND gate using free XOR techniques [30]–[32]. However, evaluation of the garbled circuit must be done layer by layer, and therefore, the depth of the circuit(s) determine the degree to which evaluation can be parallelized.

Most research on parallel secure computation just exploits the natural parallelism within each circuit or in between circuits (for performing cut-and-choose in the malicious model). For example, Husted *et al.* [33] propose using a GPU-based backend for parallelizing garbled circuit generation and evaluation. Their work exploits the natural circuit-level parallelism – however, in cases where the program is inherently sequential (e.g., a narrow and deep circuit), their technique may not be able to exploit massive degrees of parallelism. Our design ensures GraphSC primitives are implemented as low-depth circuits. Though our design currently works on a multi-core processor architecture or a compute cluster, however, conceivably, the same programming abstraction and parallel oblivious algorithms can be directly ported to a GPU-based backend; our work thus is complementary to Husted *et al.* [33].

Kreuter *et al.* [6] exploit parallelism to parallel cut-and-choose in malicious-model secure computation. In particular, cut-and-choose techniques require the garbled evaluation of multiple circuits, such that one can assign each circuit to a different processor. In comparison, we focus on parallelizing the semi-honest model. If we were to move to the malicious model, we would also benefit from the additional parallelism natural in cut-and-choose, like Kreuter *et al.* [6]. Our approach is closest to, and inspired by, the privacy-preserving matrix factorization (MF) framework by Nikolaenko *et al.* [3] that implements gradient-descent MF as a garbled circuit. As in our design, the authors rely on oblivious sorting that, as they note, is parallelizable. Though Nikolaenko *et al.* exploit this to parallelize parts of their MF computation, their overall design is not trivially parallelizable: it results in a $\Omega(|V| + |E|)$-depth circuit, containing serial passes over the data. In fact, the algorithm in [3] is equivalent to the serial algorithm presented in Algorithm 2, restricted to MF. Crucially, beyond extending our implementation to any algorithm expressed by GraphSC, not just gradient-descent MF, our design also parallelizes these serial passes (cf. Figure 4), leading to a circuit of logarithmic depth. Finally, as discussed in Section V, the garbled circuit implementation in [3] can only be run on a single machine, contrary to GraphSC.

Automated frameworks for sequential secure computation. In the sequential setting, numerous automated frameworks for secure computation have been explored, some of which [28], [29] build on (a subset of) a standard language such as C; others define customized languages [2], [23], [24], [26]. As mentioned earlier, the circuits generated by these sequential compilers may not necessarily have low depth. For general-purpose secure computation backends, several protocols have been investigated and implemented, including those based on garbled circuits [1], [18], GMW [34], somewhat or fully homomorphic encryption [35], and others [36], [37]. In this paper, we focus on a garbled circuits backend for the semi-honest setting, but our framework and programming abstractions can readily be extended to other backends as well.

Oblivious RAM and oblivious algorithms. Since Oblivious RAM (ORAM) was initially formulated by Goldreich and Ostrovsky [38], numerous subsequent works [39]–[54] improved their construction, including the new tree-based constructions [51]–[54] that have been widely adoped due to their simplicity and efficiency. Further, efficient oblivious algorithms were studied for specific functionalities [14]–[17], [55], [56] providing point solutions that outperform generic ORAM. As recent works point out [2], Oblivious RAM and oblivious algorithms are key to transforming programs into

compact circuits[2] – and circuits represent the computation model for almost all known secure computation protocols. Broadly speaking, any data oblivious algorithm admits an efficient circuit implementation whose size is proportional to the algorithm's runtime. Generic RAM programs can be compiled into an oblivious counterpart with polylogarithmic blowup [38], [41], [47], [51], [53].

In a similar manner, Oblivious Parallel RAM (OPRAM), proposed by Boyle *et al.*. [13], essentially transforms PRAM programs into low-depth circuits, also incurring a polylogarithmic blowup [13]. As mentioned earlier, their work is more of a theoretical nature and expensive in practice. In comparison, our work proposes efficient oblivious algorithms for a restricted (but sufficiently broad) class of PRAM algorithms, as captured by our GraphSC programming abstractions. As in [13], our design tackles blowups *both* due to obliviousness *and* due to parallelism: our secure, parallel implementation incurs only logarithmic blowup, and is easy to implement in practice.

Parallel programming paradigms. The past decade has given rise to parallelization techniques that are suitable to cheap modern hardware architecture. MapReduce [7] is a seminal work that presented a simple programming model for processing massive datasets on large cluster of commodity computers. This model resulted on a plethora of system-level implementations [58] and improvements [10]. A second advancement was made with Pregel [8], a simple programming model for developing efficient parallel algorithms on large-scale graphs. This also resulted in several implementations, including GraphLab [9], [12] and Giraph [59]. The simplicity of interfaces exposed by these paradigms (like the scatter, gather, and apply operations of Pregel) led to their widespread adoption, as well as to the proliferation of algorithms implemented in these frameworks. We introduce similar programming paradigms to secure computation, in the hope that it can revolutionize the field like it did to non-secure parallel programming models, thus making secure computation easily accessible to non-experts, and easily deployable over large, cheap clusters.

II. GraphSC

In this section, we formally describe GraphSC, our framework for parallel computation. GraphSC is inspired by the scatter-gather operations in GraphLab and Pregel. Several important parallel data mining and machine learning algorithms can be cast in this framework (some of these are discussed in Section V-A); a brief example (namely, the PageRank algorithm) can also be found below. We conclude this section by highlighting the challenges behind implementing GraphSC in a secure fashion.

A. Programming Abstraction

Data-augmented graphs. The GraphSC framework operates on data-augmented directed graphs. A data-augmented directed graph $\mathsf{G}(\mathsf{V},\mathsf{E},\mathsf{D})$ consists of a directed graph $\mathsf{G}(\mathsf{V},\mathsf{E})$, as well as user-defined data on each vertex and each edge denoted

[2]For secure computation, a program is translated into a sequence of circuits whose inputs can be oblivious memory accesses. Note that this is different from transforming a program into a single circuit – for the latter, the best known asymptotical result incurs quadratic overhead [57].

```
Apply(G(V, E, D), f_A)
  for each v in V
    v.data := f_A(v.data)

Scatter(G(V, E, D), f_S, b)
  for each e(u, v) in E
    if b = "in"
      e.data := f_S(e.data, v.data)
    else
      e.data := f_S(e.data, u.data)

Gather(G(V, E, D), ⊕, b)
  for each v in V
    if b = "in"
      v.data := v.data || ⊕_{∀e(u,v)∈E} e.data
    else
      v.data := v.data || ⊕_{∀e(v,u)∈E} e.data
```

Fig. 1: GraphSC semantics.

$\mathsf{D} \in (\{0,1\}^*)^{|\mathsf{V}|+|\mathsf{E}|}$. We use the notation $\mathsf{v.data} \in \{0,1\}^*$ and $\mathsf{e.data} \in \{0,1\}^*$ to denote the data associated with a vertex $\mathsf{v} \in \mathsf{V}$ and an edge $\mathsf{e} \in \mathsf{E}$ respectively.

Programming abstractions. GraphSC follows the Pregel/GraphLab programming paradigm, allowing computations that are "graph-parallel" in nature, i.e., each vertex performs computations on its own data as well as data collected from its neighbors. In broad terms, this is achieved through the following three primitives, which can be thought of as interfaces exposed by the GraphSC abstraction:

1.Scatter: A vertex propagates data to its neighboring edges and updates the edge's data. More specifically, Scatter takes a user-defined function $f_S : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$, and a bit $b \in \{\text{"in"}, \text{"out"}\}$, and updates each directed edge $\mathsf{e}(\mathsf{u},\mathsf{v})$ as follows:

$$\mathsf{e.data} := \begin{cases} f_S(\mathsf{e.data}, \mathsf{v.data}) & \text{if } b = \text{"in"}, \\ f_S(\mathsf{e.data}, \mathsf{u.data}) & \text{if } b = \text{"out"}. \end{cases}$$

Note that the bit b indicates whether the update operation is to occur over incoming or outgoing edges of each vertex.

2.Gather: Through this operation, a vertex aggregates the data from nearby edges and updates its own data. More specifically, Gather takes as input a binary aggregation operator $\oplus : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ and a bit $b \in \{$ "in", "out" $\}$ and updates the data on each vertex $\mathsf{v} \in \mathsf{V}$ as follows:

$$\mathsf{v.data} := \begin{cases} \mathsf{v.data} \;||\; \bigoplus\limits_{\forall \mathsf{e}(\mathsf{u},\mathsf{v}) \in \mathsf{E}} \mathsf{e.data} & \text{if } b = \text{"in"}, \\ \mathsf{v.data} \;||\; \bigoplus\limits_{\forall \mathsf{e}(\mathsf{v},\mathsf{u}) \in \mathsf{E}} \mathsf{e.data} & \text{if } b = \text{"out"}, \end{cases}$$

where $||$ indicates concatenation, and $\bigoplus$ is the iterated binary operation defined by $\oplus$. Hence, at the conclusion of the operation, the vertex stores both its previous value, as well as the output of the aggregation through $\oplus$.

3.Apply: Vertices perform some local computation on their data. More specifically, Apply takes a user-defined function $f_A : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$, and updates every vertex's

data as follows:

$$\mathsf{v.data} := f_A(\mathsf{v.data}).$$

A program abiding by the GraphSC abstraction can thus make arbitrary calls to such Scatter, Gather and Apply operations. Beyond determining this sequence, each invocation of Scatter, Gather, and Apply must also supply the corresponding user-defined functions f_S, f_A, and aggregation operator $\oplus$. Note that the graph structure G does not change during the execution of any of the three GraphSC primitives.

Throughout our analysis, we assume the time complexity of f_S, f_A, and the binary operator $\oplus$ (applied to only 2 arguments) is constant, i.e., it does not depend on the size of G. This is true when, e.g., both vertex and edge data take values in a finite subset of $\{0,1\}^*$, which is the case for all applications we consider[3].

Requirements for the aggregation operator $\oplus$. During the Gather operation, a vertex aggregates data from multiple adjacent edges through a binary aggregation operator $\oplus$. GraphSC requires that this aggregation operator is *commutative* and *associative*, i.e.,

- **Commutative**: For any $a, b \in \mathcal{D}$, $a \oplus b = b \oplus a$.
- **Associative**: For any $a, b, c \in \mathcal{D}$, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.

Roughly speaking, commutativity and associativity guarantee that the result of the aggregation is insensitive to the ordering of the edges.

B. Expressiveness

At a high level, GraphSC borrows its structure from Pregel/GraphLab [8], [9], [12], which is also defined by the three conceptual primitives called Gather, Apply and Scatter. There are however a few differences that are not included in GraphSC, as they break obliviousness. For instance, Pregel allows arbitrary message exchanges between vertices, which is not supported by GraphSC. Pregel also supports modification of the graph structure during computation, whereas GraphSC does not allow such modifications. Finally, GraphLab supports an asynchronous parallel computation of the primitives, whereas GraphSC, and its data oblivious implementation we describe in Section III, are both synchronous.

Despite these differences that are necessary to maintain obliviousness, the expressiveness of GraphSC is the same as that of Pregel/GraphLab. GraphSC encompasses classic graph algorithms like Bellman-Ford, bipartite matching, connected component identification, graph coloring, etc., as well as several important data mining and machine learning operations including PageRank [60], matrix factorization using gradient descent and alternating least squares [61], training neural networks through back propagation [62] or parallel empirical risk minimization through the alternating direction method of multipliers (ADMM) [63]. We review some of these examples in more detail in Section V-A.

[3]Note that, due to the concatenation operation $||$, the memory size of the data at a vertex can in theory increase after repeated consecutive Gather operations. However, in the Pregel/GraphLab paradigm, a Gather is always followed by an Apply, that merges the aggregated edge data with the vertex data through an appropriate user-defined merge operation f_A. Thus, after each iteration completes the vertex memory footprint remains constant.

Algorithm 1 PageRank example

```
1: function computePageRank(G(V, E, D))
2:     f_S(e.data, u.data) : e.data := u.data.PR / u.data.L
3:     ⊕(e_1.data, e_2.data) : e_1.data + e_2.data
4:     f_A(v.data) : v.data.PR := 0.15/|V| + 0.85 × v.data.agg
5:     for i := 1 to K do
6:         Scatter(G, f_S, "out")
7:         Gather(G, ⊕, "in")
8:         Apply(G, f_A)
9:     // Every vertex v stores its PageRank PR
```

C. Example: PageRank

Let us try to understand these primitives using the PageRank algorithm [60] as an example. Recall that PageRank computes a ranking score PR for each vertex u of a graph G through a repeated iteration of the following assignment:

$$\mathsf{PR(u)} = \frac{0.15}{|\mathsf{V}|} + 0.85 \times \sum_{\mathsf{e(v,u)} \in \mathsf{E}} \frac{\mathsf{PR(v)}}{\mathsf{L(v)}}, \forall \mathsf{u} \in \mathsf{V},$$

where $\mathsf{L(v)}$ is the number of outgoing edges. Initially, all vertices are assigned a PageRank of $\frac{1}{|\mathsf{V}|}$.

PageRank can be expressed in GraphSC as shown in Algorithm 1. The data of every vertex v comprises two real values, one for the PageRank (PR) of the vertex and the other for the number of its outgoing edges ($\mathsf{L(v)}$). The data of every edge $\mathsf{e(u,v)}$ comprises a single real value corresponding to the weighted contribution of PageRank of the outgoing vertex u.

For simplicity, we assume that each vertex v has precomputed and stored $\mathsf{L(v)}$ at the beginning of the algorithm's execution. The algorithm then consists of several iterations, each evoking a Scatter, Gather and Apply operation. The Scatter operation updates the edge data $\mathsf{e(u,v)}$ by the weighted PageRank of the outgoing vertex u, i.e., $b =$ "out" and

$$f_S(\mathsf{e.data}, \mathsf{u.data}) : \mathsf{e.data} := \frac{\mathsf{u.data.PR}}{\mathsf{u.data.L}}.$$

In the Gather operation, every vertex v adds up the weighted PageRank over incoming edges $\mathsf{e(u,v)}$ and concatenates the result with the existing vertex data, by storing it in the variable $\mathsf{v.data.agg}$. That is, $b =$ "in", and $\oplus$ is given by

$$\oplus(\mathsf{e}_1\mathsf{.data}, \mathsf{e}_2\mathsf{.data}) : \mathsf{e}_1\mathsf{.data} + \mathsf{e}_2\mathsf{.data}.$$

The Apply operation computes the new PageRank of vertex v using $\mathsf{v.data.agg}$.

$$f_A(\mathsf{v.data}) : \mathsf{v.data.PR} := \frac{0.15}{|\mathsf{V}|} + 0.85 \times \mathsf{v.data.agg}.$$

An example iteration is shown in Figure 2.

D. Parallelization and Challenges in Secure Implementation

Under our standing assumption that f_S, f_A, and $\oplus$ have $O(1)$ time complexity, all three primitives are linear in the input, i.e., can be computed in $O(|\mathsf{V}| + |\mathsf{E}|)$ time. Moreover, like Pregel/GraphLab operations, Scatter, Gather and Apply can be easily parallelized, by assigning each vertex in graph

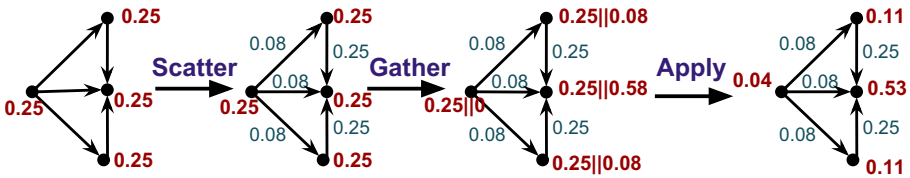


Fig. 2: **One iteration of PageRank computation.** 1. Every page starts with $\mathsf{PR} = 0.25$. 2. During Scatter, outgoing edges are updated with the weighted PageRank of vertices. 3. Vertices then aggregate the data on incoming edges in a Gather operation and store it along with their own data. 4. Finally, vertices update their PageRank in an Apply operation.

G to a different processor. Each vertex also maintains a list of all incoming edges and outgoing edges, along with their associated data. Scatter operations involve transmissions: e.g., in a Scatter "out" operation, a vertex sends its data to all its outgoing neighbors, who update their corresponding incoming edges. Gather operations on the other hand are local: e.g., in a Gather "in", a vertex simply aggregates the data in its incoming edges and appends it to its own data. Both Scatter and Gather operations can thus be executed in parallel across different processors storing the vertices. Finally, in such a configuration, Apply operations are also trivially parallelizable across vertices. Note that, in the presence of $P < |\mathsf{V}|$ processors, to avoid a single overloaded processor becoming a bottleneck, the partitioning of the graph should balance computation and communication across processors.

In this paper, we wish to address the following challenge: we wish to design a secure computation framework implementing GraphSC operations in a privacy-preserving fashion, while maintaining its parallelizability. In particular, our design should be such that, at the implementation of a program using the GraphSC primitives, only the final output of the program is revealed; the input, i.e., the directed data-augmented graph $\mathsf{G}(\mathsf{V}, \mathsf{E}, \mathsf{D})$ should *not* be leaked during the execution of the program. We note that there are several applications in which hiding the data *as well as* the graph structure of G is important. For example, in PageRank, the entire input is described by the graph structure G. As noted in [3], in the case of matrix factorization, the graph structure leaks which items a user has rated, which can again be very revealing. To highlight the difficulties that arise in implementing GraphSC in a secure fashion, we note that clear-text parallelization, as described above, leaks a lot of information. In particular:

1. The amount of data stored by vertices, based on the above partitioning of the graph, reveals information about its neighborhood.
2. The number of times a vertex is accessed during a scatter phase reveals the number of outgoing neighbors.
3. Finally, the neighbors with which each vertex communicates during a Scatter reveal the entire graph G.

These observations illustrate that, beyond the usual issues one faces in converting an algorithm to a secure, data-oblivious implementation, parallelization introduces a considerable new set of challenges. In particular, parallelization in a secure, data oblivious fashion needs to follow a radically different paradigm than the one employed in the clear: the computation and communication at each processor should reveal nothing about G.

III. GraphSC Primitives as Efficient Parallel Oblivious Algorithms

In this section, we discuss how the three primitives exposed by the GraphSC abstraction can be expressed as *parallel data oblivious* algorithms. A parallel oblivious algorithm can be converted to a parallel secure algorithm using standard techniques; we describe such a conversion in more detail in Section IV, focusing here on data-obliviousness and parallelizability.

A. Parallel Oblivious Algorithms: Definitions

In parallel oblivious algorithms, we consider N processors that make oblivious accesses to a shared memory array. Suppose that a parallel algorithm executes in T parallel steps. Then, in every time step $t \in [T]$, each processor $i \in [N]$ makes access to some memory location $\mathsf{addr}_{t,i}$. Let

$$\mathsf{Tr}(\mathsf{G}) := (\mathsf{addr}_{t,i})_{t\in[T],i\in[N]}$$

denote an *ordered* tuple that encodes all the memory accesses made by all processors in all time steps. We refer to $\mathsf{Tr}(\mathsf{G})$ as the memory trace observable by an adversary on input G.

We say that a parallel GraphSC algorithm is *oblivious*, if for any input data-augmented graphs $\mathsf{G} = (\mathsf{V}, \mathsf{E}, \mathsf{D})$ and $\mathsf{G}' = (\mathsf{V}', \mathsf{E}', \mathsf{D}')$ with $|\mathsf{V}| + |\mathsf{E}| = |\mathsf{V}'| + |\mathsf{E}'|$ and $|\mathsf{d}| = |\mathsf{d}'|$ for $\mathsf{d} \in \mathsf{D}$ and $\mathsf{d}' \in \mathsf{D}'$, we have

$$\mathsf{Tr}(\mathsf{G}) = \mathsf{Tr}(\mathsf{G}').$$

In this paper, our parallel oblivious algorithms are all deterministic. Therefore, in the above we require the traces to be identical (as opposed to identically distributed). Note that, by the above definition, a parallel oblivious algorithm *hides both the graph structure and the data on the graph's vertices and edges*. Only the "size" of the graph $|\mathsf{V}| + |\mathsf{E}|$ is revealed. Moreover, such an algorithm can also be represented as a circuit of depth $\Theta(T)$, comprising T layers, each such layer representing the state of the shared memory at time t.

B. Parallel Oblivious Algorithms: Metrics

Total work. One metric of interest is the total work for a parallel oblivious algorithm (i.e., total circuit size, or total number of operations on the shared memory). In comparison with the optimal *sequential, insecure* algorithm for computing the same function, the total work of a parallel oblivious algorithm may increase due to two reasons. First, due to the *cost of parallelism*: the most efficient (insecure) parallel algorithm may incur a blowup in terms of total work. Second,

Algorithm 2 Oblivious GraphSC on a Single Processor

G: list of tuples $\langle \mathsf{u}, \mathsf{v}, \mathsf{isVertex}, \mathsf{data} \rangle$, $M = |\mathsf{V}| + |\mathsf{E}|$

```
function Scatter(G, f_S, b = "out")
    /* b = "in" is similar and omitted */
    sort G by (u, −isVertex)
    for i := 1 to M do    /* Propagate */
        if G[i].isVertex then
            val := G[i].data
        else
            G[i].data := f_S(G[i].data, val)
function Gather(G, ⊕, b = "in")
    /* b = "out" is similar and omitted */
    sort G by (v, isVertex)
    var agg := 1_⊕    // identity w.r.t. ⊕
    for i := 1 to M do    /* Aggregate */
        if G[i].isVertex then
            G[i].data := G[i].data||agg
            agg := 1_⊕
        else
            agg := agg ⊕ G[i].data
function Apply(G, f_A)
    for i := 1 to M do
        G[i].data := f_A(G[i].data)
```

due to the *cost of obliviousness*: requiring that the algorithm is oblivious may also incur additional blowup in total work.

Parallel runtime. Parallel runtime is the total time required to execute the parallel oblivious algorithm, assuming a sufficient number of processors. When the parallel oblivious algorithm is interpreted as a circuit, the parallel runtime is equivalent to the circuit's *depth*. We often compare the parallel runtime of the parallel oblivious algorithm with the optimal *parallel, insecure* baseline of the algorithm computing the same function.

The number of processors needed to achieve parallel runtime corresponds to the maximum *width* of the circuit. If at least P processors are needed to actually achieve the parallel runtime, in the presence of P/c processors, where $c > 1$, the the runtime would be at most $\lceil c \times T \rceil$. Therefore, we can use the parallel runtime metric without sacrificing generality.

C. Single-Processor Oblivious Algorithm

Before presenting our fully-parallel solution, we describe how to implement each of the three primitives defined in Figure 1 in a data-oblivious way on a single processor (i.e., when $P = 1$). One key challenge is how to *hide the graph structure* G during computation.

Alternative graph representation: Our oblivious algorithms require an alternative representation of graphs, that does not disambiguate between edges and vertices. Both vertices and edges are represented as tuples of the form: $\langle \mathsf{u}, \mathsf{v}, \mathsf{isVertex}, \mathsf{data} \rangle$. In particular, each vertex u is represented by the tuple: $\langle \mathsf{u}, \mathsf{u}, 1, \mathsf{data} \rangle$; and each edge (u, v) is represented by the tuple: $\langle \mathsf{u}, \mathsf{v}, 0, \mathsf{data} \rangle$. We represent a graph as a *list of tuples*, i.e., $\mathsf{G} := (\mathsf{t}_i)_{i \in [|\mathsf{V}|+|\mathsf{E}|]}$ where each t_i is of the form $\langle \mathsf{u}, \mathsf{v}, \mathsf{isVertex}, \mathsf{data} \rangle$.

Terminology. For convenience, henceforth, we refer to each edge tuple as a *black cell*, and each vertex tuple as a *white cell* in the list representing graph G.

Algorithm description. We now describe the single-processor oblivious implementation of GraphSC primitives. The formal description of the implementation is provided in Algorithm 2. We also provide an example of the Scatter and Gather operations in Figure 3b, for a very simple graph structure shown in Figure 3a.

Apply. The Apply operation is straightforward to make oblivious under our new graph representation. Essentially, we make a linear scan over the list G. During this scan, we apply the function f_A to each vertex tuple in the list, and a dummy operation to each edge tuple.

Scatter. Without loss of generality, we use $b =$ "out" as an example. The algorithm for $b =$ "in" is similar. The Scatter operation then proceeds in two steps, illustrated in the first three lines of Figure 3b.

Step 1: Oblivious sort: First, perform an oblivious sort on G, so that tuples with the same source vertex are grouped together. Moreover, each vertex should appear before all the edges originating from that vertex.

Step 2: Propagate: Next, in a single linear scan, update the value of each black (i.e., edge) cell with the nearest preceding white cell (i.e., vertex), by applying the f_S function.

Gather. Again, without loss of generality, we will use $b =$ "in" as an example. The algorithm for $b =$ "out" is similar. Gather proceeds in a fashion similar to Scatter in two steps, illustrated in the last three lines of Figure 3b.

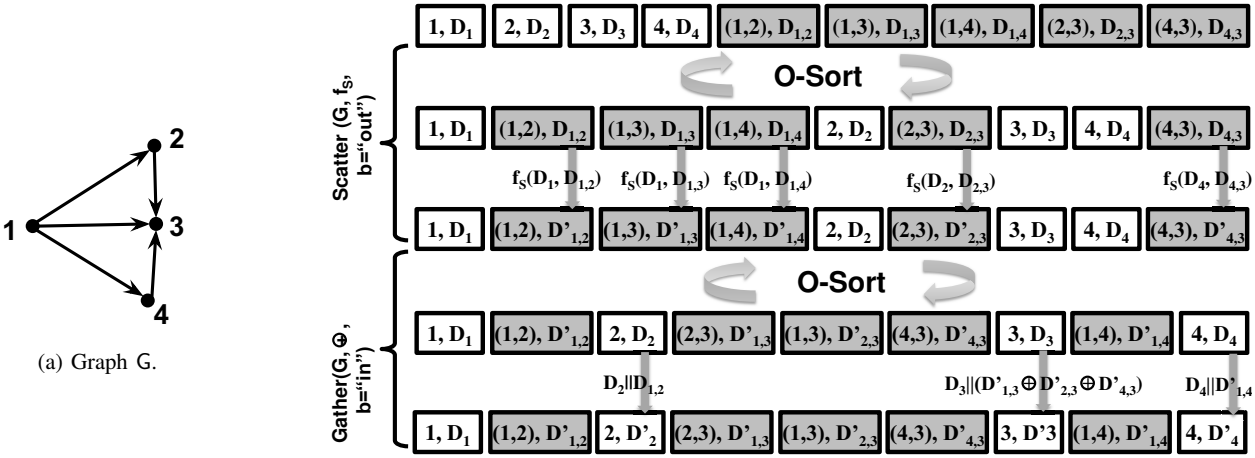
Step 1: Oblivious sort: First, perform an oblivious sort on G, so that tuples with the same destination vertex appear adjacent to each other. Further, each vertex should appear after the list of edges ending at that vertex.

Step 2: Aggregate: Next, in a single linear scan, update the value of each white cell (i.e., vertex) with the $\oplus$-sum of the longest preceding sequence of black cells. In other words, values on all edges ending at some vertex v are now aggregated into the vertex v.

Efficiency. Let $M := |\mathsf{V}| + |\mathsf{E}|$ denote the total number of tuples. Assume that the data on each vertex and edge is of $O(1)$ in length, and hence each f_S, f_A, and $\oplus$ operator is of $O(1)$ cost. Clearly, an Apply operation can be performed in $O(M)$ time. Oblivious sort can be performed in $O(M \log M)$ time using [64], [65] while propagate and aggregate take $O(M)$ time. Therefore, a Scatter and a Gather operation each runs in time $O(M \log M)$.

D. Parallel Oblivious Algorithms for GraphSC

We now describe how to parallelize the sequential oblivious primitives Scatter, Gather, and Apply described in Section III-C. We will describe our parallel algorithms assuming that there are a sufficient number of processors, namely $|\mathsf{V}| + |\mathsf{E}|$ processors. Later in Section III-E, we describe some practical optimizations when the number of processors is smaller than $|\mathsf{V}| + |\mathsf{E}|$.



(a) Graph G.

(b) Transformations of list representing graph G.

Fig. 3: Oblivious Scatter and Gather on a single processor. We apply a Scatter followed by a Gather. *Scatter*: Graph tuples are sorted so that edges are grouped together after the outgoing vertex. e.g. $D_{1,2}, D_{1,3}, D_{1,4}$ are grouped after D_1. Then, in a single pass, all edges are updated. e.g. $D_{1,3}$ is updated as $f_S(D_1, D_{1,3})$. *Gather*: Graph tuples are sorted so that edges are grouped together before the incoming vertex. e.g. $D'_{1,3}, D'_{2,3}, D'_{4,3}$ are grouped before D_3. Then, in a single pass, all vertices compute the aggregate. e.g. $D'_3 = D_3||D'_{1,3} \oplus D'_{2,3} \oplus D'_{4,3}$.

First, observe that the Apply operation can be parallelized trivially. We now demonstrate how to make the Scatter and Gather operations oblivious. Recall that both Scatter and Gather start with an oblivious sort, followed by either an *aggregate* or a *propagate* operation as described in Section III-C. The oblivious sort is a $\log(|\mathsf{V}| + |\mathsf{E}|)$-depth circuit [66], and therefore is trivial to parallelize (by parallelizing directly at the circuit level).

It thus suffices to show how to execute the *aggregate* and *propagate* operations in parallel. To highlight the difficulty behind the parallelization of these operations, recall that in a data-oblivious execution, a processor needs to, e.g., aggregate values by accessing the list representing the graph at fixed locations, which do not depend on the data. However, as seen in Figure 3b, the positions of black (i.e., edge) cells whose values are to be aggregated and stored in white (i.e., vertex) cells clearly depend on the input (namely, the graph G).

Parallelizing the aggregate operation. Recall that an aggregate operation updates the value of each white cell with values of the longest sequence of black cells preceding it. For ease of exposition, we first present a few definitions before presenting our parallel *aggregate* algorithm.

Definition 1. Longest Black Prefix: For $j \in \{1, 2, \ldots, |\mathsf{V}| + |\mathsf{E}|\}$, the longest black prefix before j, denoted $\mathsf{LBP}[1, j)$, is defined to be the longest consecutive sequence of black cells before j, not including j.

Similarly, let $1 \leq i < j \leq |\mathsf{V}| + |\mathsf{E}|$, we use the notation $\mathsf{LBP}[i, j)$ to denote the longest consecutive sequence of black cells before j, constrained to the subarray $\mathsf{G}[i \ldots j)$ (index i being inclusive, and index j being exclusive).

Definition 2. Longest Prefix Sum: Let $1 \leq i < j \leq |\mathsf{V}| + |\mathsf{E}|$, we use the notation $\mathsf{LPS}[i, j)$ to denote the "sum" (with respect to the $\oplus$ operator), of $\mathsf{LBP}[i, j)$.

Abusing notation, we treat $\mathsf{LPS}[i, j)$ is an alias for $\mathsf{LPS}[1, j)$ if $i < 1$. The parallel aggregate algorithm is described in Figure 4. The algorithm proceeds in a total of $\log(|\mathsf{V}| + |\mathsf{E}|)$ time steps. In each intermediate time step τ, a processor $j \in \{1, 2, \ldots, |\mathsf{V}| + |\mathsf{E}|\}$ computes $\mathsf{LPS}[j - 2^\tau, j)$. As a result, at the conclusion of these $\log(|\mathsf{V}| + |\mathsf{E}|)$ steps, each processor j has computed $\mathsf{LPS}[1, j)$.

This way, by time τ, all processors compute the LPS values for all segments of length 2^τ. Now, observe that $\mathsf{LPS}[j - 2^\tau, j)$ can be computed by combining $\mathsf{LPS}[j - 2^\tau, j - 2^{\tau-1})$ and $\mathsf{LPS}[j - 2^{\tau-1}, j)$ in a slightly subtle (but natural) manner as described in Figure 4. Intuitively, at each τ, a segment is aggregated with the immediately preceding segment of equal size *only if* a white cell has not be encountered so far.

At the end of $\log(|\mathsf{V}| + |\mathsf{E}|)$ steps, each processor j whose cell is white, appends its data to the aggregation result $\mathsf{LPS}[1, j)$ – this part is omitted from Figure 4 for simplicity.

Parallelizing the propagate operation. Recall that, in a *propagate* operation, each black cell updates its data with the data of the nearest preceding white cell. The *propagate* operation can be parallelized in a manner similar to *aggregate*. In fact, we can even express a *propagate* operation as a special *aggregate* operation as follows: Initially, every black cell stores *(i)* the value of the preceding white cell if a white cell precedes; and *(ii)* $-\infty$ otherwise. Next, we perform an *aggregate* operation where the $\oplus$ operator is defined to be the $\max$ operator. At the end of $\log |\mathsf{V}| + |\mathsf{E}|$ time steps, each processor has computed $\mathsf{LPS}[1, j)$, i.e., the value of the nearest white cell preceding j. Now if cell $\mathsf{G}[j]$ is black, we can overwrite its data entry with $\mathsf{LPS}[1, j)$.

Operation	Total work				Parallel time		
	Seq. insecure	Par. insecure	Par. oblivious	Blowup	Par. insecure	Par. oblivious	Blowup
Scatter	$O(\|\mathsf{E}\|)$	$O(\|\mathsf{E}\|)$	$O(\|\mathsf{E}\| \log \|\mathsf{V}\|)$	$O(\log \|\mathsf{V}\|)$	$O(1)$	$O(\log \|\mathsf{V}\|)$	$O(\log \|\mathsf{V}\|)$
Gather	$O(\|\mathsf{E}\|)$	$O(\|\mathsf{E}\| \log d_{\max})$	$O(\|\mathsf{E}\| \log \|\mathsf{V}\|)$	$O(\log_{d_{\max}} \|\mathsf{V}\|)$	$O(\log d_{\max})$	$O(\log \|\mathsf{V}\|)$	$O(\log_{d_{\max}} \|\mathsf{V}\|)$
Apply	$O(\|\mathsf{V}\|)$	$O(\|\mathsf{V}\|)$	$O(\|\mathsf{E}\|)$	$O(\|\mathsf{E}\|/\|\mathsf{V}\|)$	$O(1)$	$O(1)$	$O(1)$

TABLE I: **Complexity of our parallel oblivious algorithms assuming $|\mathsf{E}| = \Omega(|\mathsf{V}|)$.** $|V|$ denotes the number of vertices, and $|\mathsf{E}|$ denotes the number of edges. $d_{\max}$ denotes the maximum degree of a vertex in the graph. *Blowup* is defined as the ratio of the parallel oblivious algorithm with respect to the best known parallel insecure algorithm. We assume that the data length on each vertex/edge is upper-bounded by a known bound D, and for simplicity we omit a multiplicative factor of D from our asymptotical bounds. In comparison with Theorem 1, in this table, some $|\mathsf{V}|$ terms are absorbed by the $|\mathsf{E}|$ term since $|\mathsf{E}| = \Omega(|\mathsf{V}|)$.

Parallel Aggregate:
/ For convenience, assume that for $i \le 0$, $\mathsf{G}[i]$ is white; and similarly for $i \le 0$, $\mathsf{LPS}[i, j)$ is an alias for $\mathsf{LPS}[1, j)$ */.*

Initialize: Every processor j computes:

$$\mathsf{LPS}[j-1, j) := \begin{cases} \mathsf{G}[j-1].\mathsf{data} & \text{if } \mathsf{G}[j-1] \text{ is black} \\ \mathbf{1}_{\oplus} & \text{o.w.} \end{cases}; \qquad \mathsf{existswhite}[j-1, j) := \begin{cases} \mathsf{False} & \text{if } \mathsf{G}[j-1] \text{ is black} \\ \mathsf{True} & \text{o.w.} \end{cases}$$

Main algorithm: For each time step $\tau := 1$ to $\log(|\mathsf{V}| + |\mathsf{E}|) - 1$: each processor j computes

- $\mathsf{LPS}[j - 2^\tau, j) := \begin{cases} \mathsf{LPS}[j - 2^\tau, j - 2^{\tau-1}) \oplus \mathsf{LPS}[j - 2^{\tau-1}, j) & \text{if } \mathsf{existswhite}[j - 2^{\tau-1}, j) = \mathsf{False} \\ \mathsf{LPS}[j - 2^{\tau-1}, j) & \text{o.w.} \end{cases}$
- $\mathsf{existswhite}[j - 2^\tau, j) := \mathsf{existswhite}[j - 2^\tau, j - 2^{\tau-1})$ or $\mathsf{existswhite}[j - 2^{\tau-1})$

Fig. 4: Performing the aggregate operation (Step 2 of Gather) in parallel, assuming sufficient number of processors with a shared memory to store the variables.

Cost analysis. Recall our standing assumption that the maximum data length on each tuple is $O(1)$. It is not hard to see that the parallel runtime of both the *aggregate* and *propagate* operations is $O(\log(|\mathsf{V}| + |\mathsf{E}|))$. The total amount of work for both *aggregate* and *propagate* is $O((|\mathsf{V}|+|\mathsf{E}|) \cdot \log(|\mathsf{V}|+|\mathsf{E}|))$.

Based on this, we can see that Scatter and Gather each takes $O(\log(|\mathsf{V}|+|\mathsf{E}|))$ parallel time and $O((|\mathsf{V}|+|\mathsf{E}|) \cdot \log(|\mathsf{V}|+|\mathsf{E}|))$ total amount of work. Obviously, Apply takes $O(1)$ parallel time and $O(|\mathsf{V}| + |\mathsf{E}|)$ total work.

Table I illustrates the performance of our parallel oblivious algorithms for the common case when $|\mathsf{E}| = \Omega(|\mathsf{V}|)$, and the blowup in comparison with a parallel insecure version. Notice that in the insecure world, there exists a trivial $O(1)$ parallel-time algorithm to evaluate Scatter and Apply operations. However, in the insecure world, Gather would take $O(\log(|\mathsf{E}| + |\mathsf{V}|))$ parallel time to evaluate the $\oplus$-sum over $|\mathsf{E}| + |\mathsf{V}|$ variables. Notice also that the $|\mathsf{V}|$ term in the asymptotic bound is absorbed by the $|\mathsf{E}|$ term when $|\mathsf{E}| = \Omega(|\mathsf{V}|)$. The above performance characterization is summarized by the following theorem:

Theorem 1 (Parallel oblivious algorithm for GraphSC): Let $M := |\mathsf{V}| + |\mathsf{E}|$ denote the graph size. There exists a parallel oblivious algorithm for programs in the GraphSC model, where each Scatter or Gather operation requires $O(\log M)$ parallel time and $O(M \log M)$ total work; and each Apply operation requires $O(1)$ parallel time and $O(M)$ total amount of work.

E. Practical Optimizations for Fixed Number of Processors

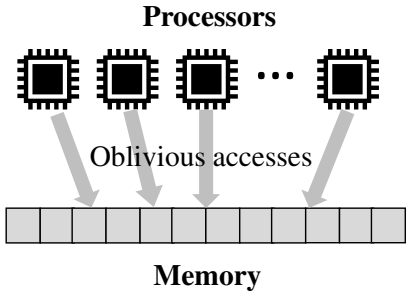
The parallel algorithm described in Figure 4 requires $M = |\mathsf{V}|+|\mathsf{E}|$ processors. In practice, however, for large datasets, the number of processors P may be smaller than M. Without loss of generality, suppose that M is a multiple of P. In this case, a naïve approach is for each processor to simulate $\frac{M}{P}$ processors, resulting in $\frac{M \log M}{P}$ parallel time, and $M \log M$ total amount of work. We propose the following practical optimization that can reduce the total parallel time to $O(\frac{M}{P} + \log P)$, and reduce the total amount of work to $O(P \log P + M)$.

We assign to each processor a consecutive range of cells. Suppose that processor j gets range $[s_j, t_j]$ where $s_j = (j-1) \cdot \frac{M}{P} + 1$ and $t_j = j \cdot \frac{M}{P}$. In our algorithm, each processor will compute $\mathsf{LPS}[1, s_j)$, and afterwards, in $O(M/P)$ time-steps, it can (sequentially) compute $\mathsf{LPS}[1, i)$ for every $s_j \le i \le t_j$. Every processor then computes $\mathsf{LPS}[1, s_j)$ as follows

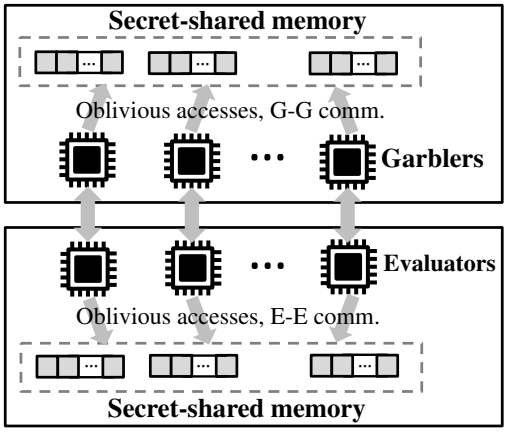
- First, every processor sequentially computes $\mathsf{LPS}[s_j, t_j + 1)$ and $\mathsf{existswhite}[s_j, t_j + 1)$.
- Now, assume that every processor started with a single value $\mathsf{LPS}[s_j, t_j + 1)$ and a single value $\mathsf{existswhite}[s_j, t_j + 1)$. Perform the parallel aggregate algorithm on this array of length P.

Sparsity of communication. In a distributed memory setting where memory is split across the processors, the conceptual shared memory is in reality implemented by inter-process communication. An additional advantage of our algorithm is that each processor needs to communicate with at most $O(\log P)$ other processors – this applies to both the oblivious sort step, and the *aggregate* or *propagate* steps. In fact, it is not hard to see that the communication graph forms a hypercube [67]. The sparsity of the communication graph is highly desirable.

Let $M := |\mathsf{V}|+|\mathsf{E}|$ and recall that the maximum amount of



(a) Architecture for parallel oblivious algorithms.



(b) Architecture for parallel secure computation.

Fig. 5: From parallel oblivious algorithms to parallel secure computation.

data on each vertex or edge is $O(1)$. The following corollary summarizes the above observations:

Corollary 1 (Bounded processors, distributed memory.): When $P < M$, there exists a parallel oblivious algorithm for programs in the GraphSC model, where (a) each processor stores $O(M/P)$ amount of data; (b) each Scatter or Gather operation requires $O(M/P + \log P)$ parallel time and $O(P \log P + M)$ total work; (c) each Apply operation requires $O(1)$ parallel time and $O(|\mathsf{E}| + |\mathsf{V}|)$ total amount of work; and (d) each processor sends messages to only $O(\log P)$ other processors.

Security analysis. The oblivious nature of our algorithms is not hard to see: in every time step, the shared memory locations accessed by each processor is fixed and independent of the sensitive input. This can be seen from Figure 4, and the description of practical optimizations in this section.

IV. From Parallel Oblivious Algorithms to Parallel Secure Computation

So far, we have discussed how GraphSC primitives can be implemented as efficient parallel oblivious algorithms, we now turn our attention to how the latter translate to parallel *secure* computation. In this section, we outline the reduction between the two, focusing on a garbled-circuit backend [1] for secure computation.

System Setting. Recall that our focus in this paper is on secure 2-party computation. As an example, Figure 5b depicts two *non-colluding* cloud service providers (e.g., Facebook and Amazon) – henceforth referred to as the two parties. The sensitive data (e.g., user preference data, sensitive social graphs) can be secret-shared between these two parties. Each party has P processors in total – thus there are in total P pairs of processors. The two parties wish to run a parallel secure computation protocol computing a function (e.g., matrix factorization), over the secret-shared data.

While in general, other secure 2-party computation protocols can also be employed, this paper focuses on a garbled circuit backend [1]. Our focus is on the semi-honest model, although this can be extended with existing techniques [6], [68]. Using this secure model, the oblivious algorithm is represented as a binary circuit. One party then acts as the *garbler* and the other acts as the *evaluator*, as illustrated in Figure 5b. To exploit parallelization, each of the two parties parallelize the computational task (garbling and evaluating the circuit, respectively) across its processors. There is a one-to-one mapping between garbler and evaluator processors: each garbler processor sends the tables it garbles to the corresponding corresponding evaluator processor, that evaluates them. We refer to such communication as *garbler-to-evaluator* (GE) communication.

Note that there is a natural correspondence between a parallel oblivious algorithm and a parallel secure computation protocol: First, each processor in the former becomes a (garbler, evaluator) pair in the latter. Second, memory in the former becomes secret-shared memory amongst the two parties. Finally, in each time step, each processor's computation in the former becomes a secure evaluation protocol between a (garbler, evaluator) pair in the latter.

Architectural choices for realizing parallelism. There are various choices for instantiating the parallel computing architecture of each party in Figure 5b.

- *Multi-core processor architecture.* At each party, each processor can be implemented by a core in a multi-core processor architecture. These processors share a common memory array.
- *Compute cluster.* At each party, each processor can be a machine in a compute cluster. In this case, accesses to the "shared memory" are actually implemented with *garbler-to-garbler* communication or *evaluator-to-evaluator* communication. In other words, the memory is conceptually shared but physically distributed.
- *Hybrid.* The architecture can be a hybrid of the above, with a compute cluster where each machine is a multi-core architecture.

While our design applies to all three architectures, we used a hybrid architecture in our implementation, exploiting both multi-core *and* multi-machine parallelism. Note that, in the case of a hybrid or cluster architecture with P machines,

Corollary 1 implies that each garbler (evaluator) communicates with only $O(\log P)$ other garblers (evaluators) throughout the entire execution. In particular, both garblers and evaluators connect through a hypercube topology. This is another desirable property of GraphSC.

Metrics. Using the above natural correspondence between a parallel oblivious algorithm and a parallel secure computation protocol, there is also a natural correspondence between the primary performance metrics in these two settings: First, the total work of the former directly characterizes (a) the total work *and* (b) the total garbler-to-evaluator (GE) communication in the latter. Second, the parallel runtime of the former directly characterizes the parallel runtime of the latter. We note that, in theory, the garbler is infinitely parallelizable, as each gate can be garbled independently. However, the parallelization of the evaluator (and, thus, of the entire system) is confined by the sequential order defined by the circuit. Thus, parallel runtime is determined by the circuit depth.

In the cluster and hybrid cases, where memory is conceptually shared but physically distributed, two additional metrics may be of interest, namely, the *garbler-to-garbler* (GG) communication and *evaluator-to-evaluator* (EE) communication. These directly relate to the parallel runtime, since in each parallel time step, each processor makes only one memory access; hence, each processor communicates with at most one other processor at each time-step.

V. Evaluation

In this section we present a detailed evaluation of our systems for a few well-known applications that are commonly used for evaluating highly-parallelizable frameworks.

A. Application Scenarios

In all scenarios, we assume that the data is secret-shared across two non-colluding cloud providers, as motivated in Section IV. In all cases, we refer to the total number of vertices and edges in the corresponding GraphSC graph as *input size*.

Histogram. A canonical use case of MapReduce is a word-count (or histogram) of words across multiple documents. Assuming a (large) corpus of documents, each comprising a set of words, the algorithm counts word occurrences across all documents. The MapReduce algorithm maps each word as a key with the value of 1, and the reducer sums up the values of all keys, resulting in the count of appearances of each word. In the secure version, we want to compute the word frequency histogram while hiding the text in each document. In GraphSC, this is a simple instance of edge counting over a bipartite graph G, where edges connect keys to words. We represent keys and words as 16-bit integers, while accumulators (i.e., key vertex data) are stored using 20-bit integers.

Simplified PageRank. A canonical use case of graph parallelization models is the PageRank algorithm. We consider a scenario in which multiple social network companies, e.g., Facebook, Twitter and LinkedIn, would like to compute the "real" social influence of users on a social graph that is the aggregate of each company's graph (assume users are uniquely identified across networks by their email address). In the secure version, each company is not willing to reveal user data and their social graph with the other network. Vertices are identified using 16-bit integers, and 1bit for isVertex (see Section III-C). The PageRank value of each vertex is stored using a 40-bit fixed-point representation, with 20-bit for the fractional part.

Matrix Factorization (MF). Matrix Factorization [61] splits a large sparse low-rank matrix into two dense low-dimension matrices that, when multiplied, closely approximate the original matrix. Following the Netflix prize competition [69], matrix factorization is widely used in recommender systems. In the secure version, we want to factorize the matrix and learn the user or item feature vectors (learning both can reveal the original input), while hiding both the ratings and items each user has rated. MF can be expressed in GraphSC using a bipartite graph with vertices representing users and items, and edges connecting each user to the items they rated, carrying the ratings as data. In addition, data at each vertex also contains a feature vector, corresponding to its respective row in the user/item factor matrix. We study two methods for matrix factorization – gradient descent and alternative least-squares (ALS) (see, e.g., [61]). In gradient descent, the gradient is computed for each rating separately, and then accumulated for each user and each item feature vectors, thus it is highly parallelizable. In ALS we alternate the computation between user feature vectors (assuming fixed item feature vectors) and item feature vectors (assuming fixed user feature vectors). For each step, each vector solves (in parallel) a linear regression using the data from its neighbors. Similar to PageRank, we use 16-bit for vertex id and 1-bit for isVertex. The user and item feature vectors are with dimension 10, with each element stored as a 40-bit fixed-point real.

The secure implementation of matrix factorization using gradient descent has been studied by Nikolaenko et al. [3] who, as discussed in Section I-C, constructed circuits of linear depth. The authors used a multi-core machine to exploit parallelization during sorting, and relied on shared memory across threads. This limits the ability to scale beyond a single machine, both in terms of the number of parallel processors (32 processors) as well as, crucially, input size (they considered no more than 17K ratings, over a 128 GB RAM server).

B. Implementation

We implemented GraphSC atop ObliVM-GC, the Java-based garbled circuit implementation that comprises the back end of the GraphSC secure computation framework [11], [70]. ObliVM-GC provides easy-to-use Java classes for composing circuit libraries. We extend ObliVM-GC with a simple MPI-like interface where processes can additionally call non-blocking `send` and blocking `receive` operations. Processes in ObliVM-GC are identified by their unique identifiers.

Finally, we implement oblivious sorting using the bitonic sort protocol [64] which sorts in $O(N \log^2 N)$ time. Asymptotically faster protocols such as the $O(N \log N)$ AKS sort [66] and the recent ZigZag sort [71] are much slower in practice for practical ranges of data sizes.

C. Setup

We conduct experiments on both a testbed that uses a LAN, and on a realistic Amazon AWS deployment. We first describe our main experiments conducted using a compute cluster

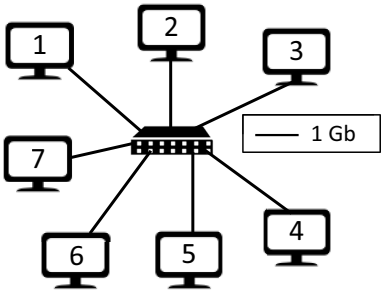


Fig. 6: Evaluation setup, all machines are connected in a star topology with 1Gbps links.

TABLE II: Servers' hardware used for our evaluation.

Machine	#Proc	Memory	CPU Freq	Processor
1	24	128 GB	1.9 GHz	AMD Opteron 6282 SE
2	24	128 GB	1.9 GHz	AMD Opteron 6282 SE
3	24	64 GB	1.9 GHz	AMD Opteron 6282 SE
4	24	64 GB	1.9 GHz	AMD Opteron 6282 SE
5	24	64 GB	1.9 GHz	AMD Opteron 6282 SE
6	32	128 GB	2.1 GHz	AMD Opteron 6272
7	32	256 GB	2.6 GHz	AMD Opteron 6282 SE

connected by a Local Area Network. Later, in Section V-I, we will describe results from the AWS deployment.

Testbed Setup on Local Area Network: Our experimental testbed consists of 7 servers with the configurations detailed in Table II. These servers are inter-connected using a star topology with 1Gbps Ethernet links as shown in Figure 6. All experiments (except the large-scale experiment reported in Section V-F that uses all of them) are performed using a pair of servers from the seven machines. These servers were dedicated to the experiments during our measurements, not running processes by other users.

To verify that our results are robust, we repeated the experiments several times, and made sure that the standard deviation is small. For example, we ran PageRank 10 times using 16 processors for an input length of 32K. The resulting mean execution time was 390 seconds, with a standard deviation of 14.8 seconds; we therefore report evaluations from single runs.

D. Evaluation Metrics

We study the gains and overheads that result from our parallelization techniques and implementation. Specifically, we study the following key metrics:

Total Work. We measure the total work using the overall number of AND gates for each application. As mentioned earlier in Section III-E, the total work grows logarithmically with respect to the number of processors P in theory – and in practice, since we employ bitonic sort, the actual growth is log-squared.

Actual runtimes. We report our actual runtimes and compare the overhead with a cleartext baseline running over GraphLab [9], [12], [72]. We stress that while our circuit size metrics are *platform independent*, actual runtime is a platform dependent metric. For example, we expect a factor of 20 speedup if the backend garbled circuit implementation adopts a JustGarble-like approach (using hardware AES-NI) – assuming roughly 2700 Mbps bandwidth provisioned between each garbler and evaluator pair.

Speedup. The obvious first metric to study is the speedup in the time to run each application as a result of adding more processors. In our applications, computation is the main bottleneck. Therefore, in the ideal case, we should observe a factor of x speedup with x factor more processors.

Communication. Parallelization introduces communication overhead between garblers and between evaluators. We study this overhead and compare it to the communication between garblers and evaluators.

Accuracy. Although not directly related to parallelization, for completeness we study the loss in accuracy obtained as a result of implementing the secure version of the applications, both when using fixed-point representation and floating-point representation of the reals.

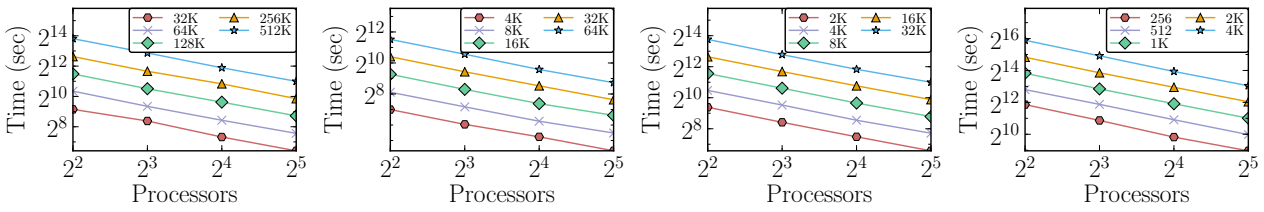
E. Main Results

Speedup. Figure 7 shows the total computation time across the different applications. For all applications except histogram we show the time of a single iteration (consecutive iterations are independent). Since in our experimental setup computation is the bottleneck, the figures show an almost ideal linear speedup as the number of processors grow. Figure 8 shows that our method is highly scalable with the input size, with an almost linear increase (a factor of $O(P/\log^2 P)$). Figure 8a provides the time to compute a histogram using an oblivious RAM implementation. We use the state-of-the-art Circuit ORAM [53] for this purpose. As the figure shows, the baseline is 2 orders of magnitude slower compared to the parallel version using two garblers and two evaluators.

Figure 8c provides the timing presented in Nikolaenko *et al.* [4] using 32 processors. As the figure shows, using a similar hardware architecture, we manage to achieve a speedup of roughly $\times 16$ compared to their results. Most of the performance gains comes from the usage of GraphSC architecture – whereas Nikolaenko *et al.* used a multi-threaded version of FastGC [5] as the secure computation backend.

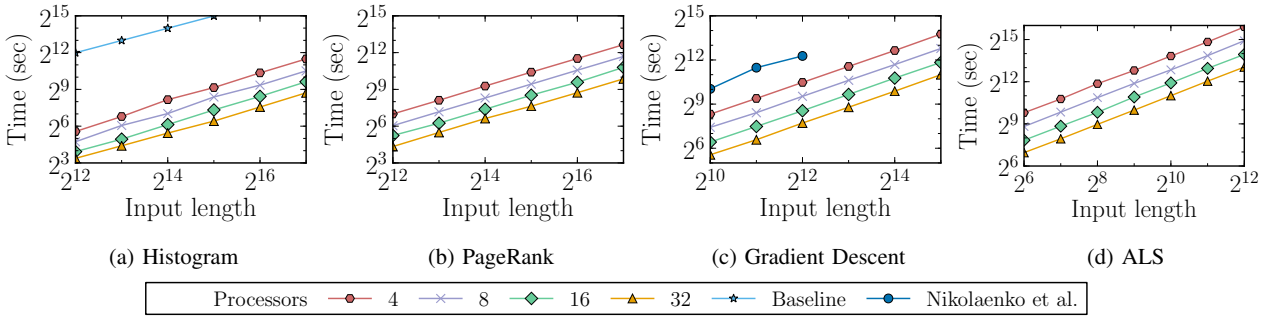
Total Work. Figure 9 shows that the total amount of work grows very slowly with respect to the number of processors, indicating that we indeed achieved a very low overhead in the total work (and overall circuit size).

Communication. Figure 10a and Figure 10b show the amount of total communication and per processor communication, respectively, for running gradient descent. Each plot shows both the communication between garblers and evaluators, and the overhead introduced by the communication between garblers (communication between evaluators is identical). Figure 10a shows that the total communication between garblers and evaluators remains constant as we increase the number of processors, showing that parallelization does not introduce overhead to the garblers-to-evaluator communication. Furthermore, the garbler-to-garbler (GG) communication is significantly lower than the garblers-to-evaluator communication, showing that the communication overhead due to parallelization is low. As



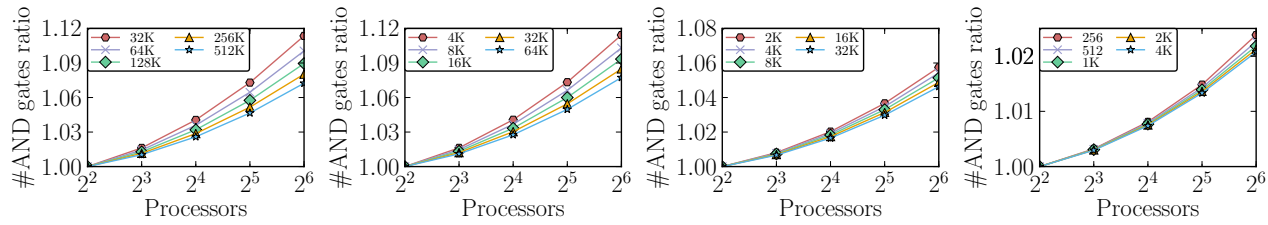
(a) Histogram (b) PageRank (c) Gradient Descent (d) ALS

Fig. 7: Computation time for increasing number of processors, showing an almost linear decrease with the number of processors. The lines correspond to different input lengths. For PageRank, gradient descent and ALS, the computation time refers to the time required for one iteration.



(a) Histogram (b) PageRank (c) Gradient Descent (d) ALS

Fig. 8: Computation time for increasing input size, showing an almost-linear increase with the input size, with a small $\log^2$ factor incurred by the bitonic sort. The lines correspond to different input lengths. For PageRank, gradient descent and ALS, the computation time refers to the time required for one iteration. In Figure 8a, the baseline is a sequential ORAM-based baseline using Circuit ORAM [53]. The ORAM-based implementation is not amenable to parallelization as explained in Section V-G. Figure 8c compares our performance with the performance of Nikolaenko *et al.* [3] who implemented the circuit using FastGC [5] and parallelized at the circuit level using 32 processors.



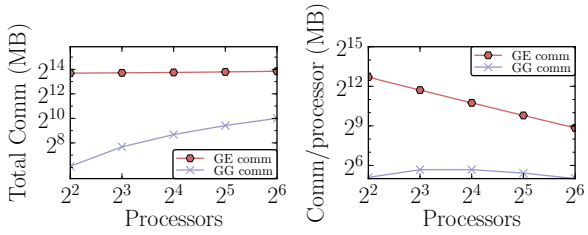
(a) Histogram (b) PageRank (c) Gradient Descent (d) ALS

Fig. 9: Total work in terms of # AND gates, normalized such that the 4 processor case is $1\times$. The different curves correspond to different input lengths. Plots are in a log-log scale, showing the expected small increase to the number of processors P. Recall that our theoretical analysis suggests that the total amount of work is $O(P \log P + M)$, where $M := |\mathsf{V}| + |\mathsf{E}|$ is the graph size. In practice, since we use bitonic sort, the actual total work is $O(P \log^2 P + M)$.

expected, adding more processors increases the total communication between garblers, following $\log^2 P$ (where P is the number of processors), due to the bitonic sort. Figure 10b shows the communications per-processor (dividing the results of Figure 10a by P). This helps understand overheads in our setting, where, for example, a cloud provider that provides secure computation services (garbling or evaluating) is interested in the communication costs of its facility rather than the total costs. As the number of processors increase, the "out-going" communication (e.g., a provider running garblers see the communication with evaluators as "out-going" communication) decreases. The GG communication (or EE communication) remains roughly the same (following $\log^2 P/P$), and significantly lower than the "out-going" communication.

Practical Optimizations. The optimization discussed in Section III-E decreases the amount of computation for the propagate and aggregate operations. We analyze the decrease in computation as a result of this optimization.

Figure 11 shows the number of (computed analytically) aggregate operation performed on an input length of 2048, using two scenarios: (a) one processor simulating multiple



(a) Total Communication (b) Communication per processor

Fig. 10: Communication of garbler-evaluator (GE) and garbler-garbler (GG) for gradient descent (input length 2048).

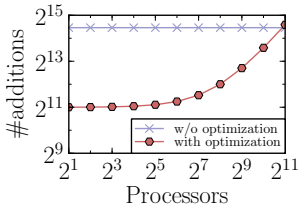


Fig. 11: Total number of aggregate operation (additions) on an input length of 2048, with and without optimization.

processors, (b) the optimization discussed in Section III-E is used. As can be seen in figure, the number of additions with optimization is much lower than the scenario where one processor simulates multiple processors. The optimized version performs worse than the single-processor version only when the number of processor comes close to the input size, a setting which is extremely unlikely for any real-world problem.

Comparison with a Cleartext Baseline. To better understand the overhead that is incurred from cryptography, we compared GraphSC's execution time with GraphLab [9], [12], [72], a state-of-the-art framework for running graph-parallel algorithms on clear text. We compute the slowdown relative to an insecure baseline, assuming that the same number of processors is employed for GraphLab and GraphSC. Using both frameworks, we ran Matrix Factorization using gradient descent with input length of 32K. For the cleartext experiments, we ran 1000 iterations of gradient descent 3 times, and computed the average time for a single iteration.

Figure 12 shows that GraphSC is about 200K - 500K times slower than GraphLab when run on 2 to 16 processors. Since GraphLab is highly optimized and extremely fast, such a large discrepancy is expected. Nevertheless, we note that increasing parallelism decreases this slowdown, as overheads and communication costs impact both systems.

Accuracy. Figures 13a and 13b show the relative error of running the secure version of PageRank compared to the

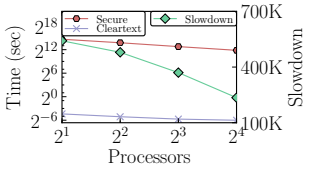
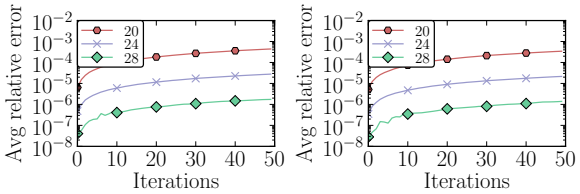


Fig. 12: Comparison with cleartext implementation on GraphLab for gradient descent (input length 32K)



(a) Fixed point (b) Floating point

Fig. 13: Relative accuracy of the secure PageRank algorithm (input length 2048 entries) compared to the execution in the clear using fixed-point and floating-point garbled-circuits implementations.

TABLE III: Summary of machines used in large-scale experiment, performing matrix factorization over the MovieLens 1M ratings dataset.

Machine	Processors	Type	JVM Memory Size	Num Ratings
1	16	Garbler	64 GB	256K
2	16	Evaluator	60.8 GB	256K
3	6	Garbler	24 GB	96K
3	6	Evaluator	24 GB	96K
4	15	Garbler	58.5 GB	240K
5	15	Evaluator	58.5 GB	240K
6	27	Garbler	113.4 GB	432K
7	27	Evaluator	121.5 GB	432K
Total	128		524.7 GB	1M

version in the clear for fixed-point and floating-point numbers, respectively. Overall, the error is relatively small, especially when using at least 24 bits for the fraction part in fixed-point or for the precision in floating-point. For example, running 10 iterations of PageRank with 24 bits for the fractional part in fixed-point representation results in an error of 10^{-5} compared to running in the clear. The error increases with more iterations since the precision error accumulates.

F. Running at Scale

In order to have a full-scale experiment of our system, we ran matrix factorization using gradient descent on the real-world MovieLens dataset that contains 1 million ratings provided by 6040 users to 3883 movies [73]. We factorized the matrix to users and movie feature vectors, each vector with a dimension of 10. We used 40-bit fixed-point representation for reals, with 20 bits reserved for the fractional part. We ran the experiment on an heterogeneous set of machines that we have in the lab. Table III summarizes the machines and the allocation of data across them.

A single iteration of gradient descent took roughly 13 hours to run on 7 machines with 128 processors, at ~104 MB data size (i.e., 1M entries). As prior machine learning literature reports [74], [75], about 20 iterations are necessary for convergence for the same MovieLens dataset – which would take about 11 days with 128 processors. In practice, this means that the recommendation system can be retrained every 11 days. As mentioned earlier, about 20× speedup is immediately attainable by switching to a JustGarble-like back end implementation with hardware AES-NI, and assuming 2700 Mbps bandwidth between each garbler-evaluator pair. One can also speed up the execution by provisioning more processors.

TABLE IV: Comparison with a naive circuit-level parallelization approach, assuming infinite number of processors (using Histogram).

Input length	Circuit Depth of GraphSC	Circuit Depth of SCVM [2]
2^{11}	267	7 M
2^{12}	322	18 M
2^{13}	385	43 M
2^{14}	453	104 M
2^{15}	527	247 M
2^{16}	608	576 M
2^{17}	695	1328 M
2^{18}	788	3039 M
2^{19}	888	6900 M
2^{20}	994	15558 M

In comparison, as far as we know, the closest large-scale experiment in running secure matrix factorization was recently performed by Nikolaenko *et al.* [3]. The authors used 16K ratings and 32 processors to factorize a matrix (on a machine similar to machine 7 in Table III), taking almost 3 hours to complete. The authors could not scale further because their framework runs on a single machine.

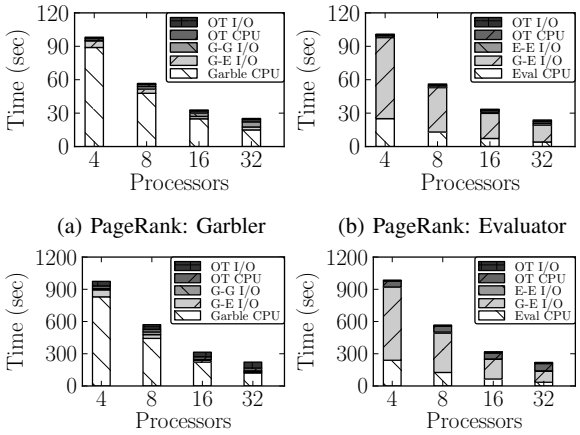
G. Comparison with Naïve Parallelization

An alternative approach to achieve parallelization is to use a naive circuit-level parallelization without requiring the developer to write code in a parallel programming paradigm. We want to assess the speedup that we can obtain using GraphSC over using such naïve parallelization. The results in this section are computed analytically and assume infinite number of processors.

In order to compare, we consider the simple histogram application and compute the depth of the circuit that is generated using GraphSC, and the one using the state-of-the-art SCVM [2] compiler. The depth is an indicator for the ability to parallelize – each "layer" in the circuit can be parallelized, but consecutive layers must be executed in sequence. Thus, the shallower the circuit is the more it is amendable to parallelization. The latter uses RAM-model secure computation and compiles a program into a sequence of ORAM accesses. We assume that for ORAM accesses, the compiler uses the state-of-the-art Circuit ORAM [53]. Due to the sequential nature of ORAM constructions, these ORAM accesses cannot be easily parallelized using circuit-level parallelism (currently only OPRAM can achieve full circuit-level parallelism, however, these results are mostly theoretical and prohibitive in practice). Table IV shows the circuit depth obtained using the two techniques. As the table suggests, GraphSC yields significantly shallower and "wider" circuits, implying that it can be parallelized much more than the naïve circuit-level parallelization techniques that are long and "narrow".

H. Performance Profiling

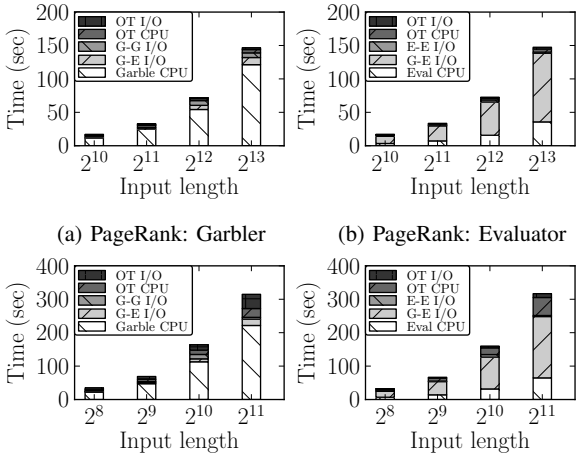
Finally, we perform micro-benchmarks to better understand the time the applications spend in the different parts of the computation and network transmissions. Figure 14 shows the breakdown of the overall execution between various operations for PageRank and gradient descent. Figure 15 shows a similar breakdown for different input sizes. As the plots show, the garbler is computation-intensive whereas the evaluator spends



(a) PageRank: Garbler (b) PageRank: Evaluator

(c) Gradient Descent: Garbler (d) Gradient Descent: Evaluator

Fig. 14: A breakdown of the execution times of the garbler and evaluator running one iteration of PageRank and gradient descent for an input size of 2048 entries Here I/O overhead means the time a processor spends blocking on I/O. The remaining time is reported as CPU time.

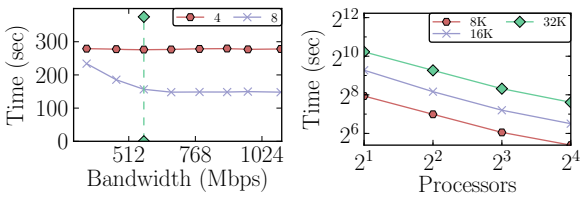


(a) PageRank: Garbler (b) PageRank: Evaluator

(c) Gradient Descent: Garbler (d) Gradient Descent: Evaluator

Fig. 15: A breakdown of the execution times of the garbler and evaluator running one iteration of PageRank and gradient descent for an increasing input size using 8 processors for garblers and 8 for evaluators.

a considerable amount of time waiting for the garbled tables (receive is a blocking operation). In our implementation, the garbler computes 4 hashes to garble each gate, and the evaluator computes only 1 hash for evaluation. This explains why the evaluation time is smaller than the garbling time. Since the computation tasks under consideration are superlinear in the size of the inputs, we see that the time spent on oblivious transfer (both communication and computation) is insignificant in comparison to the time for garbling/evaluating. Our current implementation is built atop Java, and we do not make use of hardware AES-NI instructions. We expect that the garbling and evaluation CPU will reduce noticeably if hardware AES-NI were employed [76]. We leave it for future work to port GraphSC to a C-based implementation capable of employing hardware AES-NI features.



(a) Varying bandwidths. (b) Across data centers

Fig. 16: Performance of PageRank. Figure 16a shows performance for 4 and 8 processors at varying bandwidths. The dotted vertical line indicates the inflexion point for 8 processors, below which the bandwidth becomes a bottleneck, resulting in reduced performance. Figure 16b shows the performance of PageRank running on geographically distant data centers (Oregon and North Virginia).

I. *Amazon AWS Experiments*

We conduct two experiments on Amazon AWS machines. First, we study the performance of the system under different bandwidths on the same AWS data center (Figure 16a). Second, to test the performance on a more realistic deployment, where the garbler and evaluator are not co-located, we also conduct experiments by deploying GraphSC on a pair of AWS virtual machines located in different geographical regions (Figure 16b).

The time reported for these experiments should not be compared to the earlier experiments as different machines were used.

Setup. For the experiments with varying bandwidths, both garblers and evaluators were located in the same data center (Oregon - US West). For the experiment across data centers, the garblers were located in Oregon (US West) and the evaluators were located in N. Virginia (US East). We ran our experiments on shared instances running on Intel Xeon CPU E5-2666 v3 processors clocked at 2.9 GHz. Each of our virtual machines consisted of 16 cores and 30 GB of RAM.

Results for Varying Bandwidths. Since communication between garblers and evaluators is a key component in system performance, we further study the bandwidth requirements of the system on a real-world deployment.

We measure the time for a single PageRank iteration with input length of 16K entries. We vary the bandwidth using `tc` [77], a tool for bandwidth manipulation, and then measure the exact bandwidth between machines using `iperf` [78].

Figure 16a shows the execution time for two setups, one with 4 processors (2 garblers and 2 evaluators) and the second with 8 processors. Using 4 processors the required bandwidth is always lower than the capacity of the link, thus the execution time remains the same throughout the experiment. However, when using 8 processors the total bandwidth required is higher, and when the available bandwidth is below 570 Mbps the link becomes saturated. The saturation point indicates that each garbler-evaluator pair requires a bandwidth of $570/4 \approx 142$ Mbps. GraphSC has an effective throughput of ~ 0.58M gates/sec between a pair of processors on our Amazon AWS instances. Each gate has a size of 240 bits. Hence, the theoretical bandwidth required is $0.58 \times 240 \times 10^6/2^{20} \approx 133$ Mbps.

TABLE V: Summary of key evaluation results (1 iteration).

Experiment	Input size	Time (32 processors)
Histogram	1K - 0.5M	4 sec - 34 min
PageRank	4K - 128K	20 sec - 15.5 min
Gradient Descent	1K - 32K	47 sec - 34 min
ALS	64 - 4K	2 min - 2.35 hours
Gradient Descent large scale)	1M ratings	13 hours (128 processors)

Considering GraphSC is implemented in Java, garbage collection happens intermittently due to which the communication link is not used effectively. Hence, the implementation requires slightly more bandwidth than the theoretical calculation.

Given such bandwidth requirements, the available bandwidth in our AWS setup, i.e., 2 Gbps between the machines, will saturate beyond roughly 14 garbler-evaluator pairs (28 processors). At this point, the linear speedup trend w.r.t. the number of processors (as shown in Figure 7) will stop, unless larger bandwidth becomes available. In a real deployment scenario, the total bandwidth can be increased by having multiple machines for garbling and evaluating, hence supporting more processors without affecting the speedup.

Results for Cross-Data-Center Experiments. For this experiment, the garblers are hosted in the AWS Oregon data center and the evaluators are hosted in the AWS North Virginia data center. We measure the execution time of a single iteration of PageRank for different input lengths. As in the previous experiment, we used machines with 2Gbps network links, however, measuring the TCP throughput with `iperf` resulted in ~50 Mbps per TCP connection. By increasing the receiver TCP buffer size we managed to increase the effective throughput for each TCP connection to ~400 Mbps.

Figure 16b shows that this realistic deployment manages to sustain a linear speedup when increasing the number of processors. Moreover, even 16 processors do not saturate the 2 Gbps link, meaning that the geographical distance does not impact the speedup resulting from adding additional processors. We note that if more than 14 garbler-evaluator pairs are needed (to further reduce execution time), AWS provides higher capacity links (e.g., 10 Gbps), thereby allowing even higher degrees of parallelism.

During the computation, the garbler garbles gates and sends it to the evaluator. As there are no round trips involved (i.e. garbler does not wait to receive data from the evaluator), the time required for computation across data centers is the same as in the LAN setting.

J. *Summary of Main Results*

To summarize, Table V highlights some of the results, and we present the main findings:

- As mandated from "big-data" algorithms, GraphSC provides high scalability with the input size, exhibiting an almost linear increase with the input size (up to poly-log factor).
- Parallelization provides an almost ideal linear improvement in execution time with small communication overhead (especially on computation-intensive tasks), both in a LAN based setting and across data centers.

- We ran a first-of-its-kind large-scale secure matrix factorization experiment, factorizing a matrix comprised of the MovieLens 1M ratings dataset within 13 hours on a heterogeneous set of 7 machines with a total of 128 processors.
- GraphSC supports fixed-point and floating-point reals representation, yielding an overall low rounding errors (provided sufficient fraction bits) compared to execution in the clear.

VI. Conclusion

This paper introduces GraphSC, a parallel data-oblivious and secure framework for efficient implementation and execution of algorithms on large datasets. It is our sincere hope that by seamlessly integrating modern parallel programming paradigms that are familiar to a wide range of developers into an secure data-oblivious framework will significantly increase the adoption of secure computation. We believe that this can truly change the privacy landscape, where companies that operate on potentially sensitive datasets, will be able to develop arbitrarily complicated algorithms that run in parallel on large datasets as they normally do, only without leaking information.

VII. Acknowledgments

We gratefully acknowledge Marc Joye, Manish Purohit and Omar Akkawi for their insightful inputs and various forms of support. We thank the anonymous reviewers for their insightful feedback. This research is partially supported by an NSF grant CNS-1314857, a Sloan Fellowship and a subcontract from the DARPA PROCEED program.

References

[1] A. C.-C. Yao, "How to generate and exchange secrets," in *FOCS*, 1986.

[2] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks, "Automating efficient ram-model secure computation," in *IEEE S & P*, 2014.

[3] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, "Privacy-preserving matrix factorization," in *ACM CCS*, 2013.

[4] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft, "Privacy-preserving ridge regression on hundreds of millions of records," in *IEEE (S & P)*, 2013.

[5] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits." in *USENIX Security Symposium*, 2011.

[6] B. Kreuter, a. shelat, and C.-H. Shen, "Billion-gate secure computation with malicious adversaries," in *USENIX Security symposium*, 2012.

[7] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, 2008.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *SIGMOD*, 2010.

[9] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *PVLDB*, 2012.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *HotCloud*, 2010.

[11] X. S. Wang, C. Liu, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," *IEEE Symposium on Security and Privacy (S & P)*, 2015.

[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, 2012.

[13] E. Boyle, K.-M. Chung, and R. Pass, "Oblivious parallel ram," https://eprint.iacr.org/2014/594, 2014.

[14] M. T. Goodrich, O. Ohrimenko, and R. Tamassia, "Data-oblivious graph drawing model and algorithms," *CoRR*, 2012.

[15] D. Eppstein, M. T. Goodrich, and R. Tamassia, "Privacy-preserving data-oblivious geometric algorithms for geographic data," in *SIGSPATIAL*, 2010.

[16] S. Zahur and D. Evans, "Circuit structures for improving efficiency of security and privacy tools," in *S & P*, 2013.

[17] M. Blanton, A. Steele, and M. Alisagari, "Data-oblivious graph algorithms for secure computation and outsourcing," in *ASIA CCS*. ACM, 2013.

[18] A. C.-C. Yao, "Protocols for secure computations (extended abstract)," in *FOCS*, 1982.

[19] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis, "Secure two-party computation in sublinear (amortized) time," in *ACM CCS*, 2012.

[20] a. shelat and C.-H. Shen, "Fast two-party secure computation with minimal assumptions," in *CCS*, 2013.

[21] ——, "Two-output secure computation with malicious adversaries." in *EUROCRYPT*, 2011.

[22] F. Kerschbaum, "Automatically optimizing secure computation," in *CCS*, 2011.

[23] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations."

[24] B. Kreuter, B. Mood, A. Shelat, and K. Butler, "PCF: A portable circuit format for scalable two-party secure computation," in *USENIX Security*, 2013.

[25] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay: a secure two-party computation system," in *USENIX Security Symposium*, 2004.

[26] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "Tasty: tool for automating secure two-party computations," in *CCS*, 2010.

[27] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A programming language for generic, mixed-mode multiparty computations," in *IEEE Symposium on Security and Privacy (S & P)*, 2014.

[28] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, "Secure two-party computations in ansi c," in *CCS*, 2012.

[29] Y. Zhang, A. Steele, and M. Blanton, "Picco: a general-purpose compiler for private distributed computation," in *CCS*, 2013.

[30] V. Kolesnikov and T. Schneider, "Improved Garbled Circuit: Free XOR Gates and Applications," in *ICALP*, 2008.

[31] S. G. Choi, J. Katz, R. Kumaresan, and H.-S. Zhou, "On the security of the "free-xor" technique," in *Theory of Cryptography Conference (TCC)*, 2012.

[32] B. Applebaum, "Garbling xor gates "for free" in the standard model," in *Theory of Cryptography Conference (TCC)*, 2013.

[33] N. Husted, S. Myers, A. Shelat, and P. Grubbs, "Gpu and cpu parallelization of honest-but-curious secure two-party computation," in *Annual Computer Security Applications Conference*, 2013.

[34] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in *STOC*, 1987.

[35] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *ACM symposium on Theory of computing (STOC)*, 2009.

[36] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure mpc for dishonest majority–or: Breaking the spdz limits," in *Computer Security–ESORICS 2013*, 2013.

[37] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation," in *ACM STOC*, 1988.

[38] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, 1996.

[39] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious RAM simulation," in *SODA*, 2012.

[40] R. Ostrovsky and V. Shoup, "Private information storage (extended abstract)," in *ACM Symposium on Theory of Computing (STOC)*, 1997.

[41] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in)security of hash-based oblivious RAM and a new balancing scheme," in *SODA*, 2012.

[42] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Oblivious RAM simulation with efficient worst-case access overhead," in *CCSW*, 2011.

[43] I. Damgård, S. Meldgaard, and J. B. Nielsen, "Perfectly secure oblivious RAM without random oracles," in *TCC*, 2011.

[44] D. Boneh, D. Mazieres, and R. A. Popa, "Remote oblivious storage: Making oblivious RAM practical," http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf, Tech. Rep., 2011.

[45] P. Williams, R. Sion, and B. Carbunar, "Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage," in *CCS*, 2008.

[46] P. Williams and R. Sion, "Usable PIR," in *Network and Distributed System Security Symposium (NDSS)*, 2008.

[47] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious RAM simulation," in *ICALP*, 2011.

[48] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *ACM Symposium on Theory of Computing (STOC)*, 1990.

[49] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *CRYPTO*, 2010.

[50] P. Williams and R. Sion, "SR-ORAM: Single round-trip oblivious ram," in *ACM CCS*, 2012.

[51] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((\log N)^3)$ worst-case cost," in *ASIACRYPT*, 2011.

[52] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM – an extremely simple oblivious ram protocol," in *CCS*, 2013.

[53] X. S. Wang, T.-H. H. Chan, and E. Shi, "Circuit oram: On tightness of the goldreich-ostrovsky lower bound," Cryptology ePrint Archive, Report 2014/672, 2014, http://eprint.iacr.org/.

[54] K.-M. Chung, Z. Liu, and R. Pass, "Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead," *CoRR*, 2013.

[55] X. Wang, K. Nayak, C. Liu, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," in *ACM CCS*, 2014.

[56] J. C. Mitchell and J. Zimmerman, "Data-Oblivious Data Structures," in *Theoretical Aspects of Computer Science (STACS)*, 2014.

[57] J. E. Savage, *Models of Computation: Exploring the Power of Computing*, 1997.

[58] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[59] C. Avery, "Giraph: Large-scale graph processing infrastruction on hadoop," *Hadoop Summit.*, 2011.

[60] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, 1998.

[61] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[62] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, 1988.

[63] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine Learning*, 2011.

[64] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*, 1998.

[65] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *et al.*, *Introduction to algorithms*. MIT press Cambridge, 2001.

[66] M. Ajtai, J. Komlós, and E. Szemerédi, "An o(n log n) sorting network," in *ACM symposium on Theory of computing*, 1983.

[67] R. Miller and L. Boxer, *Algorithms sequential & parallel: A unified approach*. Cengage Learning, 2012.

[68] Y. Lindell and B. Pinkas, "An efficient protocol for secure two-party computation in the presence of malicious adversaries," in *EUROCRYPT*, 2007.

[69] J. Bennett and S. Lanning, "The netflix prize," in *Proceedings of KDD cup and workshop*, 2007.

[70] "Oblivm," http://www.oblivm.com.

[71] M. T. Goodrich, "Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o(n log n) time," *CoRR*, 2014.

[72] "Graphlab powergraph tutorials," https://github.com/graphlab-code/graphlab.

[73] "Movielens dataset," http://grouplens.org/datasets/movielens/.

[74] S. Bhagat, U. Weinsberg, S. Ioannidis, and N. Taft, "Recommending with an agenda: Active learning of private attributes using matrix factorization," in *RecSys '14*. ACM.

[75] S. Ioannidis, A. Montanari, U. Weinsberg, S. Bhagat, N. Fawaz, and N. Taft, "Privacy tradeoffs in predictive analytics," in *SIGMETRICS'14*. ACM, 2014.

[76] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *IEEE Symposium on Security and Privacy (SP)*, 2013.

[77] "Tc man page," http://manpages.ubuntu.com/manpages//karmic/man8/tc.8.html.

[78] "Iperf," https://iperf.fr/.