SECURE FUNCTION EVALUATION AT SCALE

Stratis Ioannidis

joint work with:

Kartik Nayak, Xiao Shaun Wang
University of Maryland

Udi Weinsberg
Facebook

Nina Taft
Google

Elaine Shi
Cornell University
Analyzing Data from Human Subjects

- Long history in experimental/life sciences
  - Medicine
  - Psychology
  - Sociology
  - Behavioral Economics
  - ...

- Ubiquitous practice
  - Display & search ads
  - e-Commerce
  - Streaming
  - ...

- Integral to daily operations
  - User profiling
  - Targeted advertising
  - Personalized recommendations
The analyst learns only \( f(x_1, \ldots, x_n) \), and nothing else.
Grand Challenge

Apply SFE to execute **real-life, practical algorithms** over **massive datasets**
Challenge 1: Cost of Obliviousness

```python
function bs(val, s, t)
    mid = (s + t) / 2;
    if (val < mem[mid])
        bs(val, 0, mid)
    else
        bs(val, mid+1, t)
```

- Translating to **data-oblivious algorithm** can increase total work
- For “big data”, even going from $O(n)$ to $O(n^2)$ is prohibitive
Challenge 2: Maintaining Parallelism

- Very important for “big data”!
- Desirable properties:
  - Low parallel processing time
  - Low communication cost

Caveat: Communication patterns between processors *may reveal something about the data*
Oblivious RAM programs

[Gordon, Katz, Kolesnikov, Krell, Malkin, Raykova, Vahlis; CCS 2012]
[Lu, Ostrofsky; EUROCRYPT 2013]
[Gentry, Halevi, Raykova, Wichs; FOCS 2014]
[Liu, Huang, Shi, Katz, Hicks; IEEE S&P 2014]
[Zahur, Wang, Raykova, Gascon, Doerner, Evans, Katz; IEEE S&P 2016]

Oblivious PRAM programs

[Boyle, Chung, Pass; TCC 2016]
[Chen, Lin, Tessaro; TCC 2016]
Tailored Approaches

- Pick function/algorithm of interest
  - Linear regression, matrix factorization, SVMs, LDA, …

- Apply SFE techniques such as
  - Yao’s garbled circuits
  - Secret Sharing
  - Homomorphic Encryption
  - …

[De Cock, Dowsley, Nascimento, Newman, PMPML16]
[Takabi, Hesamifard, Ghasemi, PMPML16]
[Tian, Jayaraman, Gu, Evans, PMPML16]
[Schoppmann, Gascon, Raykova, Evans, Zahur, Doerner, Balle, PMPML16]
- Generic approach for SFE of **graph-parallel algorithm**
  - Scatter/Gather/Apply
  - Includes several important DM+ML algorithms

- SFE is highly parallelizable
  - Input size: $M$
  - Total Work: $O(M \log^2 M)$  Blowup: $O(\log^2 M)$
  - Parallel Time: $O(\log^2 M)$

- **GraphSC**
  - Generic Implementation
  - MF: 1M ratings, 128 cores: 13 hours
Overview

- Graph-Parallel Algorithms
- Graph-Parallel Secure Evaluation
- Implementation
Overview

- Graph-Parallel Algorithms
- Graph-Parallel Secure Evaluation
- Implementation
Graph-Parallel Algorithms

Graph-Parallel Algorithm: Computation happens over a directed graph through scatter, gather and apply operations.

[Malewicz, Austern, Bik, Dehnert, Horn, Leiser, Czajkowski; SIGMOD 2010]
[Low, Bickson, Gonzalez, Guestrin, Kyrola, Hellerstein; PVLDB 2012]
[Ching; Hadoop Summit 2011]
Graph-Parallel Algorithms

[Malewicz, Austern, Bik, Dehnert, Horn, Leiser, Czajkowski; SIGMOD 2010]
[Low, Bickson, Gonzalez, Guestrin, Kyrola, Hellerstein; PVLDB 2012]
[Ching; Hadoop Summit 2011]

Each node is a processor, edges describe communication links.
**Graph-Parallel Algorithms**

[Malewicz, Austern, Bik, Dehnert, Horn, Leiser, Czajkowski; SIGMOD 2010]
[Low, Bickson, Gonzalez, Guestrin, Kyrola, Hellerstein; PVLDB 2012]
[Ching; Hadoop Summit 2011]

---

**Scatter:** Every node sends data to its neighbors
Graph-Parallel Algorithms

[Gather:] Every node collects data from its neighbors and aggregates it.

[Malewicz, Austern, Bik, Dehnert, Horn, Leiser, Czajkowski; SIGMOD 2010]
[Low, Bickson, Gonzalez, Guestrin, Kyrola, Hellerstein; PVLDB 2012]
[Ching; Hadoop Summit 2011]
Graph-Parallel Algorithms

[Malewicz, Austern, Bik, Dehnert, Horn, Leiser, Czajkowski; SIGMOD 2010]
[Low, Bickson, Gonzalez, Guestrin, Kyrola, Hellerstein; PVLDB 2012]
[Ching; Hadoop Summit 2011]

Apply: Every node transforms its data locally
Graph-Parallel Algorithms

- There exist programming frameworks (e.g., GraphLab, Giraph) for parallelizing the execution of graph-parallel algorithms.

- Many interesting data mining, ML, and graph algorithms are graph-parallel:
  - Shortest paths
  - PageRank
  - Triangle counting
  - Graph coloring
  - Matrix Factorization through GD/ALS
  - ERM through GD
  - DNNs
  - ...

- map, reduce, reduce-by-key operations are graph-parallel
Consider a directed, sparse graph $G(V, E)$.
Graph-Parallel Algorithms: Formal definition

Both nodes \textbf{and} edges carry data.

\begin{align*}
G(V, E) \\
|E| &= O(|V|)
\end{align*}
Graph-Parallel Algorithms: Formal definition

Apply($f_A$): Every node $v \in V$ applies function $f_A$ to their data $d(v)$ in parallel.

$G(V, E)$
$|E| = O(|V|)$
Graph-Parallel Algorithms: Formal Definition

**Scatter**\(f_S\): Every node \(v \in V\) combines their data \(d(v)\) with data of adjoining edges through:

\[
d(e) \leftarrow f_S(d(e), d(v)) \quad \forall e \in N(v)
\]
Graph-Parallel Algorithms: Formal Definition

**Scatter**($f_S$): Every node $v \in V$ combines their data $d(v)$ with data of adjoining edges through:

$$d(e) \leftarrow f_S(d(e), d(v)) \quad \forall e \in N(v)$$

$$G(V, E) \quad |E| = O(|V|)$$
**Gather**($f_G, \odot$): Every node $v \in V$ aggregates adjacent edge data through binary operator $\odot$ and combines it with its local data through $f_G$. I.e.:

\[
d(v) \leftarrow f_G \left( \bigodot_{e \in N(v)} d(e) \right), \quad \text{where} \quad \bigodot_{e \in N(v)} d(e) = d(e_1) \odot d(e_2) \odot \ldots \odot d(e_k)
\]

\[
G(V, E) \quad |E| = O(|V|)
\]
Example: PageRank

Given $G(V, E)$, repeat

$$d(v) \leftarrow \gamma \frac{1}{|V|} + (1 - \gamma) \sum_{(u,v) \in N(v)} \frac{d(u)}{|N(u)|}, \quad \forall v \in V$$

until convergence
def PageRank(G):

Graph $G(V, E)$ is bidirectional
Example: PageRank

Nodes start with equal values:

\[ d(v) = \frac{1}{|V|}, \quad \forall v \in V \]

\[ \text{def } \text{PageRank}(G) : \]
\[ \text{Apply } (d(v) \mapsto 1/|V|) \]
Example: PageRank

In each iteration, nodes divide their values by their out-degree

\[ d(v) \leftarrow d(v)/N(v) \]

\texttt{def PageRank}(G):
  \text{Apply} \ (d(v) \mapsto 1/|V|)

\textbf{repeat:}
  \text{Apply}(d(v) \mapsto d(v)/N(v))
Example: PageRank

Values are scattered over outgoing edges

```python
def PageRank(G):
    Apply (d(v) ↦ 1/|V|)
    repeat:
        Apply (d(v) ↦ d(v)/N(v))
```
Example: PageRank

```
def PAGE_RANK(G):
    Apply(d(v) \mapsto 1/|V|)
    repeat:
        Apply(d(v) \mapsto d(v)/N(v))
        Scatter((x, y) \mapsto x)
```

Values are scattered over outgoing edges
Example: PageRank

**Algorithm**

```python
def PAGE_RANK(G):
    Apply \( d(v) \mapsto 1/|V| \)

repeat:
    Apply \( d(v) \mapsto d(v)/N(v) \)
    Scatter \((x, y) \mapsto x\)
    Gather \((x, y) \mapsto \gamma \frac{1}{|V|} + (1 - \gamma) y, +\)
```

**Equation**

\[
d(v) \leftarrow \gamma \frac{1}{|V|} + (1 - \gamma) \sum_{e \in N(v)} d(e)\]

**Diagram**

Values in incoming edges are **gathered**, **added**, and interpolated with \(1/|V|\):

\[
d(v) \leftarrow \gamma \frac{1}{|V|} + (1 - \gamma) \sum_{e \in N(v)} d(e)\]
Example: PageRank

```python
def PAGE_RANK(G):
    Apply (d(v) \mapsto 1/|V|)

repeat:
    Apply (d(v) \mapsto d(v)/N(v))
    Scatter ((x, y) \mapsto x)
    Gather ((x, y) \mapsto \gamma \frac{1}{|V|} + (1 - \gamma)y, +)

until convergence
```

Each iteration thus implements

\[
d(v) \leftarrow \gamma \frac{1}{|V|} + (1 - \gamma) \sum_{(u,v) \in N(v)} \frac{d(u)}{|N(u)|}, \quad \forall v \in V
\]

repeated until convergence
Example: PageRank

```python
def PAGE_RANK(G):
    Apply \((d(v) \mapsto 1/|V|)\)
    repeat:
        Apply\((d(v) \mapsto d(v)/N(v))\)
        Scatter\(((x, y) \mapsto x)\)
        Gather\(((x, y) \mapsto \gamma \frac{1}{|V|} + (1 - \gamma)y, +)\)
    until convergence
```

Input size:
\[ M = |V| + |E| = O(|V|) \]

Total work per iteration:
\[ O(M) \]

Messages per iteration:
\[ O(M) \]

Communication pattern reveals input!
Graph-Parallel Algorithms: Definition

**Definition:** An algorithm is **graph-parallel** if there exist:
- a **sparse graph** $G(V, E)$ and
- initial **node** and **edge** values, such that the algorithm can be written as a sequence of
  - **Scatter**($f_S$),
  - **Gather**($f_G, \odot$), and
  - **Apply**($f_A$) operations,
for appropriate $f_S, f_G, f_A$, and $\odot$

**Input size:**

$M = |V| + |E| = O(|V|)$

**Total work per operation:**

$O(M)$

**Messages per operation:**

$O(M)$
Example: Matrix Factorization

Dataset $\mathcal{D}$

<table>
<thead>
<tr>
<th>5</th>
<th>5</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>?</td>
<td>1</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

$r_{ij}$: rating by user $i$ to item $j$.

$r_{ij} = \langle u_i, v_j \rangle + \varepsilon_{ij}$, where $u_i \in \mathbb{R}^d$, $v_j \in \mathbb{R}^d$.

Prediction: $\hat{r}_{ij} = \langle u_i, v_j \rangle$

LSE: $(U, V) = \arg\min_{U \in \mathbb{R}^{n \times d}, V \in \mathbb{R}^{m \times d}} \sum_{(i, j, r_{ij}) \in \mathcal{D}} (r_{ij} - \langle u_i, v_j \rangle)^2$
Example: Matrix Factorization

Dataset $\mathcal{D}$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>?</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>5</td>
<td>5</td>
<td>?</td>
<td>3</td>
</tr>
</tbody>
</table>

$|\mathcal{D}| = O(n + m)$

Gradient Descent:

\[
\begin{align*}
u_i & \leftarrow u_i + \gamma \cdot \sum_{j:(i,j,r_{ij}) \in \mathcal{D}} (r_{ij} - \langle u_i, v_j \rangle) v_j \\
v_j & \leftarrow v_j + \gamma \cdot \sum_{i:(i,j,r_{ij}) \in \mathcal{D}} (r_{ij} - \langle u_i, v_j \rangle) u_i
\end{align*}
\]

Communication pattern reveals who rated what!

Can be quite revealing [Weinsberg, Bhagat, Ioannidis, Taft, RecSys 2012]
Example: High-Dimensional, Sparse ERM

\[ x_i \in \mathbb{R}^m \]

Labels \( y_i \in \{-1, +1\} \)

\[ \beta^* = \arg \min_{\beta \in \mathbb{R}^m} \sum_{i=1}^{n} \ell(\langle x_i, \beta \rangle, y_i) + \lambda \| \beta \|_2^2 \]

Non-zero elements: \(|\mathcal{D}| = O(n + m)\)

Gradient Descent:

\[ \beta_j \leftarrow \beta_j - \gamma \left( \sum_{i:x_{ij} \neq 0} \ell'(\langle x_i, \beta \rangle, y_i) \cdot x_{ik} + 2\lambda \beta_j \right), \quad \forall j \in \{1, \ldots, m\} \]

Communication pattern reveals features present!
Overview

- Graph-Parallel Algorithms
- Graph-Parallel Secure Evaluation
- Implementation
Naïve Data Oblivious Algorithm

Scatter data to everyone, gather from all

Total work+messages: \( O(|V| + |E|) \rightarrow O(|V|^2) \)
Our Solution

Two key ingredients:

- Sorting Networks
- Parallel Prefix Sum
Main Data Structure

\[
G(V, E)
\]

\[
\begin{array}{cccc|c|c|c}
  d(v_1) & d(v_2) & \cdots & d(v_n) & d((v_1, v_2)) & d((v_1, v_3)) & d((v_n, v_{23})) \\
  v_1 & v_2 & \cdots & v_n & v_2 & v_3 & \cdots & v_{23} \\
  v_1 & v_2 & \cdots & v_n & v_1 & v_1 & \cdots & v_n \\
\end{array}
\]

| \[V\] | \[E\] |
For each column in array:

- If it is blue/node, apply $f_A$ on top row.
$$d(v_1) \quad d(v_2) \quad \ldots \quad d(v_n) \quad d((v_1,v_2)) \quad d((v_1,v_3)) \quad \ldots \quad d((v_n,v_{23}))$$

**Sort** with respect to bottom row, prioritizing blue columns
Sort with respect to bottom row, prioritizing blue columns
Moving from **left** to **right**, copy item in third row until blue column is encountered.
Moving from **left** to **right**, copy item in third row until blue column is encountered.
S. Ioannidis, Secure Function Evaluation at Scale

For each column, apply $f_S$ if it is red.
Gather

\[ d(v) \leftarrow f_G \left( d(v), \bigcirc e \in N(v) d(e) \right) \]

<table>
<thead>
<tr>
<th>$v_1$</th>
<th>$v_1$</th>
<th>$\cdots$</th>
<th>$v_1$</th>
<th>$v_2$</th>
<th>$v_2$</th>
<th>$\cdots$</th>
<th>$v_n$</th>
<th>$\cdots$</th>
<th>$v_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d(v_1)$</td>
<td>$d((v_2, v_1))$</td>
<td>$d((v_1, v_{16}))$</td>
<td>$d(v_2)$</td>
<td>$d((v_1, v_2))$</td>
<td>$d(v_n)$</td>
<td>$d((v_n, v_{23}))$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Sort** w.r.t. middle row, giving blue/node columns priority
- **Aggregate data from right to left**, using binary operator $\circ$, until
  Encountering a blue column
- **For every column**, apply $f_G$ if it is blue
Parallel Sort

- Bitonic Sort

\[
\begin{align*}
    a_1 & = \min(a_1, a_2) \\
    a_2 & = \max(a_1, a_2)
\end{align*}
\]

Compare and swap

- Total Work: \(O(M \log^2 M)\)
- Depth: \(O(\log^2 M)\)
Parallel Left/Right Passes

- Modification on Parallel Prefix Sum

- Total Work: $O(M \log M)$
- Rounds: $O(\log M)$
Parallel Properties

$$d(v_1) \quad d((v_1, v_2)) \quad d((v_1, v_{16})) \quad d(v_2) \quad d((v_2, v_1)) \quad d(v_n) \quad d((v_n, v_{23}))$$

<table>
<thead>
<tr>
<th></th>
<th>$v_1$</th>
<th>$v_2$</th>
<th>$\cdots$</th>
<th>$v_{16}$</th>
<th>$v_2$</th>
<th>$v_1$</th>
<th>$\cdots$</th>
<th>$v_n$</th>
<th>$\cdots$</th>
<th>$v_{23}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>$v_2$</td>
<td>$\cdots$</td>
<td>$v_{16}$</td>
<td>$v_2$</td>
<td>$v_1$</td>
<td>$\cdots$</td>
<td>$v_n$</td>
<td>$\cdots$</td>
<td>$v_{23}$</td>
<td></td>
</tr>
<tr>
<td>$v_1$</td>
<td>$v_1$</td>
<td>$v_1$</td>
<td>$v_2$</td>
<td>$v_2$</td>
<td>$v_n$</td>
<td>$v_n$</td>
<td>$v_n$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$M = |V| + |E|$$
Parallel Properties

<table>
<thead>
<tr>
<th></th>
<th>$d(v_1)$</th>
<th>$d((v_1, v_2))$</th>
<th>$d((v_1, v_{16}))$</th>
<th>$d(v_2)$</th>
<th>$d((v_2, v_1))$</th>
<th>$d(v_n)$</th>
<th>$d((v_n, v_{23}))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>$v_2$</td>
<td>$\cdots$</td>
<td>$v_{16}$</td>
<td>$v_2$</td>
<td>$\cdots$</td>
<td>$v_n$</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$v_1$</td>
<td>$v_1$</td>
<td>$\cdots$</td>
<td>$v_1$</td>
<td>$v_2$</td>
<td></td>
<td>$v_n$</td>
<td></td>
</tr>
</tbody>
</table>

If processors connect in a hypercube network then:

- **Scatter, Gather**:
  - Parallel Time: $O\left(\frac{M}{p} \log^2 M\right)$
  - Total Work: $O(M \log^2 M)$

- **Apply**:
  - Parallel Time: $O(M/p)$
  - Total Work: $O(M)$

Serial execution in the clear: $O(M)$

**CoO**: $\sim \log^2 M$

**Speedup**: $\sim \frac{p}{\log^2 M}$
Overview

- Graph-Parallel Algorithms
- Graph-Parallel Secure Evaluation
- Implementation
From Data Oblivious Algorithm to SFE

High-Level Data Oblivious Program → Binary Circuit → Yao’s Garbed Circuit Protocol

ObliVM

Our Implementation: GraphSC

- Extension to ObliVM
- Supports **thread** + **cluster** parallelism

- Programmer defines $f_S$, $f_G$, $f_A$, $\circ$ and sequence of Scatter/Gather/Apply calls, GraphSC takes care of:
  - Implementing **scatter**, **gather**, and **apply**, through **sorting** and **left/right passes**
  - Translation to **binary circuit** through ObliVM
  - Garbling and Evaluation over multiple nodes forming **hypercube** network.

https://github.com/kartik1507/GraphSC
Example: PageRank over GraphSC

def PageRank(G):
    Apply \( d(v) \mapsto 1/|V| \)

repeat:
    Apply \( d(v) \mapsto d(v)/N(v) \)
    Scatter \( (x, y) \mapsto x \)
    Gather \( (x, y) \mapsto \gamma \frac{1}{|V|} x + \gamma y + \) until convergence

public class PageRankNode<T> extends GraphNode<T> {
    T[] pr;
}

public class PageRank<T> implements ParallelGadget<T> {
    @Override
    public <T> void compute(PageRankNode<T>[] graph, final CompEnv<T> env) {
        final IntegerLib<T> lib = new IntegerLib(env);
        final ArithmeticLib<T> flib = new FixedPointLib<T>(env, 40/*Width*/, 20/*Offset*/);
        // Set initial PageRank
        new SetInitialPageRankGadget<T>(env).setInputs(graph).compute();
        for (int i = 0; i < ITERATIONS; i++) {
            // 1. Write weighted PR to edges
            new Scatter<T>(env, false /*isEdgeIncoming*/) {
                @Override
                public void writeToEdge(PageRankNode<T> vertex, PageRankNode<T> edge, T cond) {
                    T[] div = flib.div(vertex.pr, vertex.l);
                    edge.pr = lib.mux(div, edge.pr, cond);
                }
            }.setInputs(graph).compute();
            // 2. Compute PR based on edges
            new Gather<T>(env, true /*isEdgeIncoming*/, new PageRankNode<T>(env)) {
                @Override
                public PageRankNode<T> aggregate(PageRankNode<T> agg, PageRankNode<T> edge) {
                    PageRankNode<T> ret = new PageRankNode<T>(env);
                    ret.pr = flib.add(agg.pr, edge.pr);
                    return ret;
                }
                @Override
                public void writeToVertex(PageRankNode<T> agg, PageRankNode<T> node) {
                    T[] d = flib.publicValue(0.15);
                    T[] dp = flib.publicValue(0.85);
                    T[] newPageRank = flib.add(flib.div(d, N), flib.multiply(dp, agg));
                    node.pr = lib.mux(node.pr, newPageRank, node.isVertex);
                }
            }.setInputs(graph).compute();
        }
    }
}

S. Ioannidis, Secure Function Evaluation at Scale
the communication with evaluators as “out-going” communication (e.g., a provider running garblers see total costs. As the number of processors increase, the “out-going” communication services (garbling or evaluating) is inter-

Figure 10a shows the communications per-processor (dividing the results number of processors), due to the bitonic sort. Figure 10b shows the expected small increase to the number of processors.

Figure 8c compares our performance with the performance of Nikolaenko.

The lines correspond to different input lengths. Plots are in a log-log scale, showing the expected small increase to the number of processors.

The optimization discussed in Sec-

The time required for one iteration. In Figure 8a, the baseline is a sequential ORAM-based baseline using two scenarios: (a) one processor simulating multiple.

Practical Optimizations.

Recall that our theoretical analysis suggests that the total amount of work is different input lengths. Plots are in a log-log scale, showing the expected small increase to the number of processors.

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 two scenarios: (a) one processor simulating multiple.

Practical Optimizations.

Recall that our theoretical analysis suggests that the total amount of work is different input lengths. Plots are in a log-log scale, showing the expected small increase to the number of processors.

The time required for one iteration. In Figure 8a, the baseline is a sequential ORAM-based baseline using two scenarios: (a) one processor simulating multiple.

Practical Optimizations.

Recall that our theoretical analysis suggests that the total amount of work is different input lengths. Plots are in a log-log scale, showing the expected small increase to the number of processors.

The time required for one iteration. In Figure 8a, the baseline is a sequential ORAM-based baseline using two scenarios: (a) one processor simulating multiple.

Practical Optimizations.

Recall that our theoretical analysis suggests that the total amount of work is different input lengths. Plots are in a log-log scale, showing the expected small increase to the number of processors.

The time required for one iteration. In Figure 8a, the baseline is a sequential ORAM-based baseline using two scenarios: (a) one processor simulating multiple.
GraphSC vs. GraphLab

Matrix Factorization, 32K ratings
Matrix Factorization using gradient descent: 1M ratings, 6K users, 4K movies

We used only 7 machines!

Time taken: ~13 hours (1 iteration)

11 days -> few hours by using more machines

7 machine cluster, 128 processors, 525 GB RAM

One run: 20 iterations - 11 days
Parallel SFE at scale is possible for a broad array of algorithms

Implementations of additional algorithms
- DNN, ERM, graph coloring, ADMM, etc.

Further acceleration over FPGAs in the datacenter

[Fang, Ioannidis, Leeser; FPGA 2017]
Thank you!


Xin Fang, Stratis Ioannidis, Miriam Leeser. “Secure Function Evaluation using an FPGA Overlay Architecture”. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017