Efficient Secure Outsourcing of Large-Scale Sparse Linear Systems of Equations

Sergio Salinas, *Member, IEEE*, Changqing Luo, *Student Member, IEEE*, Xuhui Chen, *Student Member, IEEE*, Weixian Liao, *Student Member, IEEE*, and Pan Li, *Member, IEEE*

Abstract—Solving large-scale sparse linear systems of equations (SLSEs) is one of the most common and fundamental problems in big data, but it is very challenging for resource-limited users. Cloud computing has been proposed as a timely, efficient, and cost-effective way of solving such expensive computing tasks. Nevertheless, one critical concern in cloud computing is data privacy. Specifically, clients' SLSEs usually contain private information that should remain hidden from the cloud for ethical, legal, or security reasons. Many previous works on secure outsourcing of linear systems of equations (LSEs) have high computational complexity, and do not exploit the sparsity in the LSEs. More importantly, they share a common serious problem, i.e., a huge number of memory I/O operations. This problem has been largely neglected in the past, but in fact is of particular importance and may eventually render those outsourcing schemes impractical. In this paper, we develop an efficient and practical secure outsourcing algorithm for solving large-scale SLSEs, which has low computational and memory I/O complexities and can protect clients' privacy well. We implement our algorithm on Amazon Elastic Compute Cloud, and find that the proposed algorithm offers significant time savings for the client (up to 74 percent) compared to previous algorithms.

Index Terms—Sparse linear systems of equations, cloud computing, privacy, computational complexity, memory I/O complexity

1 Introduction

We are now in the age of big data [1], [2]. We need to deal with huge data sets in many areas such as biomedicine, power systems, finance, engineering and scientific simulations, and social networks. For example, in power systems, real-time analysis like state estimation and power flow optimization involves enormous amounts of data collected from the electric grid [3]; financial firms need to process huge amounts of consumer and business data for portfolio optimization, etc.; in engineering and scientific simulations, for instance, aircraft design or meteorological simulations, we can easily have tons of data in a single simulation; and social network analysis is based on data from millions or billions of users[4], [5]. Obviously, we have massive data in all these fields, and such data needs to be stored, managed, and more importantly, computed. However, both individuals and organizations face a formidable challenge in trying to analyze such huge amounts of data in a timely and cost-effective way.

In particular, it is infeasible for users to analyze large-scale data-sets on traditional computer hardware due to its limited computing capacity and RAM (random access memory). To overcome this limitation, many governments have built supercomputers that can complete very heavy computing tasks, but have large installation and operating costs (in the range of tens of millions of dollars or even higher) and usually have restricted access. Besides, even an in-house computing cluster can be very expensive and may still lack enough memory and computing power to analyze large-scale data sets.

This challenge has attracted significant attention from industry, academia and governments. Recently, cloud computing has been proposed as an efficient, and cost-effective way for resource-limited users to analyze large-scale data sets. In this computing paradigm, cloud clients outsource their computing tasks to a cloud server [6], [7], [8], [9], [10], which contains a large amount of computing resources and offers them on a on-demand and pay-per-use basis [11]. In cloud computing, clients share the cloud resources with each other, and avoid purchasing, installing, and maintaining sophisticated and expensive computing hardware and software.

Nevertheless, one critical concern in cloud computing is data privacy. To be more prominent, in many cases, clients' data are very sensitive and should be hidden from the cloud for ethical, legal, or security reasons. For example, in power flow state estimation that can be solved by least squares problems, a power company's data may disclose the topology of the system, thus enabling attacks to the electric grid [12]; in portfolio optimization, we may have quadratic or nonlinear programming problems that could reveal companies' proprietary investment strategies. Therefore, in order for people to really adopt cloud computing, we have

- *S. Salinas is with the Department of Electrical Engineering and Computer Science, Wichita State University, Wichita, KS 67260. E-mail: salinas@cs.wichita.edu.*
- *C. Luo, X. Chen, W. Liao, and P. Li are with the Department of Electrical Engineering and Computer Science, Case Western Reserve University, Cleveland, OH 44106. E-mail: {changqing.luo, xuhui.chen2, weixian.liao, lipan}@case.edu.*

Manuscript received 31 Dec. 2015; revised 1 Feb. 2017; accepted 24 Feb. 2017. Date of publication 7 Mar. 2017; date of current version 7 Mar. 2018.
Recommended for acceptance by G. Karypis.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TBDATA.2017.2679760

to design new tools and technologies that allow clients to outsource their computations to the cloud while preserving the privacy of their data. Moreover, the fact that users usually lack computing and memory storage resources limits the complexity of the operations that they can perform to hide their data, which makes secure outsourcing an even more challenging problem.

We notice that many problems that involve large-scale sensitive data are fundamentally based on solving sparse linear systems of equations (SLSEs) of the form $\mathbf{Ax} = \mathbf{b}$. For example, as mentioned above, in image processing, power flow estimation, and portfolio optimization, least squares problems or quadratic programming problems can be reformulated as SLSEs. In addition, in social network analysis and web search engine design, the eigenvector centrality problem is naturally posed as an SLSE which contains private information such as social network users' data and innovative search engine algorithms. Therefore, in this paper, we focus on efficiently and securely solving large-scale SLSEs, one of the most common and fundamental problems in big data.

Some previous works on securely outsourcing computing tasks to the cloud could be used to solve SLSEs. However, they have high computing requirements. Specifically, in [13], Gennaro et al. employ fully homomorphic encryption (FHE) to securely offload computations to the cloud. FHE offers theoretical privacy guarantees, but it is a computationally intensive operation, and large-scale computations based on FHE ciphertexts are very expensive, even for the cloud. Wang et al. [14], [15] design methods to privately outsource a linear programming problem. A client may employ such methods to find the solution to an LSE by requesting the cloud to solve a special linear program. Unfortunately, to protect data privacy, the client needs to perform a matrix-matrix multiplication that is prohibitively expensive because this operation has computational complexity of $\mathcal{O}(\rho)$ where $n^2 < \rho \leq Mn$ (for $n \times n$ matrices with M non-zero elements).

Recently a few secure outsourcing algorithms have been developed specifically for solving LSEs. Lei et al. [16] and Atallah et al. [17] design secure matrix inversion algorithms that use matrix permutations to preserve data privacy. To find the solution to an LSE, a client needs to perform operations with computational complexity of $\mathcal{O}(n^2)$ and $\mathcal{O}(M)$, respectively. Chen et al. [18] outsource an LSE to the cloud by employing matrix permutations. Wang et al. [19] develop an iterative algorithm to solve LSEs, where a client transforms and encrypts the coefficient matrix using homomorphic encryption, and the cloud carries out computations on ciphertexts. Specifically, the client needs to perform two matrix-vector multiplications, which require $\mathcal{O}(M)$ floating-point operations (flops), and $\mathcal{O}(M)$ homomorphic encryptions. Note that performing homomorphic encryptions has high computational complexity ($\mathcal{O}(\log_2 e)$ flops per encrypted value, were e is the key size). Although it is proposed that the client could outsource this computation to a trusted third-party, it may not always exist. The use of homomorphic encryption forces the cloud to operate on ciphertexts, which then has to use specialized linear algebra software. Besides, the proposed algorithm only works for LSEs whose coefficient matrices are diagonally dominant, and the privacy will be compromised if the number of iterations approaches or exceeds n. Later on, Chen et al. [20] also propose similar solutions to outsourcing linear programs and LSEs while preserving users' privacy. We notice that most such works' computational complexity is still high,[1] and they do not exploit the sparsity in the LSEs to potentially reduce computational complexity.

More importantly, previous works [16], [17], [18], [19], [20] impose a large burden of memory I/O operations on the client. This problem has been largely neglected in the previous secure outsourcing algorithm design. But we emphasize that the number of times an algorithm accesses local data is of particular importance for outsourcing a large-scale SLSE and may render the outsourcing algorithm impractical. The reason is as follows. Most often a client lacks enough RAM memory to store a large-scale sparse matrix completely at once. So, instead of working on RAM memory directly, as is the case with smaller matrices, the client can only load a small section of the large-scale sparse matrix at a time and write the results to external memory when it is done. However, reading and writing operations from and into external memory have a very high latency compared to the same operations in RAM. For example, our experiments show that reading a matrix once with 3.7×10^6 non-zero elements and size 1.2 GB on a laptop that has 4 GB RAM and a hard disk at 5400RPM would take about 10 minutes. Therefore, any practical algorithm for large-scale SLSEs should only incur as small the number of memory I/O operations for the client as possible. To better capture the special memory I/O requirement of large-scale SLSEs, we propose a new definition of *"external memory I/O complexity"*, which is the number of values that are read/written from/into external memory. In this paper, we call it *"memory I/O complexity"* for brevity. Previous works have very high I/O complexity. For example, in [16], [17], the client needs to access a large-scale dense matrix at least twice since the inversion operation does not preserve the sparsity of the original LSE. In [19], the client has to read a large-scale sparse matrix at least five times. Both algorithms may take an unacceptably long time due to the latency of the large number of I/O operations in practice.

Aiming to reduce both computational and memory I/O complexities, in this paper, we develop an efficient and practical secure outsourcing algorithm for solving large-scale SLSEs. Specifically, to protect its data privacy while preserving sparsity, the client transforms the coefficient matrix $\mathbf{A}_{m\times n}$ into matrix $\hat{\mathbf{A}}_{m\times n}$, by adding a sparse matrix with random values and randomly permuting its rows and columns. We prove that matrix $\hat{\mathbf{A}}$ has the property of computational indistinguishability under a chosen-plaintext attack (CPA). Then, based on the conjugate gradient method, the client finds the solution vector $\mathbf{x}$ iteratively with the help of the cloud. Since the client delegates expensive matrix-vector operations to the cloud, it has computational complexity of $\mathcal{O}(\hat{M}')$, where

1. Our previous work [21] proposes an efficient secure outsourcing scheme for solving general LSEs that has much lower complexity than previous works, but it does not aim at sparse LSEs.

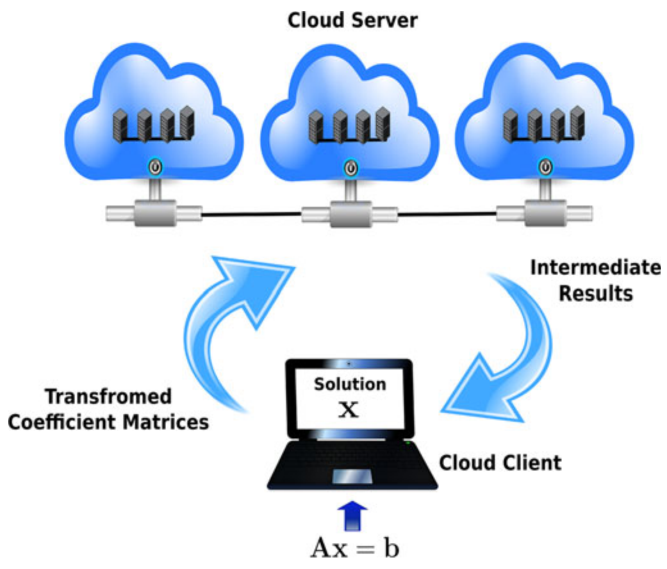


Fig. 1. A secure architecture for outsourcing LSEs.

$0 \leq \hat{M}' \leq n^2$ is an arbitrary number chosen by the user that determines the number of non-zero elements in the transformed matrix sent to the cloud. The algorithm preserves the privacy of the client by letting the cloud operate on the transformed matrix and some intermediate values, rather than on $\mathbf{A}$, $\mathbf{x}$, or $\mathbf{b}$. Moreover, both the client and the cloud are able to exploit the sparsity of the transformed matrix for computational savings, and use traditional linear algebra software, which avoids the costly exponentiations required for ciphertext-based operations as in previous works like [13] and [19].

We summarize our main contributions as follows.

- We develop an efficient and practical algorithm to securely outsource the computation of large-scale sparse LSEs
- The proposed algorithm requires operations with low computational and memory I/O complexities at the client and at the cloud. In particular, the computational complexity is $\mathcal{O}(\hat{M}')$ and the memory I/O complexity is $2\hat{M}' + 4M' + 2M + 8n$ at the client, where M' is the number of non-zero elements in $\mathbf{A}^\top\mathbf{A}$. Besides, our proposed algorithm allows the cloud to take advantage of the sparsity of the SLSE for computational savings. We compare the complexities of our algorithm at both the client and the cloud with those of previous algorithms and find that our algorithm is much more efficient.
- We show that the cloud is unable to obtain any information about the client's SLSE.
- We implement our algorithm on Amazon Elastic Compute Cloud (EC2) and a laptop. We find that the proposed algorithm offers significant time savings for the client (up to 74 percent) compared to previous algorithms.

2 Problem Formulation

2.1 System Architecture

We consider an asymmetric two-party computing architecture as shown in Fig. 1, where a cloud client (CC) is resource-limited and a remote cloud server (CS) has abundant computing resources. The CC intends to solve a large-scale computing task, but cannot complete it on its own. So the CC offloads the most expensive computations to the CS and collaborates with it to find the solution to the task. In this work, we concentrate on the computing task of finding the solution to a large-scale SLSE:

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{1}$$

where $\mathbf{A} \in \mathbb{R}^{m\times n}$ $(m \geq n)$ is a full-rank coefficient matrix with M non-zero elements (for $n \leq M \leq mn$), $\mathbf{x} \in \mathbb{R}^{n\times 1}$ is the solution vector, and $\mathbf{b} \in \mathbb{R}^{m\times 1}$ is the constant vector.

To reduce the storage complexity of the coefficient matrix $\mathbf{A}$, the client uses a sparse matrix representation called compressed sparse column (CSC). Particularly, in CSC, instead of storing $\mathbf{A}$ as a two-dimensional vector, the client stores the non-zero elements in $\mathbf{A}$ in a top-to-bottom left-to-right order into vector `val` $\in \mathbb{R}^{M\times 1}$, and uniquely determines their positions in $\mathbf{A}$ by using two auxiliary vectors: the row index `row_index` $\in \mathbb{R}^{M\times 1}$ and the column index `column_pointer` $\in \mathbb{R}^{n\times 1}$ vectors. Elements in `row_index` indicate the row in $\mathbf{A}$ each element of `val` is located in, while `column_pointer` stores the indices of the elements in `val` that start a column in $\mathbf{A}$. By using the CSC format, the client can reduce its storage and memory I/O requirements as long as $2M + n < mn$.

2.2 Threat Model

We assume a malicious threat model for the CS. That is, the CS tries to extract information from the CC's data and from the results of its own computations, and may attempt to deviate from the proposed protocols and return erroneous results.

To enable the CC to securely delegate computing tasks to the CS, the data that the CC shares with the CS should appear random. This notion of privacy is known as computational indistinguishability [22]. We identify two types of private data contained in a matrix: the values of its non-zero elements and the positions of its non-zero elements. In what follows we define computational indistinguishability for both types of data.

Definition 1. *Computational Indistinguishability: Two probability ensembles $X = \{X_s\}_{s\in\mathbb{N}}$ and $Y = \{Y_s\}_{s\in\mathbb{N}}$, are computationally indistinguishable if for every probabilistic polynomial time distinguisher D there exists a negligible function $\mu(\cdot)$ such that*

$$\left|Pr[D(X_s) = 1] - Pr[D(Y_s) = 1]\right| < \mu, \tag{2}$$

where the notation $D(X_s)$ (or $D(Y_s)$) means that x (or y) is chosen according to distribution X_s (or Y_s) and then $D(x)$ (or $D(y)$) is run.

Moreover, this definition can be extended to the case where a distinguisher D has access to multiple samples of the vectors X and Y, i.e., when comparing two matrices.

Definition 2. *Let $\mathbf{R} \in \mathbb{R}^{m\times n}$ be a random matrix with entries in its jth column sampled from a uniform distribution with interval $[-R_j, R_j]$ $\forall j \in [1, n]$. Matrices $\mathbf{R}$ and $\mathbf{Q}$ are computationally indistinguishable in value if for any probabilistic* *polynomial time distinguisher $D(\cdot)$ there exists a negligible function $\mu(\cdot)$ such that*

$$|P[D(r_{i,j}) = 1] - Pr[D(q_{i,j}) = 1]| \leq \mu \qquad \forall\ i, j, \tag{3}$$

where $r_{i,j}$ is the element in the ith row and jth column of **R**, *and $q_{i,j}$ is the element in the ith row and jth column of* **Q**. *Distinguisher $D(\cdot)$ outputs 1 when it identifies the input as a non-uniform distribution in the range $[-R_j, R_j]$, and zero otherwise.*

Definition 2 refers to the inability of an attacker to tell apart the elements of a matrix **Q** from the elements of a random matrix **R**. However, the position of the elements in **Q**, (i.e., **Q**'s structure), may also contain private information that should be hidden from the CS. To protect a matrix's structure, we can make it appear random by permuting the rows and columns. In particular, we permute the rows and the columns of a matrix by taking advantage of pseudorandom functions as defined below.

Definition 3. *Let F be a function and f a truly random function. We say F is a pseudorandom function if for all probabilistic polynomial-time distinguishers D, there exists a negligible function μ such that*

$$|Pr[D^F(1^n) = 1] - Pr[D^f(1^n) = 1]| \leq \mu. \tag{4}$$

Distinguishers D^F and D^f have oracle access to functions F and f, respectively.

We give the definition of secure permutation below.

Definition 4. *We say that a permutation scheme has indistinguishable permutations under a chosen-plaintext attack (or is CPA-secure in structure) if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function μ, such that the probability of distinguishing two permutations in a CPA indistinguishability experiment is less than $1/2 + \mu$.*

3 A Privacy and Sparsity Preserving Matrix Transformation

Before delving into the details about our proposed algorithm for outsourcing large-scale SLSEs, we first present a privacy and sparsity preserving matrix transformation scheme.

To delegate a computing task to the CS, the CC first needs to perform some computations on its data. These computations should require a moderate effort from the CC, hide the data from the CS, and allow the CS to return a meaningful result. To this end, we design an efficient privacy and sparsity preserving matrix transformation that offers computational indistinguishability, that is, every probabilistic polynomial time algorithm is unable to differentiate between the transformed matrix and a random matrix with non-negligible probability. In particular, the proposed matrix transformation conceals the values of the non-zero elements of **A** by adding a carefully designed random matrix, and hides its structure by randomly permuting its rows and columns. Moreover, the proposed transformation allows the CS to exploit the sparsity of **A** by not making it a dense matrix in the computations. In the rest of this section, we explain in detail the proposed transformation, and show that the transformed matrix is indeed computationally indistinguishable from a random one.

3.1 Privacy-Preserving Matrix Addition

We first present a matrix addition scheme that can transform matrix **A** into a matrix that is computationally indistinguishable from a random matrix, but does not preserve sparsity.

Specifically, to hide all the elements of **A**, the CC performs the following matrix addition:

$$\bar{\mathbf{A}} = \mathbf{A} + \bar{\mathbf{Z}}, \tag{5}$$

where $\bar{\mathbf{Z}} \in \mathbb{R}^{m\times n}$ is a random matrix, and $\bar{a}_{i,j} = a_{i,j} + \bar{z}_{i,j}$ (for $i \in [1, m],\ j \in [1, n]$). We assume that the values of matrix **A** are within the range $[-K, K]$, where $K = 2^l$ ($l > 0$) is a positive constant.

To reduce the CC's computational complexity, the random matrix **Z** is formed by a vector outer-product, i.e.,

$$\bar{\mathbf{Z}} = \bar{\mathbf{u}}\bar{\mathbf{v}}^\top, \tag{6}$$

where $\bar{\mathbf{u}} \in \mathbb{R}^{m\times 1}$ is a vector of uniformly distributed random variables with probability density functions as follows: $f_U(\bar{u}_i)$ is equal to $1/2c$ for $-c < \bar{u}_i < c$ and 0 otherwise, where $c = 2^p$ ($p > 0$) is a positive constant, and $i \in [1, m]$. Vector $\bar{\mathbf{v}} \in \mathbb{R}^{n\times 1}$ is a vector of arbitrary positive constants ranging from 2^l and 2^{l+q} ($q > 0$).

Note that element $\bar{z}_{i,j} = \bar{u}_i\bar{v}_j$ (for $i \in [1, m],\ j \in [1, n]$), is the product of a random variable and a positive constant. Thus, $\bar{z}_{i,j}$ is also a random variable with its probability density function defined as [23]

$$f_{\bar{Z}}(\bar{z}_{i,j}) = \begin{cases} \frac{1}{2L_j} & -L_j < \bar{z}_{(i,j)} < L_j \\ 0 & \text{otherwise,} \end{cases}$$

where $L_j = cv_j$ (for $j \in [1, n]$) and hence is between 2^{p+l} and 2^{p+l+q}. We can now arrive at a theorem about the computational indistinsguishability between $\bar{\mathbf{A}}$ and a matrix with columns filled with values taken from uniform distributions.

Theorem 1. *Let $\mathbf{R} \in \mathbb{R}^{m\times n}$ be a random matrix with entries in column j sampled from a uniform distribution on the interval $[-L_j, L_j]$ for $j \in [1, n]$. Matrices* **R** *and* $\bar{\mathbf{A}}$ *are computationally indistinguishable.*

Proof. According to Definition 2, we need to show that $r_{i,j}$ and $\bar{a}_{i,j}$ (for $i \in [i, m], j \in [1, n]$) are computationally indistinguishable for matrices **R** and $\bar{\mathbf{A}}$ to be computationally indistinguishable. In particular, we show that any probabilistic polynomial time distinguisher D cannot distinguish $\bar{a}_{i,j}$ from $r_{i,j}$ for any $i \in [1, m], j \in [1, n]$, except with non-negligible success probability.

Recall that values from **R** and **A** are in the intervals $[-L_j, L_j]$ and $[-K, K]$, respectively. Thus, we have that $\bar{a}_{i,j} \in [-K - L_j, K + L_j]$, and hence $r_{i,j}, \bar{a}_{i,j} \in [-2^\kappa, 2^\kappa]$ where $\kappa = p + l + q + 1$. The best strategy for distinguisher D when presented with a sample $x = \bar{a}_{i,j}$ is to return $b \leftarrow \{0, 1\}$ with equal probability if $-L_j \leq x \leq L_j$, and 1 if $x < -L_j$ or $x > L_j$. Therefore, when $x = \bar{a}_{i,j}$, we have that the success probability of the distinguisher is given by

$$\begin{aligned}&Pr[D(\bar{a}_{i,j})=1]\\&=\frac{1}{2}Pr[-L_j\le \bar{a}_{i,j}\le L_j]\\&\quad+Pr[\bar{a}_{i,j}<-L_j]+Pr[\bar{a}_{i,j}>L_j]\\&=\frac{1}{2}\big(1-Pr[\bar{a}_{i,j}<-L_j]-Pr[\bar{a}_{i,j}>L_j]\big)\\&\quad+Pr[\bar{a}_{i,j}<-L_j]+Pr[\bar{a}_{i,j}>L_j]\end{aligned}$$

where

$$\begin{aligned}Pr[\bar{a}_{i,j}>L_j]&=Pr[a_{i,j}+\bar{z}_{i,j}>L_j]\\&=Pr[\bar{z}_{i,j}>L_j-a_{i,j}]\\&\le Pr[\bar{z}_{i,j}>L_j-K]\\&=\frac{K}{2L_j}.\end{aligned}$$

Similarly, we find that $Pr[\bar{a}_{i,j}<-L_j]\le\frac{K}{2L_j}$. Consequently, we have that the probability of success for distinguisher D, when $x=\bar{a}_{i,j}$, is bounded as follows:

$$0<Pr[D(\bar{a}_{i,j}=1)]\le\frac{1}{2}+\frac{K}{2L_j}.$$

On the other hand, if $x=r_{i,j}$, we can obtain that $Pr[D(r_{i,j})=1]=\frac{1}{2}$.

According to equation (3), for any $i\in[1,m]$, $j\in[1,n]$ we get that

$$|Pr[D(\bar{a}_{i,j})=1]-Pr[D(r_{i,j})=1]|\le\frac{K}{2L_j}.$$

Note that $K=2^l$ and $L_j\in[2^{p+l},2^{p+l+q}]$. Thus, we have

$$\mu(\kappa)=\frac{K}{2L_j}\le\frac{2^l}{2^{p+l}}=\frac{1}{2^p}=\frac{1}{2^{\kappa-l-q-1}}$$

which is a negligible function. By union bound, it concludes the proof. □

3.2 Privacy and Sparsity Preserving Matrix Addition

In the matrix transformation described above, the CC masks the values of $\mathbf{A}$ unconditionally, which results in $\bar{\mathbf{A}}$ having a dense structure. However, it is undesirable for the transformed matrix to have a dense structure because it introduces unnecessary computations when we solve the SLSE. Next, we develop a both privacy and sparsity preserving matrix addition scheme based on matrix partitions.

Specifically, to achieve a privacy and sparsity preserving matrix transformation, the CC partitions $\mathbf{A}$ into sub-matrices of equal sizes and only masks those sub-matrices that contain at least one non-zero element. In particular, the CC divides matrix $\mathbf{A}$ into smaller matrices $\mathbf{a}_{\gamma,\theta}\in\mathbb{R}^{\alpha\times\beta}$ for $\gamma\in[1,m/\alpha]$, $\theta\in[1,n/\beta]$, where m/α and n/β are integers.[2]

2. If m/α (or n/β) is not an integer, the CC can form an additional sub-matrix row (column) of size $m \bmod \alpha$ (or $n \bmod \beta$) to cover the remaining rows (columns).

Matrix $\mathbf{a}_{\gamma,\theta}$ contains elements $a_{i,j}$ for $i\in[(\gamma-1)\alpha+1,\gamma\alpha]$, $j\in[(\theta-1)\beta+1,\theta\beta]$.

Similar to Equation (5), the CC transforms a matrix through a matrix addition as follows:

$$\tilde{\mathbf{A}}=\mathbf{A}+\tilde{\mathbf{Z}},$$

where matrix $\tilde{\mathbf{Z}}\in\mathbb{R}^{m\times n}$ is a linear combination of sparse random matrices, i.e.,

$$\tilde{\mathbf{Z}}=\sum_{(\gamma,\theta)\in\mathcal{Z}}\tilde{\mathbf{z}}_{\gamma,\theta},$$

where $\mathcal{Z}$ is defined as follows

$$\mathcal{Z}=\{(\gamma,\theta)\mid\exists a_{i,j}\in\mathbf{a}_{\gamma,\theta}:a_{i,j}\neq0\},$$

and matrix $\tilde{\mathbf{z}}_{\gamma,\theta}\in\mathbb{R}^{m\times n}$ for $(\gamma,\theta)\in\mathcal{Z}$ is defined as

$$\tilde{\mathbf{z}}_{\gamma,\theta}=\tilde{\mathbf{u}}_\gamma\tilde{\mathbf{v}}_\theta^\top.$$

The CC forms vector $\tilde{\mathbf{u}}_\gamma\in\mathbb{R}^{m\times1}$ by setting $\bar{\mathbf{u}}$ as in equation (6), copying elements $\bar{u}_i$ for $i\in[(\gamma-1)\alpha+1,\gamma\alpha]$, and padding the remaining entries with zeros, i.e.,

$$\tilde{\mathbf{u}}_\gamma=[0\ \dots\ \bar{u}_{(\gamma-1)\alpha+1}\ \dots\bar{u}_{\gamma\alpha}\dots0]$$

Vector $\tilde{\mathbf{v}}_\theta\in\mathbb{R}^{n\times1}$ can be found in a similar way

$$\tilde{\mathbf{v}}_\theta=[0\ \dots\ \bar{v}_{(\theta-1)\beta+1}\ \dots\bar{v}_{\theta\beta}\dots0].$$

We now arrive at a theorem about the computational indistinguishability between submatrices $\tilde{\mathbf{a}}_{\gamma,\theta}=\mathbf{a}_{\gamma,\theta}+\tilde{\mathbf{z}}'_{\gamma,\theta}$ for $(\gamma,\theta)\in\mathcal{Z}$ and a random matrix with entries sampled from a uniform distribution. Note that matrix $\tilde{\mathbf{z}}'_{\gamma,\theta}$ contains elements in rows $i\in[(\gamma-1)\alpha+1,\gamma\alpha]$ and columns $j\in[(\theta-1)\beta+1,\theta\beta]$ of matrix $\tilde{\mathbf{z}}_{\gamma,\theta}$.

Theorem 2. *Let $\mathbf{r}\in\mathbb{R}^{\alpha\times\beta}$ be a random matrix with entries in column j sampled from a uniform distribution with interval $[-L_j,L_j]$ for $j\in[(\theta-1)\beta+1,\theta\beta]$. Matrices $\mathbf{r}$ and $\tilde{\mathbf{a}}_{\gamma,\theta}\in\mathbb{R}^{\alpha\times\beta}$ for $\gamma\in[1,m/\alpha]$ $\theta\in[1,n/\beta]$ are computationally indistinguishable.*

Proof. The proof follows a similar procedure to that in the proof for Theorem 1. □

The choice of submatrix dimensions, α and β, determines the number of additional non-zero elements in $\tilde{\mathbf{A}}$ compared to $\mathbf{A}$, the amount of extra computations needed to solve the LSE, and the number of zero elements that are revealed to the CS as well.

3.3 Privacy-Preserving Matrix Permutation

The privacy and sparsity preserving random matrix addition in Section 3.2 may reveal some of the zero elements and their positions, which are also part of the CC's private information. Thus, to hide the structure of coefficient matrix $\mathbf{A}$, we propose to let the CC randomly permutate the rows and columns of $\tilde{\mathbf{A}}$.

To randomly permute $\tilde{\mathbf{A}}$'s row index vector $\mathbf{q}\in\mathbb{Z}^{+m}$, we compute the following

$$\mathbf{q}'=\mathcal{M}(\mathbf{q}),\quad\hat{\mathbf{q}}'=F(\mathbf{r},\mathbf{q}'),\quad\hat{\mathbf{q}}=\mathcal{M}^{-1}(\hat{\mathbf{q}}'),\qquad(7)$$

where $\mathcal{M} : \mathbb{Z}^{+m} \rightarrow \{0,1\}^w$ ($w = \lceil \log_2 m! \rceil$) is a function that maps an index vector to a bit string, $F : \{0,1\}^w \times \{0,1\}^w \rightarrow \{0,1\}^w$ is a function with two inputs, $\mathbf{r} \in \{0,1\}^w$ is a random bit string, and $\mathcal{M}^{-1} : \{0,1\}^w \rightarrow \mathbb{Z}^{+m}$ is the inverse of $\mathcal{M}$. We denote these computations as

$$Perm^F(\mathbf{r}, \mathbf{q}) = \hat{\mathbf{q}}. \tag{8}$$

Similarly, we can denote by $Perm^F(\mathbf{r}', \mathbf{c})$ the random permutation of the column index $\mathbf{c} \in \mathbb{Z}^{+n}$ of $\tilde{\mathbf{A}}$, where $\mathbf{r}' \in \{0,1\}^{w'}$ ($w' = \lceil \log_2 n! \rceil$) is a random bit string. To apply the permutations $Perm^F(\mathbf{r}, \mathbf{q})$ and $Perm^F(\mathbf{r}', \mathbf{c})$ to matrix $\tilde{\mathbf{A}}$, the CC performs the following multiplications:

$$\hat{\mathbf{A}} = \mathbf{P}\tilde{\mathbf{A}}\mathbf{T}, \tag{9}$$

where $\mathbf{P} \in \mathbb{R}^{m\times m}$ and $\mathbf{T} \in \mathbb{R}^{n\times n}$ are permutation matrices and their elements are defined by

$$\begin{aligned} p_{i,j} &= \delta_{\pi(i),j}, \quad \text{for } i \in [1,m], j \in [1,m] \\ t_{i,j} &= \delta_{\pi'(i),j}, \quad \text{for } i \in [1,n], j \in [1,n], \end{aligned}$$

where i and j are the row and column indexes, respectively. Besides, the Kronecker delta function is given by

$$\delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j, \end{cases}$$

and the function $\pi(\cdot)$ maps an original index i to the corresponding permuted index, i.e., $\pi(i) = \hat{q}_i$ (for $i \in [1,m]$) and $\pi'(i) = \hat{c}_i$ (for $i \in [1,n]$).

The CC recovers matrix $\tilde{\mathbf{A}}$ from the permuted matrix through the following operation

$$\tilde{\mathbf{A}} = \mathbf{P}^\top \hat{\mathbf{A}} \mathbf{T}^\top,$$

where $\top$ denotes the transpose operation. This result is due to the fact that the permutation matrices are orthogonal, i.e, $\mathbf{P}^\top\mathbf{P} = \mathbf{I}$ and $\mathbf{T}\mathbf{T}^\top = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix.

We now arrive at a theorem about the computational indistinguishability of the structure of matrix $\hat{\mathbf{A}}$.

Theorem 3. *If $F(\cdot,\cdot)$ is a pseudorandom function, then the row and column permutations described above are computationally indistinguishable in structure under a CPA.*

Proof. Please refer to Appendix A in the online supplemental material for the detailed proof, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TBDATA.2017.2679760. □

Note that after the privacy-preserving matrix permutation, the resulted matrix has the property of computationally indistinguishability, particularly, the non-zero elements in any column of the matrix are computationally indistinguishable from a uniform distribution and the non-zero elements' positions are computationally indistinguishable under a CPA. We can clearly see that after the proposed matrix addition and matrix permutation, we can successfully transform the original matrix $\mathbf{A}$ into a new matrix $\hat{\mathbf{A}}$ that protects both the values of the non-zero elements and the positions of them in $\mathbf{A}$.

4 Secure Outsourcing of Large-Scale SLSEs

In this section, we develop a practical and light-weight algorithm to securely outsource a large-scale SLSE to the CS based on the conjugate gradient method (CGM).

4.1 The Conjugate Gradient Method

We notice that solving an SLSE in $\mathbf{A}'\mathbf{x} = \mathbf{b}$ is equivalent to solving the following unconstrained quadratic program

$$\min f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}'\mathbf{x} - \mathbf{b}'\mathbf{x}, \tag{10}$$

when $\mathbf{A}'$ is nonsingular, symmetric and positive definite with M' non-zero elements [24]. Therefore, we can use the CGM algorithm that solves the above optimization problem to solve the SLSE.

Specifically, as any gradient directions (GD) method, the CGM employs a set of vectors $\mathcal{P} = \{\mathbf{p}_0, \mathbf{p}_1, \dots \mathbf{p}_n\}$ that are conjugate with respect to $\mathbf{A}'$, that is, at iteration k the following condition is met:

$$\mathbf{p}_k^\top \mathbf{A}'\mathbf{p}_i = 0, \text{ for } i = 0, \dots, k-1. \tag{11}$$

Using the conjugacy property of vectors in $\mathcal{P}$, we can find the solution in at most n steps by computing a sequence of solution approximations as follows:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k, \tag{12}$$

where α_k is the one-dimensional minimizer of (10) along $\mathbf{x}_k + \alpha_k\mathbf{p}_k$. The minimizer α_k can be found by setting (10) to zero and taking its gradient when $\mathbf{x} = \mathbf{x}_{k+1}$

$$\nabla f(\mathbf{x}_{k+1}) = \mathbf{A}'\mathbf{x}_{k+1} - \mathbf{b}' = 0. \tag{13}$$

By replacing $\mathbf{x}_{k+1}$ with (12) and multiplying by $\mathbf{p}_k^\top$ from the left, we get

$$\alpha_k = \frac{-\mathbf{r}_k^\top \mathbf{p}_k}{\mathbf{p}_k^\top \mathbf{A}'\mathbf{p}_k}, \tag{14}$$

where $\mathbf{r}_k = \mathbf{A}'\mathbf{x}_k - \mathbf{b}'$ is called the residual.

Moreover, we can find the residual iteratively based on (12) as follows:

$$\begin{aligned} \mathbf{r}_{k+1} &= \mathbf{A}'\mathbf{x}_{k+1} - \mathbf{b}' \\ &= \mathbf{A}'(\mathbf{x}_k + \alpha_k\mathbf{p}_k) - \mathbf{b}' = \mathbf{r}_k + \alpha_k\mathbf{A}'\mathbf{p}_k. \end{aligned} \tag{15}$$

Efficiently finding the set of conjugate vectors $\mathcal{P}$ is a major challenge in GD methods. The CGM algorithm offers an efficient way of finding $\mathcal{P}$ that has low storage and computational complexity. In particular, the CGM finds a new conjugate vector $\mathbf{p}_{k+1}$ at iteration k by a linear combination of the negative residual, i.e., the steepest descent direction of $f(\mathbf{x})$, and the current conjugate vector $\mathbf{p}_k$, that is,

$$\mathbf{p}_{k+1} = -\mathbf{r}_{k+1} + \beta_{k+1}\mathbf{p}_k, \tag{16}$$

where β_{k+1} is chosen in such a way that $\mathbf{p}_{k+1}^\top$ and $\mathbf{p}_k$ meet condition (11). By multiplying $\mathbf{p}_k^\top\mathbf{A}'$ from the left in (16), we get

$$\mathbf{p}_k^\top \mathbf{A}'\mathbf{p}_{k+1} = -\mathbf{p}_k^\top \mathbf{A}'\mathbf{r}_{k+1} + \mathbf{p}_k^\top \mathbf{A}'\beta_{k+1}\mathbf{p}_k,$$

which leads to

$$\beta_{k+1} = \frac{\mathbf{p}_k^\top \mathbf{A}'(\mathbf{p}_{k+1} + \mathbf{r}_{k+1})}{\mathbf{p}_k^\top \mathbf{A}'\mathbf{p}_k}.$$

Since as mentioned above $\mathbf{p}_k^\top$ and $\mathbf{p}_{k+1}^\top$ are conjugate with respect to $\mathbf{A}'$, we have $\mathbf{p}_k^\top \mathbf{A}'\mathbf{p}_{k+1} = 0$. Note that $\mathbf{p}_k^\top \mathbf{A}'\mathbf{r}_{k+1}$ is a scalar and $\mathbf{A}'$ is symmetric. Thus, we have

$$\beta_{k+1} = \frac{\mathbf{r}_{k+1}^\top \mathbf{A}'\mathbf{p}_k}{\mathbf{p}_k^\top \mathbf{A}'\mathbf{p}_k}. \tag{17}$$

Moreover, since $\mathbf{x}_k$ minimizes $f(\mathbf{x})$ along $\mathbf{p}_k$, it can be shown that $\mathbf{r}_k^\top \mathbf{p}_i = 0$ for $i = 0, 1, \dots, k-1$ [24]. Using this fact and equation (16), a more efficient computation for (14) can be found, namely,

$$\alpha_k = \frac{-\mathbf{r}_k^\top(-\mathbf{r}_k + \beta_k \mathbf{p}_{k-1})}{\mathbf{p}_k^\top \mathbf{A}'\mathbf{p}_k} = \frac{\mathbf{r}_k^\top \mathbf{r}_k}{\mathbf{p}_k^\top \mathbf{A}'\mathbf{p}_k}.$$

Similarly, using (15), we can find a more efficient formulation for β_{k+1}. First, we replace $\mathbf{A}'\mathbf{p}_k$ with $\frac{1}{\alpha}(\mathbf{r}_{k+1} - \mathbf{r}_k)$ in (17) to get

$$\beta_{k+1} = \frac{\mathbf{r}_{k+1}^\top(\mathbf{r}_{k+1} - \mathbf{r}_k)}{\mathbf{p}_k^\top(\mathbf{r}_{k+1} - \mathbf{r}_k)}.$$

Then, using the fact that $\mathbf{p}_k^\top \mathbf{r}_{k+1} = 0$ and $\mathbf{r}_{k+1}^\top \mathbf{r}_k = 0$ [24], we find that $\beta_{k+1} = -\frac{\mathbf{r}_{k+1}^\top \mathbf{r}_{k+1}}{\mathbf{p}_k^\top \mathbf{r}_k}$. By replacing $\mathbf{p}_k$ with $-\mathbf{r}_k + \beta_k \mathbf{p}_{k-1}$ above, and applying $\mathbf{p}_{k-1}^\top \mathbf{r}_k = 0$, we get

$$\beta_{k+1} = \frac{\mathbf{r}_{k+1}^\top \mathbf{r}_{k+1}}{\mathbf{r}_k^\top \mathbf{r}_k}.$$

To summarize the above, the CGM algorithm is as follows. At iteration $k = 0$, we have

$$\mathbf{r}_0 = \mathbf{A}'\mathbf{x}_0 - \mathbf{b}' \tag{18}$$

$$\mathbf{p}_0 = -\mathbf{r}_0 \tag{19}$$

$$k = 0 \tag{20}$$

and at iteration $k \geq 0$ we have the following iterative equations:

$$\alpha_k = \frac{\mathbf{r}_k^\top \mathbf{r}_k}{\mathbf{p}_k^\top \mathbf{A}'\mathbf{p}_k} \tag{21}$$

$$\mathbf{r}_{k+1} = \mathbf{r}_k + \alpha_k \mathbf{A}'\mathbf{p}_k \tag{22}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \tag{23}$$

$$\beta_{k+1} = \frac{\mathbf{r}_{k+1}^\top \mathbf{r}_{k+1}}{\mathbf{r}_k^\top \mathbf{r}_k} \tag{24}$$

$$\mathbf{p}_{k+1} = -\mathbf{r}_{k+1} + \beta_{k+1}\mathbf{p}_k. \tag{25}$$

Compared to other methods, e.g., Gaussian eliminations, QR decomposition, CGM offers a feasible algorithm for extremely large-scale systems.

4.2 The Privacy-Preserving CGM Algorithm

In what follows, we describe our proposed privacy-preserving CGM algorithm (PPCGM) that exploits the CGM method to securely shift the relatively more expensive operations, i.e., matrix-vector multiplications, in each iteration to the CS. Specifically, we first consider the case that matrix $\mathbf{A}$ is nonsingular, symmetric and positive definite as required by the CGM algorithm, and then discuss the case when it is not.

4.2.1 Initialization

We first consider that matrix $\mathbf{A}$ is nonsingular, symmetric and positive definite, and hence have $\mathbf{A}' = \mathbf{A}$ and $\mathbf{b}' = \mathbf{b}$. Thus, in the initialization step, the CC sets the initial solution vector $\mathbf{x_0}$ to a random vector of $\mathbb{R}^{n\times 1}$, and tries to compute $\mathbf{r}_0$ and $\mathbf{p}_0$ according to Equations (18) and (19). Since computing $\mathbf{A}'\mathbf{x}_0$ requires a matrix-vector multiplication, the CC can outsource this computation to the CS.

Particularly, the CC generates a masked coefficient matrix $\hat{\mathbf{A}}' = \mathbf{P}(\mathbf{A}' + \tilde{\mathbf{Z}})\mathbf{T}$, where $\tilde{\mathbf{Z}}$, $\mathbf{P}$, and $\mathbf{T}$ are $n \times n$ random matrices that can be constructed as described in Section 3, and sends it with the permuted solution vector $\hat{\mathbf{x}}_0 = \mathbf{T}^\top \mathbf{x}_0$ to the CS. The CS helps the CC compute the term $\mathbf{A}'\mathbf{x}_0$ in a privacy-preserving manner by computing the following intermediate value

$$\mathbf{h}_0 = \hat{\mathbf{A}}'\hat{\mathbf{x}}_0 = \mathbf{P}(\mathbf{A}' + \tilde{\mathbf{Z}})\mathbf{x}_0.$$

Upon receiving $\mathbf{h}_0$, the CC computes the residual vector as follows

$$\begin{aligned} \mathbf{r}_0 &= \mathbf{A}'\mathbf{x}_0 - \mathbf{b}' \\ &= \mathbf{P}^\top \mathbf{h}_0 - \sum_{(\gamma,\theta)\in\mathcal{Z}'} \tilde{\mathbf{u}}_\gamma(\tilde{\mathbf{v}}_\theta^\top \mathbf{x}_0) - \mathbf{b}' \\ &= \mathbf{P}^\top \mathbf{h}_0 - \sum_{\gamma|(\gamma,\theta)\in\mathcal{Z}'} \tilde{\mathbf{u}}_\gamma \sum_{\theta|(\gamma,\theta)\in\mathcal{Z}'} \tilde{\mathbf{v}}_\theta^\top \mathbf{x}_0 - \mathbf{b}'. \end{aligned} \tag{26}$$

Note that we use different parameters α', β', and $\mathcal{Z}'$ for $\tilde{\mathbf{Z}}$ compared with α, β, and $\mathcal{Z}$ for $\tilde{\mathbf{Z}}_0$. By computing $\tilde{\mathbf{v}}_\theta^\top \hat{\mathbf{x}}_0$ first in equation (26), the CC only performs vector-vector computations, which have linear complexity. This is possible due to the fact that $\tilde{\mathbf{Z}}$ is formed by rank-one matrices and can be decomposed into outer-vector products. If we had formed $\tilde{\mathbf{Z}}$ arbitrarily, the client would not be able to experience any computational or storage complexity gains.

At the end of the initialization step, the client sets the conjugate vector $\mathbf{p}_0 = -\mathbf{r}_0$, and transmits $\hat{\mathbf{p}}_{01}^\top = \mathbf{p}_0^\top \mathbf{P}^\top$, $\hat{\mathbf{p}}_{02} = \mathbf{T}^\top \mathbf{p}_0$, and $\mathbf{r}_0$ to the CS.

4.2.2 Main Iteration

Exploring the CGM iteration, i.e., equations (21), (22), (23), (24), and (25), we notice that equations (21) and (22) need matrix-vector multiplications involving the coefficient matrix $\mathbf{A}'$, while the rest of the equations only require vector-vector multiplications. We exploit these equations to design an efficient collaborative computation between the CC and the CS, where the CS helps compute (21) and (22), and the CC carries out the rest of the equations by itself. To protect the CC's privacy, the CS carries out computations with the transformed matrix $\hat{\mathbf{A}}'$, instead of $\mathbf{A}'$. In what follows, we describe a set of operations that allow the CC to efficiently find $\mathbf{x}_n$, while protecting its data privacy.

To compute α_k, the CC and the CS carry out equation (21) in two steps. First, based on the $\hat{\mathbf{p}}_{k1}^\top$ and $\hat{\mathbf{p}}_{k2}$ received from the CC, the CS computes an intermediate scalar

$$t_k = \hat{\mathbf{p}}_{k1}^\top \hat{\mathbf{A}}' \hat{\mathbf{p}}_{k2} = \mathbf{p}_k^\top (\mathbf{A}' + \tilde{\mathbf{Z}}) \mathbf{p}_k. \tag{27}$$

Second, the CC finds α_k using t_k as follows

$$\begin{aligned}\alpha_k &= \frac{\mathbf{r}_k^\top \mathbf{r}_k}{t_k - \sum_{(\gamma,\theta)\in\mathcal{Z}'} (\mathbf{p}_k^\top \tilde{\mathbf{u}}_\gamma)(\tilde{\mathbf{v}}_\theta^\top \mathbf{p}_k)} \\ &= \frac{\mathbf{r}_k^\top \mathbf{r}_k}{t_k - \sum_{\gamma|(\gamma,\theta)\in\mathcal{Z}'} (\mathbf{p}_k^\top \tilde{\mathbf{u}}_\gamma) \sum_{\theta|(\gamma,\theta)\in\mathcal{Z}'} \tilde{\mathbf{v}}_\theta^\top \mathbf{p}_k}.\end{aligned} \tag{28}$$

Similarly, the CC exploits the CS's resources to find $\mathbf{r}_{k+1}$. The CS first calculates the intermediate vector

$$\mathbf{f}_k = \hat{\mathbf{A}}' \hat{p}_{k2} = \mathbf{P}(\mathbf{A}' + \tilde{\mathbf{Z}}) \mathbf{p}_k, \tag{29}$$

which allows the CC to compute $\mathbf{r}_{k+1}$ as follows

$$\begin{aligned}\mathbf{r}_{k+1} &= \mathbf{r}_k + \alpha_k \left(\mathbf{P}^\top \mathbf{f}_k - \sum_{(\gamma,\theta)\in\mathcal{Z}'} \tilde{\mathbf{u}}_\gamma (\tilde{\mathbf{v}}_\theta^\top \mathbf{p}_k) \right) \\ &= \mathbf{r}_k + \alpha_k \big(\mathbf{P}^\top \mathbf{f}_k - \sum_{\gamma|(\gamma,\theta)\in\mathcal{Z}'} \tilde{\mathbf{u}}_\gamma \sum_{\theta|(\gamma,\theta)\in\mathcal{Z}'} \tilde{\mathbf{v}}_\theta^\top \mathbf{p}_k \big).\end{aligned} \tag{30}$$

Note that when calculating α_k and $\mathbf{r}_{k+1}$ we have also used the fact that $\tilde{\mathbf{Z}}$ is formed by rank-one matrices to provide computational gains to the CC. That is, the CC carries out the computations of α_k and $\mathbf{r}_{k+1}$ in linear time via vector-vector multiplications and vector-vector additions.

Equations (23), (24), and (25) only require vector-vector operations, hence they all can be computed entirely by the CC itself. At the end of the kth iteration, the CC transmits $\hat{\mathbf{p}}_{(k+1)1}^\top = \mathbf{p}_{k+1}^\top \mathbf{P}^\top$ and $\hat{\mathbf{p}}_{(k+1)2} = \mathbf{T}^\top \mathbf{p}_{k+1}$ to the CS for the next iteration $k+1$. Iterations terminate according to the stopping criteria suggested by Golub and Van Loan [25], i.e., $\sqrt{\mathbf{r}_k^\top \mathbf{r}_k} \le \nu ||\mathbf{b}'||_2$, where ν is a tolerance value. After the PPCGM algorithm converges, the CC recovers the solution vector $\mathbf{x}^*$.

We summarize the PPCGM algorithm in Table 1, available in Appendix B in the online supplemental material, available online. Moreover, we note that since the CS has an economic incentive to allocate less computational resources to the CC and return erroneous solutions, the CC should be able to verify the results from the CS. In particular, at the end of the algorithm the CC can multiply $\mathbf{A}'$ by the obtained solution vector $\mathbf{x}$, and compare the product to the constant vector $\mathbf{b}'$. As in [19], the solution vector $\mathbf{x}$ can be deemed correct if $||\mathbf{A}'\mathbf{x} - \mathbf{b}'||_2 \le \epsilon$, where ϵ is a small value. Since the result verification is not the main focus of this paper, we refer the readers to other works for more detailed discussions.

4.2.3 Preconditioning

To accelerate the convergence of our algorithm PPCGM, the CC can reformulate its SLSEs into an equivalent system with better numerical properties. Specifically, since the convergence of the CGM depends on $\mathbf{A}''$s condition number, the CC can instead solve the following SLSE:

$$\mathbf{A}_p \mathbf{x}_p = \mathbf{b}_p, \tag{31}$$

where $\mathbf{A}_p = \mathbf{C}^{-\top} \mathbf{A}' \mathbf{C}^{-1}$, $\mathbf{C} \in \mathbb{R}^{n\times n}$ is a preconditioning matrix, $\mathbf{x}_p = \mathbf{C}\mathbf{x}$ is the variable vector, $\mathbf{b}_p = \mathbf{C}^{-\top}\mathbf{b}'$ is the constant vector, and $^{-\top}$ denotes the inverse and transpose operations. By lowering the condition number of $\mathbf{A}_p$, preconditioning reduces the number of iterations that it takes the PPCGM algorithm to solve an SLSE [25]. In Section 4.2.4, we introduce a transformation to solve general SLSEs, which can also be used by the CC to efficiently outsource the preconditioning in (31) to the CS.

4.2.4 Solving General SLSEs

As shown in Section 4.1, the CGM only works with symmetric and positive definite matrices. Therefore, the CC may be unable to directly solve some SLSEs with our proposed algorithm PPCGM. In particular, we have the following two cases: *1) the SLSE has a coefficient matrix* $\mathbf{A} \in \mathbb{R}^{m\times n}$, *where* $m > n$*: in this case the SLSE is an overdetermined system and the objective is to minimize* $||\mathbf{A}\mathbf{x} - \mathbf{b}||_2$, *i.e., a least-squares problem*; and *2) the SLSE has a nonsingular, non-symmetric, and non-positive definite coefficient matrix* $\mathbf{A} \in \mathbb{R}^{n\times n}$. We can see that the CC can solve both of the above types of SLSEs by formulating the following equivalent SLSE:

$$\mathbf{A}'\mathbf{x} = \mathbf{b}', \tag{32}$$

where $\mathbf{A}' = \mathbf{A}^\top \mathbf{A}$ is symmetric and positive definite, and $\mathbf{b}' = \mathbf{A}^\top \mathbf{b}$.

Since computing $\mathbf{A}'$ requires a matrix-matrix multiplication, which has computational complexity of $\mathcal{O}(\rho)$, where $n^2 \le \rho \le M'n$ (M' denotes the number of non-zero elements), the CC can outsource the computation to the CS. To be more prominent, the CC generates two $m \times n$ random matrices $\tilde{\mathbf{Z}}_0$ and $\tilde{\mathbf{Z}}_1$, and permutation matrices $\mathbf{P_0}, \mathbf{T_0}, \mathbf{Q_0}$ as described in Section 3, and then sends two masked matrices $\hat{\mathbf{A}}_0$ and $\hat{\mathbf{A}}_1$ to the CS:

$$\hat{\mathbf{A}}_0 = \mathbf{Q}_0^\top (\mathbf{A} + \tilde{\mathbf{Z}}_0) \mathbf{P}_0^\top, \tag{33}$$

$$\hat{\mathbf{A}}_1 = \mathbf{Q}_0^\top (\mathbf{A} + \tilde{\mathbf{Z}}_1) \mathbf{T}_0. \tag{34}$$

As proved before, $\hat{\mathbf{A}}_0$ and $\hat{\mathbf{A}}_1$ are computationally indistinguishable from a random matrix and hence do not reveal any information about $\mathbf{A}$. The CS carries out the following secure computation:

$$\mathbf{G} = \hat{\mathbf{A}}_0^\top \hat{\mathbf{A}}_1 \tag{35}$$

$$= \mathbf{P}_0 (\mathbf{A}^\top \mathbf{A} + \mathbf{M}) \mathbf{T}_0. \tag{36}$$

where $\mathbf{M} = \tilde{\mathbf{Z}}_0^\top \mathbf{A} + \mathbf{A}^\top \tilde{\mathbf{Z}}_1 + \tilde{\mathbf{Z}}_0^\top \tilde{\mathbf{Z}}_1$. Thus, upon receiving $\mathbf{G}$ from the CS, the CC can obtain the symmetric positive definite matrix $\mathbf{A}'$ by

$$\mathbf{A}' = \mathbf{P}_0^\top \mathbf{G} \mathbf{T}_0^\top - \mathbf{M}. \tag{37}$$

To avoid matrix-matrix multiplications in the calculation of $\mathbf{M}$, the CC can replace $\tilde{\mathbf{Z}}_0$ and $\tilde{\mathbf{Z}}_1$ with $\sum_{(\gamma,\theta)\in\mathcal{Z}} \tilde{\mathbf{u}}_\gamma^0 \tilde{\mathbf{v}}_\theta^{0\top}$ and $\sum_{(\gamma,\theta)\in\mathcal{Z}} \tilde{\mathbf{u}}_\gamma^1 \tilde{\mathbf{v}}_\theta^{1\top}$, respectively, i.e.,

$$\begin{aligned}\mathbf{M} &= \sum_{(\gamma,\theta)\in\mathcal{Z}} \tilde{\mathbf{v}}_\theta^0\Big(\tilde{\mathbf{u}}_\gamma^{0\top}\mathbf{A}\Big) + \sum_{(\gamma,\theta)\in\mathcal{Z}}\Big(\mathbf{A}^\top\tilde{\mathbf{u}}_\gamma^1\Big)\tilde{\mathbf{v}}_\theta^{1\top} \\ &\quad + \sum_{(\gamma,\theta)\in\mathcal{Z}} \tilde{\mathbf{v}}_\theta^0\Big(\tilde{\mathbf{u}}_\gamma^{0\top}\tilde{\mathbf{u}}_\gamma^1\Big)\tilde{\mathbf{v}}_\theta^{1\top} \\ &= \sum_{\theta|(\gamma,\theta)\in\mathcal{Z}} \tilde{\mathbf{v}}_\theta^0 \sum_{\gamma|(\gamma,\theta)\in\mathcal{Z}} \tilde{\mathbf{u}}_\gamma^{0\top}\mathbf{A} + \left(\sum_{\gamma|(\gamma,\theta)\in\mathcal{Z}} \mathbf{A}^\top\tilde{\mathbf{u}}_\gamma^1\right)\cdot \\ &\quad \sum_{\theta|(\gamma,\theta)\in\mathcal{Z}} \tilde{\mathbf{v}}_\theta^{1\top} + \sum_{(\gamma,\theta)\in\mathcal{Z}} \tilde{\mathbf{v}}_\theta^0\Big(\tilde{\mathbf{u}}_\gamma^{0\top}\tilde{\mathbf{u}}_\gamma^1\Big)\tilde{\mathbf{v}}_\theta^{1\top}.\end{aligned} \tag{38}$$

Note that multiplying a matrix by $\mathbf{P}_0^\top$, $\mathbf{T}_0^\top$, or $\mathbf{Q}_0$ has complexity of at most $\mathcal{O}(n^2)$ and can be done efficiently by the CC. We summarize this SLSE transformation scheme in Table 2, available in Appendix C in the online supplemental material, available online.

Remark 1. The matrix $\mathbf{A}'$ can be calculated just once for many SLSEs that share the same $\mathbf{A}'$ but have different $\mathbf{b}''$s. For example, power system operators solve many state estimation problems for system monitoring and control. These problems have different measurements, i.e., $\mathbf{b}''$s, but the same $\mathbf{A}'$ which depends on network topology and does not change frequently. Thus, finding $\mathbf{A}'$ once is enough to solve a large number of SLSEs.

Remark 2. Since the transformation by employing $\mathbf{A}' = \mathbf{A}^\top\mathbf{A}$ will greatly increase the condition number of the original SLSE (1), solving an SLSE of type 2) based on (32) may take a large number of iterations. To accelerate the convergence of our algorithm PPCGM, the CC can reduce the condition number of (32) by applying the preconditioning technique described in Section 4.2.3. Alternatively, the CC may easily extend our proposed secure outsourcing techniques to the Generalized Minimal Residual Method (GMRES) [26], which is a gradient direction method that can directly solve (1) but has a larger memory requirement than the CGM.

5 Performance and Privacy Analysis

In this section we analyze the computational, memory I/O, and communication complexity of the proposed PPCGM algorithm, and compare it with those of the previous works. We also present a thorough privacy analysis. Note that previous works can only work with square coefficient matrices. To perform fair comparisons, we assume that they employ our proposed SLSE transformation algorithm in Table 1 (available in Appendix B in the online supplemental material, available online) to securely transform an arbitrary coefficient matrix of into a square, symmetric and positive definite matrix.

5.1 Computational Complexity

We define the computational complexity of a party as the number of floating-point (flops) operations (additions, subtractions, multiplications, and divisions), bitwise operations, and encryptions that the party needs to perform. We note that an encryption takes many flops, and a flop takes some bitwise operations. To determine the overall computational complexity for the client and the cloud in PPCGM, we look into Tables 1 and 2 in detail which are available in Appendices B and C in the online supplemental material, available online respectively.

5.1.1 Cloud Client

If the original coefficient matrix is not symmetric and definite positive, the CC runs the algorithm in Table 2 to construct such a matrix in equation (32). To this end, in *line 1* of Table 2, the client generates the random vectors $\bar{\mathbf{u}}^0$ and $\bar{\mathbf{u}}^1$, the random matrices $\tilde{\mathbf{Z}}_0$ and $\tilde{\mathbf{Z}}_1$, and permutation matrices $\mathbf{P}_0$, $\mathbf{T}_0$, and $\mathbf{Q}_0$. To get $\bar{\mathbf{u}}^0$ and $\bar{\mathbf{u}}^1$, the client uses a pseudorandom number generator like the Mersenne Twister [27], which takes a constant number of bitwise operations, say C_R, for each of the $2m$ random numbers. To get $\tilde{\mathbf{Z}}_0$ and $\tilde{\mathbf{Z}}_1$, the client multiplies $\tilde{\mathbf{u}}_\gamma^0\tilde{\mathbf{v}}_\theta^{0\top}$ and $\tilde{\mathbf{u}}_\gamma^1\tilde{\mathbf{v}}_\theta^{1\top}$ (for $(\gamma,\theta)\in\mathcal{Z}$) via $2\alpha\beta|\mathcal{Z}|$ flops, where $|\mathcal{Z}|$ is the number of submatrices in $\mathbf{A}$ with at least one non-zero value. To obtain the permutation matrices (i.e., $\mathbf{P}_0$, $\mathbf{T}_0$, and $\mathbf{Q}_0$), the CC computes two mappings using a traditional search ($2n$, $2n$, $2m$ flops for $\mathbf{P}_0$, $\mathbf{T}_0$, and $\mathbf{Q}_0$, respectively), and one pseudorandom function using the Advanced Encryption Standard (AES), which takes $C_F = \mathcal{O}(w)$ bitwise operations ($w = \lceil\log_2 m!\rceil$). Thus, the total complexity in line 1 is $(2+2C_R)m + 4n + 3C_F$ bitwise operations plus $2\alpha\beta|\mathcal{Z}|$ flops. In *line 2*, in each of Equations (33) and (34), there are 1 sparse matrix addition with $\alpha\beta|\mathcal{Z}|$ flops, and two random permutation matrix and sparse matrix multiplications with $2\alpha\beta|\mathcal{Z}|$ flops. Thus, the total computational complexity in line 2 is $6\alpha\beta|\mathcal{Z}|$ flops. In *line 4*, the client computes Equations (37) and (38). Denote the number of nonzero elements in $\mathbf{G} = \hat{\mathbf{A}}_0^\top\hat{\mathbf{A}}_1$ by $\hat{M}$. Then, Equation (37) requires two matrix permutations and a matrix sum with at most $2\hat{M}$ flops and $\hat{M}$ flops, respectively. In Equation (38), the client performs $2(n/\beta - 1) + (|\mathcal{Z}| - 1)$ rank one matrix additions with zero flops (since we are adding matrices with no overlapping non-zero elements, the client can efficiently build the resulting matrix by appending the matrices' CSC vectors); for each θ at most $2(m/\alpha - 1)$ row/column additions with at most $2mn/\alpha$ flops, i.e., at most $2mn^2/(\alpha\beta)$ flops; at most $2m/\alpha$ sparse matrix and sparse vector products which totally takes $4M - 2n$ flops; at most $2n/\beta$ outer vector products with no more than $2\hat{M}$ flops (the complexity is no more than twice the number of nonzero elements in $\mathbf{M}$, hence no more than twice the number of nonzero elements in $\mathbf{G}$); $|\mathcal{Z}|$ outer vector products with $|\mathcal{Z}| \times \beta^2$ flops; $|\mathcal{Z}|$ sparse inner vector products that are essentially at most m/α different sparse inner vector products with $(m/\alpha)\times(2\alpha - 1)$, i.e., $2m$ flops, and $|\mathcal{Z}|$ scalar and sparse vector products with $|\mathcal{Z}|\times\beta$ flops. Thus, the total computational complexity in line 4 is $5\hat{M} + 4M + 2mn^2/(\alpha\beta) + (\beta^2+\beta)|\mathcal{Z}| + 2m - 2n$. In addition, in *line 5*, since there are M (note that $M < M'$) nonzero values in $\mathbf{A}$, there are at most $2M - n$ flops. Consequently, the overall computational complexity for the cloud client in Table 2 is $5\hat{M} + 6M + 2mn^2/(\alpha\beta) + (8\alpha\beta + \beta^2 + \beta)|\mathcal{Z}| + 2m - 3n$ flops plus $(2+2C_R)m + 4n + 3C_F$ bitwise operations. Note that and $\beta^2|\mathcal{Z}| \le \hat{M}$. We can see that when $\alpha = \Theta(m)$ and $\beta = \Theta(n)$, the computational complexity for the cloud client in Table 2 is $\mathcal{O}(\hat{M}) + \mathcal{O}(M)$ flops.

In *line 1* of Table 1, the client generates the random vector $\bar{\mathbf{u}}$, the matrices $\tilde{\mathbf{Z}}$, $\mathbf{P}$, and $\mathbf{T}$, which require $C_Rn + 4n + 2C_F$ bitwise operations plus $\alpha'\beta'|\mathcal{Z}'|$ flops. Denote the number of nonzero elements in $\hat{\mathbf{A}}'$ by $\hat{M}'$. Then, we have $\hat{M}' = \alpha'\beta'|\mathcal{Z}'|$. In *line 2*, the client constructs the transformed coefficient matrix $\hat{\mathbf{A}}'$ through a matrix addition, two permutation matrix and matrix multiplications, and a permutation matrix and vector multiplication, which takes $3\hat{M}' + n$ flops. In *line 4*, the client computes $\mathbf{r}_0$ through at most n/β' different inner vector products with $(n/\beta') \times (2\beta' - 1)$, i.e., $2n$ flops; at most $\min\{(n/\alpha')(n/\beta' - 1), |\mathcal{Z}'| - 1\}$ scalar additions, i.e., $\min\{n^2/(\alpha'\beta'), |\mathcal{Z}'|\}$ flops; at most n/α' sparse vector and scalar multiplications with totally $(n/\alpha') \times \alpha'$, i.e., n flops; at most $n/\alpha' - 1$ vector additions with no flops (due to nonverlapping non-zero elements and the CSC method mentioned above); two vector subtractions with $2n$ flops; and one permutation matrix and vector multiplication with n flops; which takes $\min\{(n^2/(\alpha'\beta'), |\mathcal{Z}'|\} + 6n$ flops in total. In each iteration, to find α_k in *line 6*, according to equation (28), the client performs 1 inner vector multiplications with $2n - 1$ flops; at most n/β' inner vector multiplications with $(n/\beta') \times (2\beta' - 1)$, i.e., $2n$ flops; at most $\min\{(n/\alpha')(n/\beta' - 1), |\mathcal{Z}'| - 1\}$ scalar additions, i.e., $\min\{(n^2/(\alpha'\beta'), |\mathcal{Z}'|\}$ flops; at most n/α' inner vector multiplications with $(n/\alpha') \times (2\alpha' - 1)$, i.e., $2n$ flops; at most n/α' scalar and scalar multiplications with n/α' flops; at most $n/\alpha' - 1$ scalar additions with $n/\alpha' - 1$ flops; one scalar subtraction, and one scalar division; which has a total of $\min\{(n^2/(\alpha'\beta'), |\mathcal{Z}'|\} + (6 + 2/\alpha')n$ flops. In *line 7*, $\sum_{\theta|(\gamma,\theta)\in\mathcal{Z}'} \tilde{\mathbf{v}}_\theta^\top \mathbf{p}_k$ has already been computed in Equation (28) and hence does not require any more computations. The client needs to perform at most n/α' vector scalar multiplications with $(n/\alpha') \times \alpha'$, i.e., n flops; at most $n/\alpha' - 1$ vector additions with zero flops (due to nonverlapping non-zero elements and the CSC method mentioned above); one permutation matrix and vector multiplication with n flops; one scalar and vector multiplication with n flops; and two vector addition/subtraction with $2n$ flops; to find $\mathbf{r}_{k+1}$. The total cost is $5n$ flops. Similarly, *in line 8*, we can see that computing $\mathbf{x}_{k+1}$, β_{k+1}, $\mathbf{p}_{k+1}$ requires $2n$, $4n - 1$, and $2n$ flops, respectively, resulting in $8n - 1$ flops in total. In *line 9*, computing $\hat{\mathbf{p}}_{(k+1)1}^\top$ and $\hat{\mathbf{p}}_{(k+1)2}$ requires $2n$ flops. Our experiments show that iterations usually end fairly quickly. Thus, when $\alpha' = \Theta(n)$ and $\beta' = \Theta(n)$, the total computational complexity of the PPCGM algorithm in Table 1 is $\mathcal{O}(\hat{M}')$ flops.

Note that $M < M'$, $M' < \hat{M}$, and $M' < \hat{M}'$. Therefore, the overall computational complexity of the PPCGM algorithm is $\mathcal{O}(\hat{M}) + \mathcal{O}(\hat{M}')$ flops.

The scheme proposed by [16] does not specifically solve LSEs with sparse matrices. To make fair comparisons, we assume [16] employs the SLSE transformation algorithm in Table 2 to transform an arbitrary coefficient matrix into a square matrix first, which takes $\mathcal{O}(\hat{M}) + \mathcal{O}(M)$ flops plus $\mathcal{O}(m) + \mathcal{O}(n)$ bitwise operations as shown above. Then, a client encrypts its coefficient matrix by multiplying it with two permutation matrices, which takes $2M'$ flops. Similarly, the client performs $2M'$ multiplications to decrypt the received inverse matrix. To solve an LSE the client performs an additional matrix and vector multiplication, which incurs $2M'' - n$ flops where M'' is the number of nonzero elements in $(\mathbf{A}')^{-1}$. Note that the inverse of a sparse matrix is generally not sparse any more, and we generally have $M'' = \Theta(n^2)$. Thus, the total computational complexity is $\mathcal{O}(M'') + \mathcal{O}(\hat{M})$ flops.

The secure outsourcing proposed in [19] requires a client to perform a problem transformation that takes one diagonal matrix inversion with n flops, a sparse matrix-vector multiplication with $2M' - n$ flops, the multiplication of diagonal matrix and a matrix with a zero diagonal with at most M' flops, the multiplication of a diagonal matrix and a vector with n flops, and an additive homomorphic encryption and random permutations of the elements of an $n \times m$ matrix with M' homomorphic encryptions and M' flops respectively. These operations take a total of $4M' + n$ flops plus M' homomorphic encryptions. Then, in each iteration the client decrypts a vector and performs a vector addition, which takes n flops and n decryptions. Considering the transformation of an arbitrary coefficient matrix into a square matrix, the total computational complexity for this work is $\mathcal{O}(\hat{M})$ flops $+\mathcal{O}(M')$ encryptions.

5.1.2 Cloud Server

In the proposed PPGCM algorithm, the CC and the CS run the algorithm in Table 2 to construct a symmetric and positive definite matrix, and hence the CS needs to compute $\mathbf{G}$ in Table 2 through one matrix multiplication, which takes at most $\hat{M}n$ flops. Thus, the computation complexity for the CS in Table 2 is $\mathcal{O}(\hat{M}n)$. In Table 1, the CS computes $\mathbf{h}_0$ in *line 3* via a matrix vector multiplication with complexity $2\hat{M}' - n$. To compute t_k in *line 5*, the cloud performs two matrix vector multiplications with complexity $4\hat{M}' - 2n$. In *line 5*, the CS also computes $\mathbf{f}_k$, which takes $2\hat{M}' - n$ flops. The CS totally performs $(2\hat{M}' - n) + (4\hat{M}' - 2n)k$ flops in Table 1. Therefore, the total complexity of the CS is $\mathcal{O}(\hat{M}n)$ in the proposed PPGCM.

In [16], the CS computes a matrix inversion that takes $\mathcal{O}(M''n)$ flops. Considering the complexity of running the algorithm in Table 2, the total complexity of the CS is $\mathcal{O}(M''n)$ flops.

The CS in [19] runs an iterative algorithm that computes a matrix vector multiplication over encrypted data in each iteration, which takes $n^2 - n$ multiplications and n^2 exponentiations. We note that each exponentiation takes $\mathcal{O}(d^2)$ floating point operations, where d is the bit-length of the encrypted values. Thus, considering the complexity of running the algorithm in Table 2, the total computational complexity of the CS is $\mathcal{O}(d^4n^2)$.

5.2 Memory I/O Complexity

As mentioned before, to better capture the memory I/O requirement of large-scale SLSEs, we propose a new definition of memory I/O complexity, which is the number of values that are read/written from/into external memory.

If the original LSE system is not symmetric and positive definite, the CC runs the algorithm in Table 2, which reads the original coefficient matrix in $2M + n$ I/O operations and writes the new coefficient matrix with $2M' + n$ I/O operations (due to the CSC matrix representation method introduced in Section 2.1). The CC also needs to read $\mathbf{b}$ and write $\mathbf{b}'$ once, respectively, which takes $2n$ I/O operations. In line 2 of Table 1, to construct $\hat{\mathbf{A}}'$, the CC reads $\mathbf{A}'$ and writes $\hat{\mathbf{A}}'$ to external memory, which takes $2M' + 2\hat{M}' + 2n$ I/O operations. Computing $\mathbf{r}_0$ requires one read of $\hat{\mathbf{b}}'$ which takes n I/O operations. In the main iteration phase, the CC is able to make all of its operations within the RAM memory. At the final iteration, stores the solution $\mathbf{x}^*$ into the external memory, which takes n I/O operations. Therefore, the total

TABLE 3
Computational and Memory I/O Complexity Comparison

Algorithm	Computational Complexity at the CC	Memory I/O Complexity at the CC	Computational Complexity at the CS	Communication Complexity	Matrix Type
Gennaro et al. [13]	$\mathcal{O}(M')$ FHE crypt ops	$6M' + 2M + 6n$ I/O ops	$\Omega(M''n)$	$\mathcal{O}(M')$	General
Wang et al. [14], [15]	$\mathcal{O}(\rho)$ ($n^2 < \rho \leq Mn$) flops	$2M' + 2M + n^2 + 5n$ I/O ops	$\mathcal{O}(M''n)$	$\mathcal{O}(M'')$	General
Lei et al. [16]	$\mathcal{O}(M'')$ flops	$4M'' + 6M' + 2M + 10n$ I/O ops	$\mathcal{O}(M''n)$	$\mathcal{O}(M'')$	General
Atallah et al. [17]	$\mathcal{O}(M'')$ flops	$8M' + 2M + 4n$ I/O ops	$\mathcal{O}(M''n)$	$\mathcal{O}(M'')$	General
Chen et al. [18]	$\mathcal{O}(M')$ flops	$7M' + 2M + 12n$ I/O ops	$\mathcal{O}(M'n)$	$\mathcal{O}(M')$	General
Wang et al. [19]	$\mathcal{O}(M')$ crypt ops	$10M' + 2M + 10n$ I/O ops	$\mathcal{O}(d^4n^2)$	$\mathcal{O}(M')$	Diagonally Dominant
Our scheme	$\mathcal{O}(\hat{M}')$ flops	$2\hat{M}' + 4M' + 2M + 8n$ I/O ops	$\mathcal{O}(\hat{M}n)$	$\mathcal{O}(\hat{M}')$	General

memory I/O complexity of our scheme is no more than $2\hat{M}' + 4M' + 2M + 8n$.

In [16], the CC hides its coefficient matrix using permutation matrices that need one read of $\mathbf{A}'$, and one write of the resulting matrix, which takes $4M' + 2n$ I/O operations. The client decrypts the received inverse matrix similarly, which in general is no longer a sparse matrix, so it takes another $2M'' + n$ I/O operations. To find the solution vector, the CC performs a read of the inverse matrix and vector $\mathbf{b}$ and a write of the final solution, which takes $2M'' + 3n$ I/O operations. Note that transforming an arbitrary matrix into a square matrix for matrix inversion incurs $2M' + 2M + 4n$ memory I/O operations. The memory I/O complexity in [16] for general matrices is thus $4M'' + 6M' + 2M + 10n$.

In [19], the CC protects its data by transforming the problem through a matrix transformation, which takes $4M' + 3n$ I/O operations, and by encrypting the coefficient matrix, which takes additional $4M' + 2n$ I/O operations. At the final iteration, the CC also stores the result in external memory which takes n I/O operations. Similarly, an arbitrary matrix needs to be transformed into a square matrix first, which takes $2M' + 2M + 4n$ I/O operations. Thus, the total memory I/O complexity for the CC is $10M' + 2M + 10n$ I/O operations.

5.3 Communication Complexity

We define the communication complexity as the number of non-zero values that the CC and the CS need to transmit to each other to solve an SLSE.

If the original LSE is non-symmetric and non-positive definite, then the CC and CS collaborate to run the transformation algorithm in Table 2. In the first transmission, the CC uploads $\hat{\mathbf{A}}_0$ and $\hat{\mathbf{A}}_1$, which both have $\alpha\beta|\mathcal{Z}|$ non-zero elements. In the second transmission, the CS sends matrix $\mathbf{G}$ to the CC, which has $\hat{M}$ non-zero elements. Hence, the overall communication complexity of Table 2 is $2\alpha\beta|\mathcal{Z}| + \hat{M}$. We can see that when $\alpha = \Theta(m)$ and $\beta = \Theta(n)$, the total communication complexity in Table 2 is $\mathcal{O}(\hat{M})$.

In the initialization phase of Table 1, the CC sends $\hat{\mathbf{A}}'$ and $\hat{\mathbf{x}}$, which have $\hat{M}'$ and n non-zero values, respectively. The CS replies by transmitting $\mathbf{h}_0$, which has n non-zeros. In the main iteration phase, the CC transmits $\hat{\mathbf{p}}_{k0}$ and $\hat{\mathbf{p}}_{k1}$, which both have n non-zero values. The CS responds with t_k, a scalar, and $\mathbf{f}_k$, which is a vector with n non-zero values. Our experiments show that the proposed PPCGM algorithm usually converges fairly quickly when $\alpha' = \Theta(n)$ and $\beta' = \Theta(n)$. Therefore, the total communication complexity of the PPCGM algorithm in Table 1 is $\mathcal{O}(\hat{M}')$ non-zero value transmissions. The overall complexity of our proposed PPCGM algorithm for Tables 1 and 2 is $\mathcal{O}(\hat{M}) + \mathcal{O}(\hat{M}')$ non-zero value transmissions.

In [16], the CC transmits its concealed matrix to the CS, which has M' non-zero elements, and the CS replies with a concealed inverse matrix, which has M'' non-zero elements. Considering that transforming an arbitrary matrix into a square one requires the transmission of $\mathcal{O}(\hat{M})$ non-zero elements, the total communication complexity for [16] is $\mathcal{O}(\hat{M}) + \mathcal{O}(M'')$.

In [19], the CC transmits a concealed matrix with M' non-zero values to the CS during the initialization phase. In the main iteration, the CC and CS each transmit a vector of length n. Taking into account the LSE transformation in Table 2, the total communication complexity of [19] is $\mathcal{O}(\hat{M}) + \mathcal{O}(M')$ non-zero element transmissions.

A summary of computational, memory I/O, and communication complexity comparison between our algorithm and previous works is also shown in Table 3. Note that most previous works assume square matrices in their study. Although we have analyzed the complexities of several existing schemes, to facilitate more inclusive comparison, we consider square matrices as well, i.e., $m = n$, and disregard the complexity of running the algorithm in Table 2, i.e., focusing on solve the transformed SLSE. Again, note that $M < M'$, $M' < \hat{M}$, $M' < \hat{M}'$, and $M'' = \Theta(n^2)$. We can easily see that our proposed scheme has the lowest complexities both at the CC and CS.

5.4 Privacy Analysis

Exploring the PPCGM algorithm proposed in Section 4, we observe that the CS only has access to the transformed coefficient matrices $\hat{\mathbf{A}}'/\hat{\mathbf{A}}_0/\hat{\mathbf{A}}_1$ and the conjugate vector $\mathbf{p}_k$. According to Theorem 1 the transformed matrices $\hat{\mathbf{A}}'/\hat{\mathbf{A}}_0/\hat{\mathbf{A}}_1$ are computationally indistinguishable from a random matrix. Thus, the CS cannot derive any information about the non-zero elements of coefficient matrices $\mathbf{A}/\mathbf{A}'$ from the transformed matrices $\hat{\mathbf{A}}'/\hat{\mathbf{A}}_0/\hat{\mathbf{A}}_1$. Besides, Theorem 3 guarantees that the positions of the non-zero elements in the matrices sent to the CS are computationally indistinguishable under a CPA. In addition, in contrast to [16], [17], [18], our proposed matrix addition can conceal the amount of non-zero values in the matrices $\mathbf{A}/\mathbf{A}'$, which is also very important in many applications. For example, in power system state estimation, the system matrix contains the topology of the power grid, which can be used by attackers to launch attacks against the grid [12]. By introducing additional non-zero values and permuting the rows and

TABLE 4
Sparse Matrices

Name	Dimension	Non-zeros M
S1	$n = 1.8 \times 10^3$	39.3×10^3
S2	$n = 2.9 \times 10^3$	174.2×10^3
S3	$n = 14.8 \times 10^3$	715.8×10^3
S4	$n = 25.7 \times 10^3$	3.7×10^6

columns of $\mathbf{A}/\mathbf{A}'$, our matrix transformation conceals the network topology. Consequently, the CS is unable to find out either the original positions of or the total number of the non-zero elements, and hence the structure of matrices $\mathbf{A}/\mathbf{A}'$.

We also observe that the CS is unable to derive information about the solution vector $\mathbf{x}_n$. Specifically, to calculate $\mathbf{x}_n$ the CS needs the knowledge of α_k, which is calculated with $\mathbf{r}_k$. However, the CC keeps α_k and $\mathbf{r}_k$ private. We also note from (25) that even if the CS stores $\mathbf{p}_k$ for all k, it cannot calculate $\mathbf{r}_k$ because β_k is kept private by the CC. Moreover, from (22), $\mathbf{r}_k$ also remains unknown from the CS since it would need the coefficient matrix $\mathbf{A}'$ to find it.

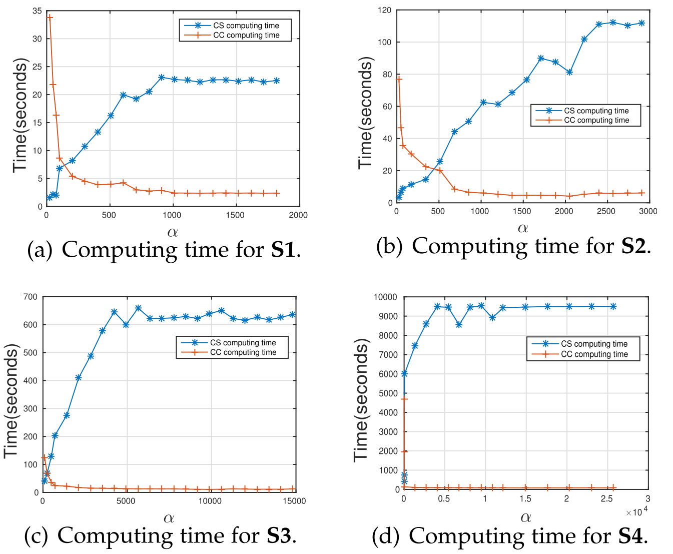
In addition, by keeping α_k and $\mathbf{r}_k$ private, the CC also prevents the CS to learn about the vector $\mathbf{b}'$ and hence $\mathbf{b}$.

We also note that different from [19] where privacy can only be protected if the algorithm converges within n iterations, the privacy in our algorithm can be protected no matter how many iterations are needed.

6 Experiment Results

In this section, we evaluate the computational and memory I/O complexity of the proposed scheme for secure outsourcing of large-scale SLSEs. We implement both the CC and the CS parts of the algorithm in Matlab 2014b. We run the CC on a laptop with a dual-core 2.4 GHz CPU, 4 GB RAM memory, and a 320 GB hard disk at 5,400RPM. The CS is set up on Amazon Elastic Compute Cloud (EC2). As explained in Section 4.2.1, transforming $\mathbf{A}$ into $\mathbf{A}'$ can be done just once for many LSEs, and it needs to be done before many previous LSE outsourcing algorithms can work. Therefore, we focus on the performance of solving $\mathbf{A}'\mathbf{x} = \mathbf{b}'$ with nonsingular, symmetric, and positive definite coefficient matrices of dimension $n \times n$, with n ranging from 1,800 to 25,700. We test our algorithms by solving real-world SLSEs taken from the University of Florida database [28], which includes SLSEs for computational fluid dynamics, aircraft design, transistor design, and structural engineering problems. Table 4 summarizes the testing matrices' parameters.

We first investigate the impact of submatrix size, i.e., α, β, α', and β', on the computing time of the CC and of the CS. In particular, we solve four SLSEs, i.e., with coefficient matrices **S1**, **S2**, **S3**, and **S4**, using different submatrix sizes by setting $\alpha = \beta = \alpha' = \beta'$, and present the results of computing time in Fig. 2. We observe that the CC's computing time becomes lower when α increases. This is because the number of computations performed by the CC in equations (26), (28), and (30) decreases as α increases, which is consistent with our complexity analysis in Section 5.1. In contrast, we find that the CS experiences less computing time when the



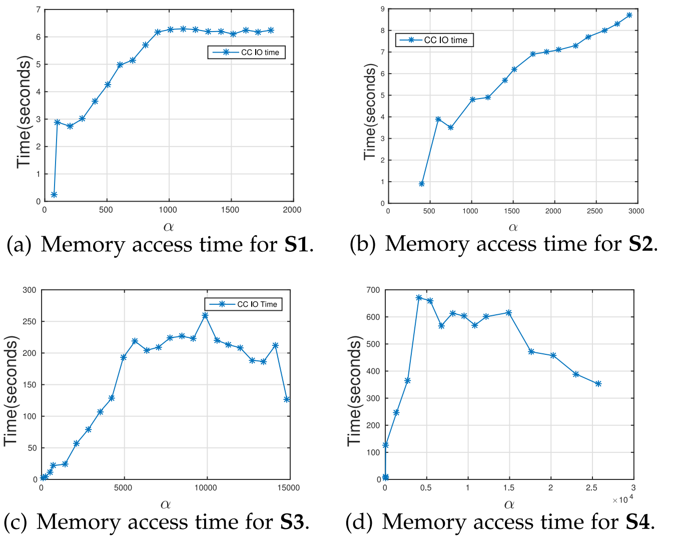
(a) Computing time for **S1**. (b) Computing time for **S2**.
(c) Computing time for **S3**. (d) Computing time for **S4**.

Fig. 2. The computing time of the CC and that of the CS to solve SLSEs with different values of α.

submatrix size is smaller. The reason for this is that the CS performs matrix and vector multiplications with $\hat{\mathbf{A}}'$, which has fewer non-zero elements as α decreases. Therefore, in order to have low overall computing time, we need a small α since the computing time at the CS is generally much higher than that at the CC. However, α cannot be too small because otherwise the CC's computing time would become very high and outweigh the CS's computing time.

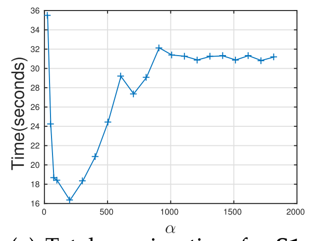
We then evaluate the CC's running time due to memory I/O operations and show the results in Fig. 3. As expected, the CC's I/O time is generally shorter when α is smaller. The reason is that a smaller α introduces fewer non-zero elements to $\hat{\mathbf{A}}'$, which results in fewer I/O operations. For example, In Fig. 3a, we can see that the CC only takes a couple of seconds to perform all the I/O operations when $\alpha = 100$. We observe similar results when when solving the other SLSEs.

To explore the total running time of our proposed algorithm PPCGM, we focus on the total computing and memory I/O access time. Fig. 4 shows the total running

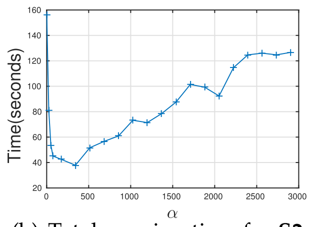


(a) Memory access time for **S1**. (b) Memory access time for **S2**.
(c) Memory access time for **S3**. (d) Memory access time for **S4**.

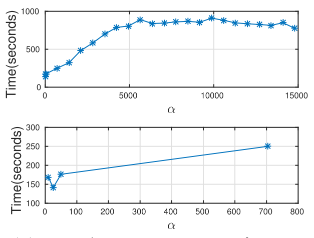
Fig. 3. The I/O memory access time with different values of α.



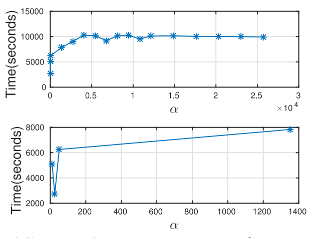
(a) Total running time for **S1**.



(b) Total running time for **S2**.



(c) Total running time for **S3**.



(d) Total running time for **S4**.

Fig. 4. The total running time under different values of α.

TABLE 5
Total Communication Time between the CC and the CS Under a 1 Gbps Connection

Matrix Name	Vector Size	Total Communication Time
S1	112.5×10^3b	789×10^{-6} s
S2	181.2×10^3b	2.05×10^{-3} s
S3	925×10^3b	53.4×10^{-3} s
S4	1.6×10^6b	160.9×10^{-3} s

TABLE 6
Comparison of Total Computing Time

Matrix Name	Our Algorithm	[16]
S3	143.2 s	596.5 s
S4	2533.4 s	8870.7 s

time with different submatrix sizes. Since the CS performs the most expensive computation in the iterations of our algorithm, i.e., matrix and vector multiplications, the total running time is dominated by the CS's computing time when α is large, and hence increases as α increases. On the other hand, when α is very small, the total running time is dominated by the CC's computing time and hence decreases as α increases. We can easily see such results when comparing the plots in Fig. 4 to those in Fig. 2. Consequently, we generally need a small α so as to have low total running time.

We summarize the total communication time of our test problems with a 1 Gbps connection to the cloud in Table 5. We observe that the communication time of our proposed algorithm PPCGM is very small compared to the combined computing and memory I/O access time. For instance, the communication time of S3 is 53.4×10^{-3} s, which is only about 0.4 percent of the total running time and can be neglected. Note that a dedicated 1 Gbps connection is a practical and cost-efficient option for the CC as most cloud service providers offer this service at a low price. For example, Amazon Web Services offers a 1 Gbps connection for \$0.3/hour [29].

TABLE 7
Comparison of Total Memory I/O Access Time

Matrix Name	Our Algorithm	[16]
S3	11.5 s	336.98 s
S4	12.32 s	861.41 s

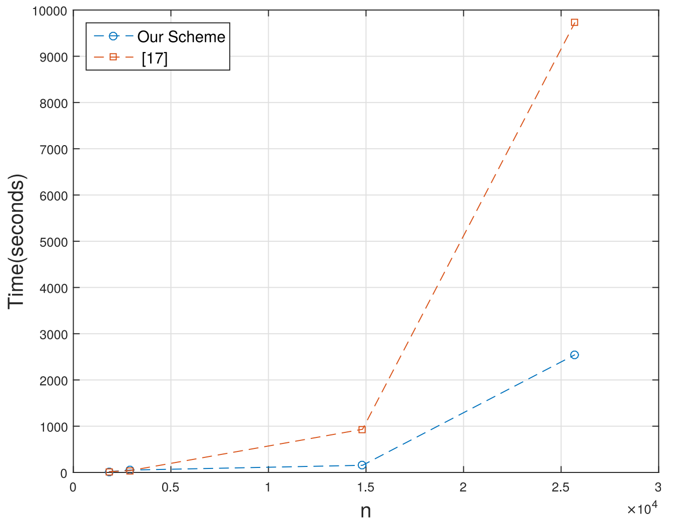


Fig. 5. The total running time of our algorithm compared with that of [16] with different sparse matrices.

To further evaluate our algorithm, we compare our results with those of the matrix inversion algorithm in [16],[3] which has the lowest computational complexity in the previous literature as shown in Table 3.

We first summarize some detailed results on the total computing time and the total memory I/O access time in Tables 6 and Table 7, respectively. We can observe that although our algorithm deals with an increased number of non-zeros due to matrix $\hat{\mathbf{A}}'$ having more non-zeros than $\mathbf{A}$, the computing time and the memory I/O time of our algorithm are much less than those of the algorithm in [16] because of more efficient algorithm design. Besides, we can observe that the total running time saving offered by our algorithm is very attractive. For example, in the case of a sparse matrix of size $n = 25{,}700$, the total running time of our algorithm is 2545.72 seconds, compared to a total of 9732.11 seconds of the algorithm in [16]. Thus, our algorithm can achieve as high as 74 percent time saving, which is very impressive. We also compare in Fig. 5 the total running time of our algorithm with that of [16] with different sparse matrices. We can find that the time saving of our algorithm becomes more and more significant compared to [16] as the sparse matrix becomes larger.

7 Conclusion

In this paper, we have investigated the problem of securely outsourcing large-scale SLSEs. In particular, to protect the cloud client's privacy and preserve the sparsity of the SLSEs, we have developed a privacy and sparsity preserving matrix transformation scheme based on linear algebra

3. Note that [17] also has the same lowest computational complexity as [16] and employs similar techniques.

and shown that it is CPA-secure. Then, we have devised an algorithm based on the conjugate gradient method that can solve large-scale SLSEs efficiently while protecting the client's privacy. Formal analysis has demonstrated that our proposed algorithm has much lower computational and memory I/O complexities than previous works at both the client and the cloud, and can protect the client's privacy well. We have also conducted extensive experiments on Amazon Elastic Compute Cloud (EC2) and found that our algorithm offers significantly less total running time compared to previous works.

ACKNOWLEDGMENTS

This work was partially supported by the U.S. National Science Foundation under grants CNS-1343220, CNS-1149786, ECCS-1128768.

REFERENCES

[1] President's Council of Advisors on Science and Technology, "Big data and privacy: A technological perspective," May 2014. [Online]. Available: http://www.whitehouse.gov/sites/default/files/microsites/ostp/PCAST/pcast_big_data_and_privacy_may_2014.pdf

[2] T. Kraska, "Finding the needle in the big data systems haystack," *IEEE Internet Comput.*, vol. 17, no. 1, pp. 84–86, Jan. 2013.

[3] A. Ipakchi and F. Albuyeh, "Grid of the future," *IEEE Power Energy Mag.*, vol. 7, no. 2, pp. 52–62, Mar. 2009.

[4] H.-C. Chu, D.-J. Deng, and J.-H. Park, "Live data mining concerning social networking forensics based on a Facebook session through aggregation of social data," *IEEE J Sel. Areas Commun.*, vol. 29, no. 7, pp. 1368–1376, Aug. 2011.

[5] C. Jiang, Y. Chen, and K. Liu, "Graphical evolutionary game for information diffusion over social networks," *IEEE J. Sel. Topics Signal Process.*, vol. 8, no. 4, pp. 524–536, Aug. 2014.

[6] Y. Simmhan, et al., "Cloud-based software platform for big data analytics in smart grids," *Comput. Sci. Eng.*, vol. 15, no. 4, pp. 38–47, Jul. 2013.

[7] E. E. Schadt, M. D. Linderman, J. Sorenson, L. Lee, and G. P. Nolan, "Computational solutions to large-scale data management and analysis," *Nature Rev. Genetics*, vol. 11, no. 9, pp. 647–657, Sep. 2010.

[8] H. Demirkan and D. Delen, "Leveraging the capabilities of service-oriented decision support systems: Putting analytics and big data in cloud," *Decision Support Syst.*, vol. 55, no. 1, pp. 412–421, 2013.

[9] E. Kohlwey, A. Sussman, J. Trost, and A. Maurer, "Leveraging the cloud for big data biometrics: Meeting the performance requirements of the next generation biometric systems," presented at the IEEE World Congr. Serv. (SERVICES), Washington, DC, USA, Jul. 2011.

[10] U. Kang, D. Chau, and C. Faloutsos, "Pegasus: Mining billion-scale graphs in the cloud," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, Mar. 2012, pp. 5341–5344.

[11] S. Sakr, A. Liu, D. Batista, and M. Alomari, "A survey of large scale data management approaches in cloud environments," *IEEE Commun. Surveys Tutorials*, vol. 13, no. 3, pp. 311–336, Mar. 2011.

[12] Y. Liu, P. Ning, and M. K. Reiter, "False data injection attacks against state estimation in electric power grids," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2009, pp. 21–32.

[13] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Proc. 30th Annu. Conf. Advances Cryptology*, 2010, pp. 465–482.

[14] C. Wang, K. Ren, and J. Wang, "Secure and practical outsourcing of linear programming in cloud computing," in *Proc. Int. Conf. Comput. Commun.*, 2011, pp. 820–828.

[15] C. Wang, B. Zhang, and K. R. J. A. Roveda, "Privacy-assured outsourcing of image reconstruction service in cloud," *IEEE Trans. Emerging Topics Comput.*, vol. 1, no. 1, pp. 166–177, Jun. 2013.

[16] X. Lei, X. Liao, T. Huang, H. Li, and C. Hu, "Outsourcing the large matrix inversion computation to a public cloud," *IEEE Trans. Cloud Comput.*, vol. 1, no. 1, pp. 78–87, Jan.-Jun. 2013.

[17] M. J. Atallah, K. Pantazopoulos, J. R. Rice, and E. E. Spafford, "Secure outsourcing of scientific computations," in *Trends in Software Engineering*, vol. 54, M. V. Zelkowitz, Ed. Amsterdam, Netherlands: Elsevier, 2002, pp. 215–272.

[18] X. Chen, X. Huang, J. Li, J. Ma, W. Lou, and D. S. Wong, "New algorithms for secure outsourcing of large-scale systems of linear equations," *IEEE Trans. Inform. Forensics Secur.*, vol. 10, no. 1, pp. 69–78, Jan. 2015.

[19] C. Wang, K. Ren, J. Wang, and Q. Wang, "Harnessing the cloud for securely outsourcing large-scale systems of linear equations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 6, pp. 1172–1181, Jun. 2013.

[20] F. Chen, T. Xiang, and Y. Yang, "Privacy-preserving and verifiable protocols for scientific computation outsourcing to the cloud," *J. Parallel Distrib. Comput.*, vol. 74, no. 3, pp. 2141–2151, 2014.

[21] S. Salinas, X. Chen, C. Luo, and P. Li, "Efficient secure outsurcing of large-scale linear systems of equations," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2015, pp. 605–613.

[22] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Boca Raton, FL, USA: Chapman and Hall/ CRC, 2008.

[23] A. Leon-Garcia, *Probability, Statistics, and Random Processes for Electrical Engineers*. Upper Saddle River, NJ, USA: Prentice Hall, 2008.

[24] J. Nocedal and S. J. Wright, *Numerical Optimization*. Berlin, Germany: Springer, 2006.

[25] G. H. Golub and C. F. V. Loan, *Matrix Computations*. Baltimore, MD, USA: The John Hopkins University Press, 2013.

[26] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Scientific Stat. Comput.*, vol. 7, no. 3, pp. 856–869, 1986.

[27] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. and Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998.

[28] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, 2011.

[29] A. W. Services, "Aws direct connect pricing," 2016. [Online]. Available: https://aws.amazon.com/directconnect/pricing/