pGRASS-Solver: A Graph Spectral Sparsification-Based Parallel Iterative Solver for Large-Scale Power Grid Analysis

Zhiqiang Liu and Wenjian Yu, *Senior Member, IEEE*

***Abstract*—With the increase in the complexity of VLSI chips, power grid analysis has become a challenging task, because linear equations of extremely large size need to be solved. Recent graph sparsification-based solvers have shown promising performance for power grid analysis. However, existing graph sparsification algorithms are implemented in serial computing, while factorization and backward/forward substitution of the sparsifier's Laplacian matrix are hard to parallelize. On the other hand, partition-based iterative methods which are inherently parallel lack a direct control of the relative condition number of the preconditioner and consume more memory. In this work, we propose a novel parallel iterative solver called pGRASS-Solver. We first propose a practically efficient parallel graph sparsification algorithm. Then, the domain decomposition method (DDM) is utilized to solve the sparsifier's Laplacian matrix. To further improve the efficiency, a variant of DDM which employs partial Cholesky factorization and Schur complement matrix sparsification is proposed. Thus, we obtain an efficient parallel preconditioner, which not only leads to fast convergence but also enjoys ease of parallelization. Numerous experiments are conducted to illustrate the superior efficiency of the proposed pGRASS-Solver for large-scale power grid analysis, showing an average 6.8× speedup over a recent parallel iterative solver (Wang et al. 2017). Moreover, it solves a real-world power grid matrix with 0.36 billion nodes and 8.7 billion nonzeros within 20 min on a 16-core machine, which is 10.9× faster than the best result of sequential graph sparsification-based solver (Liu et al. 2022).**

***Index Terms*—Domain decomposition method (DDM), graph spectral sparsification, iterative solver, parallel computing, power grid analysis, preconditioned conjugate gradient (PCG) algorithm.**

I. Introduction

POWER grid analysis is an indispensable step in the modern very large-scale integrated (VLSI) circuits design. It is a computationally challenging task due to the extremely large size of power grids. Many methods have been developed for efficient power grid analysis, including direct solvers, iterative solvers [3], [4] and other specialized methods, such as the hierarchical matrix-based method [5] and domain decomposition method (DDM) [6], [7], [8], [9], [10]. Direct methods, such as Cholesky or LU decomposition [11], [12], solve the simulation problem exactly but do not scale well to large problems due to the excessive memory requirement. On the other hand, iterative methods, such as the Krylov subspace methods [13] or algebraic multigrid (AMG) methods [3], [14], [15], usually consume less memory thereby achieving more scalable performance. Among the most popular iterative methods, graph spectral sparsification-based iterative solvers have shown highly scalable performance for large circuit simulation tasks [2], [16], [17], [18].

Graph spectral sparsification aims to find an ultrasparse subgraph (called sparsifier) which can preserve the spectral properties of the original graph. Spectral sparsification approaches have been extensively studied in both theory [19], [20], [21], [22], [23], [24] and practice [2], [17], [18], [25], [26]. An effective resistance-based sampling method was proposed in [22]. However, computing effective resistances with respect to general graphs can be extremely time consuming. Another approach exploiting effective resistances in a spanning tree instead of the original graph was proposed in [20], which usually causes a much greater number of edges recovered for achieving a similar spectral approximation level. The "BSS process" proposed in [21] can construct ϵ-sparsifiers with $O(n\epsilon^{-2})$ edges for every graph, but the cubic time complexity prevents it from being applied to large-scale practical problems. GRASS proposed in [17] and [18] is the first practically efficient spectral graph sparsification algorithm. It leverages spectral perturbation analysis for identifying and recovering spectrally critical off-tree edges and can produce high-quality spectral sparsifiers (low relative condition number of graph Laplacians). Two different approaches were then proposed in [2] and [25] to speed up the graph sparsification phase. SF-GRASS in [25] leverages spectral graph coarsening and graph signal processing techniques, while feGRASS in [2] is based on effective edge weights and a concept of spectral edge similarity. The both approaches can largely reduce the runtime of graph sparsification. It is shown that feGRASS-based solver achieves better performance than GRASS-based solver and also other preconditioned conjugate gradient (PCG) solvers such as AMG-PCG [3]. However, all these graph sparsification algorithms are implemented under the assumption of serial computing. Besides, factorization and

Manuscript received 2 July 2022; revised 24 October 2022; accepted 21 December 2022. Date of publication 10 January 2023; date of current version 22 August 2023. This work is supported in part by the National Key Research and Development Program of China under Grant 2019YFB2205002, and in part by NSFC under Grant 62090025. The preliminary version has been presented at the IEEE/ACM International Conference on Computer-Aided Design (ICCAD) in 2021 [DOI: 10.1109/ICCAD51958.2021.9643489]. This article was recommended by Associate Editor C. Zhuo. *(Corresponding author: Wenjian Yu.)*

The authors are with the Department of Computer Science and Technology, BNRist, Tsinghua University, Beijing 100084, China (e-mail: liu-zq20@mails.tsinghua.edu.cn; yu-wj@tsinghua.edu.cn).

Digital Object Identifier 10.1109/TCAD.2023.3235754

1937-4151 © 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.

backward/forward substitution of the sparsifier's Laplacian matrix, which are required by graph sparsification-based PCG solvers, are also hard to parallelize.

Partition-based methods are another type of methods which employ divide-and-conquer techniques for parallel computing. DDM is based on the Schur complement matrix [6], [7], but forming and solving the dense Schur complement matrix can be even more costly than solving the original equations. The additive Schwarz method (ASM) introduced in [8] and [9] utilizes overlapping domain decomposition to build a block-structure preconditioner but lacks a direct control of the relative condition number of the preconditioner. Recently, a new block Jacobi preconditioner was proposed in [1], which aims to combine graph sparsification techniques and partition-based methods to deliver both fast convergence and good parallelism. It first partitions the maximum spanning tree (MST) and then forms the block Jacobi preconditioner for parallel computing. However, the MST-guided method requires more memory and more iteration steps than graph sparsification-based methods.

In our preliminary work [27], we propose a parallel graph sparsification-based iterative solver named pGRASS-Solver. The pGRASS-Solver combines the convergence property of graph sparsification-based methods and the divide-and-conquer nature of DDM. It works well for tree-like sparsifiers but the parallel efficiency decreases as the density of sparsifier increases. In this work, to further improve the efficiency of pGRASS-Solver, we propose a variant of DDM which employs partial Cholesky factorization (PCF) and Schur complement matrix sparsification. To avoid confusion, we call the solver developed in [27] pGRASS-Solver1, while the extended and improved one is called pGRASS-Solver2. Our main contributions are summarized as follows.

1) A practically efficient parallel graph spectral sparsification algorithm called pGRASS is proposed, which employs a divide-and-conquer strategy to calculate effective resistances and a parallel edge-recovering technique.
2) DDM is leveraged to solve the sparsifier's Laplacian matrix, thus, a graph sparsification-based parallel preconditioner is obtained. It combines the convergence property of graph sparsification techniques and the inherently parallel nature of DDM. It is also an explicit preconditioner which can be easily reused for the linear systems with multiple right-hand sides.
3) To further improve the efficiency of the proposed solver, a variant of DDM utilizing PCF and spectral sparsification for Schur complement matrices is proposed. With this method, the cost for constructing and applying the preconditioner can be reduced largely while the quality of the preconditioner is almost unaffected.
4) Combining these techniques, we have developed an efficient parallel iterative solver called pGRASS-Solver. Extensive experiments, including power grid DC analysis and transient analysis, have been conducted to verify the efficiency of the proposed solver. Experimental results show that pGRASS-Solver achieves an average 7.9× speedup over sequential feGRASS-based solver [2] and an average 6.8× speedup over the parallel MST-guided method [1] for simulating 16 large-scale power grid DC benchmarks [28], [29] on a 16-core machine. The results on transient analysis show that an average 8.6× speedup is gained over the feGRASS-based solver [2] and several tens times speedup can be gained over the direct solver CHOLMOD [12]. Besides, pGRASS-Solver succeeds in solving a real-world power grid matrix with 0.36 billion nodes and 8.7 billion nonzeros within 20 min, which is 10.9× faster than the best result of sequential graph sparsification-based solver. As far as we know, it is the first time that a power grid matrix containing more than eight billion nonzeros can be solved within half an hour on a 16-core machine.

The remainder of this article is organized as follows. In Section II, we briefly introduce the background of power grid analysis, graph spectral sparsification, and DDM. In Section III and Section IV, an efficient parallel iterative solver is presented in detail. Extensive experimental results for large-scale power grid analysis are demonstrated in Section V. Finally, we draw the conclusions in Section VI.

II. Background

A. Problem of Power Grid Analysis

Power grid analysis aims at analyzing the supply noise of power delivery networks (PDNs) in integrated circuits. In DC analysis the power grid is modeled as a resistive network. It can be formulated as the following problem:

$$Gx = b \tag{1}$$

where G is the conductance matrix, x and b denote the unknown vector of node voltages and the vector of current sources, respectively. In transient analysis the power grid is modeled as an RLC network. It can be formulated as differential algebra equations (DAEs) via modified nodal analysis

$$\begin{bmatrix} G & A_l^T \\ -A_l & O \end{bmatrix}\begin{bmatrix} x \\ i_l \end{bmatrix} + \begin{bmatrix} C & O \\ O & L_0 \end{bmatrix}\begin{bmatrix} \frac{dx}{dt} \\ \frac{di_l}{dt} \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}. \tag{2}$$

Here, G and C denote the conductance matrix and the capacitance matrix, respectively. L_0 is a diagonal matrix whose diagonal elements are inductance values of inductors and A_l is the incidence matrix corresponding to inductors. x, i_l and b denote the vector of node voltages, inductor branch currents, and current sources.

With time integration schemes like the backward Euler scheme, the DAEs are converted to a set of linear equation systems for solving the node voltages at consecutive time points. In this work, we use the backward Euler scheme and solve the following linear equation at each time point:

$$\left(G + \frac{C}{h} + h\tilde{L}\right)x(t+h) = \frac{C}{h}x(t) + b(t+h) - A_l^T i_l(t) \tag{3}$$

where h is the time step and $\tilde{L}$ denotes

$$\tilde{L} = A_l^T L_0^{-1} A_l. \tag{4}$$

The direct solver for sparse matrix can be very efficient for power grid transient simulation with a fixed time step

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.

because the expensive Cholesky factorization can be executed only once and the results can be reused [30]. However, the maximum step size is limited by the smallest distance among the breakpoints of current source waveforms. If varied time steps are adopted, the direct solver can be extremely time-consuming because the expensive matrix factorization needs to be performed whenever the time step changes.

The iterative solver, such as PCG solver, is more suitable for varied time steps because it allows larger time steps to reduce the total time for solving (3) in a transient simulation. It is desired that the preconditioner in the iterative solver can be constructed only once and only the solution phase needs to be performed in each subsequent time step, so as to make the iterative solver more competitive to the direct solver. In this work, an efficient parallel iterative solver is proposed to accelerate both DC and transient analysis of power grids.

B. Graph Spectral Sparsification-Based Iterative Solver

Given an undirected weighted graph $\mathcal{G} = (V, E, w)$, its Laplacian matrix $L_{\mathcal{G}}$ can be defined as follows. Here, V and E are the sets of vertices (nodes) and edges and w is a positive weight function

$$L_{\mathcal{G}}(i,j) = \begin{cases} -w_{i,j}, & (i,j) \in E \\ \sum_{(i,k)\in E} w_{i,k}, & i = j \\ 0, & \text{otherwise.} \end{cases} \quad (5)$$

The Laplacian matrix is singular because the smallest eigenvalue is 0. In many applications, it is desired to solve the Laplacian matrix excluding one row and one column. For simplicity, we still use $L_{\mathcal{G}}$ to denote the resulted symmetric positive definite (SPD) matrix. Now consider solving the following linear equation:

$$L_{\mathcal{G}} x = b. \quad (6)$$

$L_{\mathcal{G}}$ is SPD, so we can employ the PCG algorithm to solve the linear equation [13]. Constructing an effective preconditioner is essential to obtain an efficient linear equation iterative solver.

Graph spectral sparsification aims to find an ultrasparse subgraph $\mathcal{P}$ (called sparsifier) which can preserve spectral properties of the original graph $\mathcal{G}$. A subgraph $\mathcal{P}$ is called k-ultrasparse if it has $n-1+k$ edges. It can be constructed by recovering k off-tree edges into a spanning tree. The similarity between $\mathcal{G}$ and $\mathcal{P}$ can be measured by the relative condition number $\kappa(L_{\mathcal{G}}, L_{\mathcal{P}})$. Smaller relative condition number implies a higher similarity level. The Laplacian matrix of the sparsifier can be leveraged as an effective preconditioner to accelerate PCG iteration [2], [17], [18], [25]. The PCG algorithm will converge in at most $O(\kappa(L_{\mathcal{G}}, L_{\mathcal{P}})^{1/2} \log(1/\epsilon))$ iterations to find an ϵ-accurate solution [31].

Existing graph spectral sparsification typically involves the following two steps [2], [17], [18], [25]: 1) extract a spectrally critical spanning tree from the original graph $\mathcal{G}$ and 2) recover a few spectrally critical off-tree edges from $\mathcal{G}$ into the spanning tree to form the sparsifier $\mathcal{P}$. In this work, a parallel graph sparsification algorithm called pGRASS will be presented in Section III-B. It can be seen as a parallel ersion of feGRASS [2], which is briefly introduced as follows.

To reduce the average stretch of extracted spanning tree, the concept of effective edge weight is utilized in feGRASS. Effective edge weight contains not only weight information but also topological information. It is defined as follows [2]:

$$W_{\text{eff}}(e) = w_{i,j} \times \frac{\log(\max\{\deg(i), \deg(j)\})}{\text{dist}(r,i) + \text{dist}(r,j)} \quad (7)$$

where $\deg(i)$ denotes the degree of vertex i, r is a root node of the tree, and $\text{dist}(r, i)$ denotes the unweighted distance between r and i which can be computed with breadth-first search (BFS). The resulted maximum-effective-weight spanning tree (MEWST) usually has a lower stretch than maximum-weight spanning tree (MWST) [2].

To identify spectrally important off-tree edges, the effective resistance is leveraged as a spectral criticality metric in feGRASS. More specifically, it calculates $w_{i,j}R_{\mathcal{P}}(i,j)$ for each off-tree edge and then sorts them by $w_{i,j}R_{\mathcal{P}}(i,j)$. Here, $R_{\mathcal{P}}(i,j)$ denotes the effective resistance across i and j in $\mathcal{P}$. Then, the concept of spectral edge similarity is utilized to further improve the sparsifier's approximation level. Spectral edge similarity reflects the drop of effective resistance during the edge-recovery procedure. *Spectral edge similarity* between two off-tree edges $e_1 = (p, q)$ and $e_2 = (s, t)$ is defined as follows:

$$\text{Similarity}(e_1, e_2) = f_{p,q}^T L_{\mathcal{P}}^+ f_{s,t} = f_{s,t}^T L_{\mathcal{P}}^+ f_{p,q} \quad (8)$$

where $f_{p,q} = f_p - f_q$, and f_p and f_q is the pth and the qth column of identity matrix, respectively. After an edge $e = (i, j)$ is recovered, the off-tree edges whose effective resistances drop largely should be excluded from subsequent edge recovery. Those off-tree edges can be obtained by executing β-layer BFS from i and j, respectively. Readers may refer to [2] for more details.

Graph sparsification-based iterative solvers can be employed to solve the problem of power grid analysis. For DC analysis (1), the resistive network can be seen as a weighted undirected graph. The conductance matrix is just the Laplacian matrix of that graph. So the Laplacian matrix of the sparsifier can be utilized as an efficient preconditioner. For transient analysis, the coefficient matrix in (3) can be also modeled as a graph. There are three types of edges in that graph, whose weight values are $1/r$, c/h, and $h\tilde{l}$ corresponding to resistors, capacitors, and inductors, respectively. Note that it is a dynamic graph, if varied time steps are used. Suppose $h \in [h_{\min}, h_{\max}]$. A static graph of the dynamic graph can be constructed by setting those weight values to $1/r$, $c/h_{\max}$, and $h_{\min}\tilde{l}$. Compared to sparsifying the dynamic graph at each time step, it can be more efficient to sparsify the static graph at the begining of transient simulation and to reuse the resulted preconditioner in the following time steps. This strategy is adopted in this work. Suppose the dynamic graph and the static graph are denoted as $\mathcal{G}_h$ and $\mathcal{G}_s$, respectively, and the sparsifier obtained by sparsifying $\mathcal{G}_s$ is denoted as $\mathcal{P}_s$. It can be shown that

$$\left\| L_{\mathcal{P}_s}^{-1} L_{\mathcal{G}_h} \right\| \le \left\| L_{\mathcal{P}_s}^{-1} L_{\mathcal{G}_s} \right\| \left\| L_{\mathcal{G}_s}^{-1} L_{\mathcal{G}_h} \right\| \quad (9)$$

and

$$\left\| \left(L_{\mathcal{P}_s}^{-1} L_{\mathcal{G}_h} \right)^{-1} \right\| \le \left\| \left(L_{\mathcal{P}_s}^{-1} L_{\mathcal{G}_s} \right)^{-1} \right\| \left\| \left(L_{\mathcal{G}_s}^{-1} L_{\mathcal{G}_h} \right)^{-1} \right\|. \quad (10)$$

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.

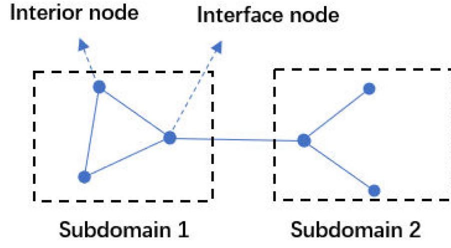

Fig. 1. Graph partitioned into two subdomains.

Because the relative condition number $\kappa(L_{\mathcal{G}_h}, L_{\mathcal{P}_s})$ equals to $\|L_{\mathcal{P}_s}^{-1}L_{\mathcal{G}_h}\|\|(L_{\mathcal{P}_s}^{-1}L_{\mathcal{G}_h})^{-1}\|$, we can derive

$$\kappa\left(L_{\mathcal{G}_h}, L_{\mathcal{P}_s}\right) \le \kappa\left(L_{\mathcal{G}_s}, L_{\mathcal{P}_s}\right)\kappa\left(L_{\mathcal{G}_h}, L_{\mathcal{G}_s}\right). \tag{11}$$

By setting weight values as above, it can be shown that $L_{\mathcal{G}_h} - L_{\mathcal{G}_s}$ and $(h_{\max}/h_{\min})L_{\mathcal{G}_s} - L_{\mathcal{G}_h}$ are both positive semidefinite. Therefore, the eigenvalues of the matrix $L_{\mathcal{G}_s}^{-1}L_{\mathcal{G}_h}$ lie in $[1, (h_{\max}/h_{\min})]$, leading to $\kappa(L_{\mathcal{G}_h}, L_{\mathcal{G}_s}) \le (h_{\max}/h_{\min})$. This means the increase of relative condition number due to this static graph is bounded. And, in practice $\kappa(L_{\mathcal{G}_h}, L_{\mathcal{G}_s})$ can be close to 1, for the power grid cases like those in [28]. Thus, $\kappa(L_{\mathcal{G}_h}, L_{\mathcal{P}_s})$ is usually slightly larger than $\kappa(L_{\mathcal{G}_s}, L_{\mathcal{P}_s})$, and $L_{\mathcal{P}_s}$ is an effective preconditioner for the dynamic coefficient matrix.

C. Domain Decomposition Method

DDM is a specialized method which is developed for solving linear systems in parallel. It employs the techniques of graph partitioning and Schur complement matrix [32]. In this work, we focus on solving linear equations whose coefficient matrices are graph Laplacian matrices (excluding one row and one column), so there is a natural map between the coefficient matrix and graph. Suppose the graph is partitioned into m subdomains. There are two types of nodes in each subdomain, interior nodes, and interface nodes, as shown in Fig. 1. After the unknowns are reordered, (6) becomes

$$\begin{bmatrix} A_1 & E_1 & O & O & \cdots & O & O \\ E_1^T & C_1 & O & F_{12} & \cdots & O & F_{1,m} \\ O & O & A_2 & E_2 & \cdots & O & O \\ O & F_{12}^T & E_2^T & C_2 & \cdots & O & F_{2,m} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ O & O & O & O & \cdots & A_m & E_m \\ O & F_{1,m}^T & O & F_{2,m}^T & \cdots & E_m^T & C_m \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \vdots \\ x_m \\ y_m \end{bmatrix} = \begin{bmatrix} f_1 \\ g_1 \\ f_2 \\ g_2 \\ \vdots \\ f_m \\ g_m \end{bmatrix}. \tag{12}$$

Here, O denotes a zero matrix. x_i and y_i denote the unknowns on interior and interface nodes in the ith subdomain, respectively. f_i and g_i are the right-hand sides corresponding to the interior and interface nodes in the ith subdomain, respectively. Matrices $A_1,\ldots, A_m$ correspond to the interior nodes of m subdomains, and $C_1, \ldots, C_m$ correspond to the interface nodes. Matrices E_i reflect the connections between the interior nodes and the interface nodes in the ith subdomain, and matrices $F_{i,j}$ reflect the connections between the interface nodes in the ith subdomain and the jth subdomain.

After all interior nodes are eliminated, (12) becomes

$$Sy \equiv \begin{bmatrix} S_1 & F_{12} & \cdots & F_{1,m} \\ F_{12}^T & S_2 & \cdots & F_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ F_{1,m}^T & F_{2,m}^T & \cdots & S_m \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} g_1 - E_1^T A_1^{-1} f_1 \\ g_2 - E_2^T A_2^{-1} f_2 \\ \vdots \\ g_m - E_m^T A_m^{-1} f_m \end{bmatrix} \tag{13}$$

where S is called overall Schur complement matrix and S_i is called local Schur complement matrix, which satisfies

$$S_i = C_i - E_i^T A_i^{-1} E_i. \tag{14}$$

To obtain the solution to (12), one can first solve (13) to get interface unknowns y_i and then solve the following equations to get interior unknowns x_i:

$$A_i x_i = f_i - E_i y_i, \ \ i = 1, 2, \ldots, m. \tag{15}$$

Most steps in DDM are inherently parallel and the main bottleneck is to solve (13). The overall Schur complement matrix S is much denser than the original coefficient matrix. When the number of interface nodes is large, it can be even more costly to solve the Schur complement matrix than to solve the original matrix.

In this work, DDM is leveraged to solve the Laplacian matrix of the sparsifier to obtain a parallel preconditioner. The motivation is that the sparsifier is ultrasparse so the number of interface nodes can be small, resulting in a small Schur complement matrix which can be solved efficiently. This strategy combines both advantages of graph sparsification-based approaches and partitioning-based methods, which leads to fast convergence and enjoys ease of parallelization. It works well for the cases where the sparsifier is tree-like. However, the efficiency of the proposed solver deteriorates as the density of the sparsifier increases. To address this problem, a variant of DDM which employs PCF and Schur complement matrix sparsification is proposed. The cost for constructing and applying the preconditioner can be further reduced and the efficiency of the proposed solver can be further improved. The details of theses techniques will be presented in the next two sections.

III. Parallel Iterative Solver Based on Graph Spectral Sparsification and Domain Decomposition Method

In this section, a parallel iterative solver, based on parallel graph spectral sparsification and DDM, is proposed for tackling large-scale power grid analysis problems.

A. Idea

There are mainly three stages in graph sparsification-based PCG solver: 1) run the graph spectral sparsification algorithm to obtain the ultrasparse sparsifier (graph sparsification stage); 2) factorize the sparsifier's Laplacian matrix (preconditioner factorization stage); and 3) run the PCG algorithm with the sparsifier as the preconditioner (PCG iteration stage), where in each iteration the matrix-vector multiplication and the forward/backward substitution of the Cholesky factor are executed. Note that it is essentially solving the linear system

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.

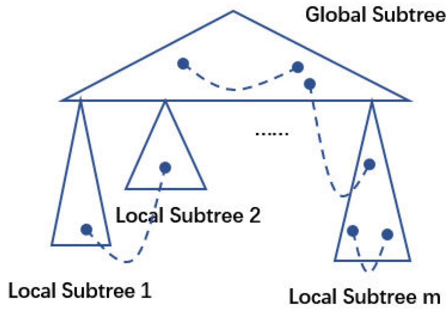

Fig. 2. Spanning tree, which is partitioned into a global subtree and m local subtrees.

whose coefficient matrix is the sparsifier's Laplacian matrix in each iteration, whereas the sparsifier's Cholesky factorization is not required.

When analyzing a typical power grid, the three stages take time of the same order of magnitude, as reported in [2]. For example, to simulate the power grid named "thupg10" in [29], the three stages consume 79.6, 87.1, and 29.3 s, respectively. Notice that the relative tolerance for iteration termination is set 10^{-3} in [2]. If one tightens the termination criterion, the time for the third step would be longer. Therefore, to obtain an efficient parallel solver, all these three steps need to be parallelized efficiently.

Previous graph sparsification-based PCG solvers are difficult to parallelize. For the graph sparsification stage, all the existing practical graph sparsification algorithms are implemented under the assumption of serial computing. For the preconditioner factorization stage and PCG iteration stage, Cholesky factorization and backward/forward substitution of the highly irregular ultrasparse preconditioner matrix are also hard to parallelize. To address the first problem, a practically efficient parallel graph spectral sparsification algorithm called pGRASS is proposed. To tackle the second problem, DDM is employed to solve the ultrasparse preconditioner matrix. The number of interface nodes is small, so the Schur complement matrix can be solved efficiently. The resulted parallel graph sparsification-based PCG solver is named pGRASS-Solver. We will present these two ideas detailedly in the next two sections.

B. Parallel Graph Spectral Sparsification

The serial feGRASS algorithm [2] can be divided into four steps: 1) construct the MEWST; 2) compute effective resistances; 3) sort off-tree edges; and 4) recover off-tree edges. The details of the parallelization of each step are presented as follows.

To construct the MEWST, two basic graph algorithms, including BFS and the Kruskal algorithm are required. Parallel BFS and parallel Kruskal algorithm have been extensively studied and we just use the implementation in problem-based benchmark suite (PBBS) [33]. As for parallel sorting, we use the multiway mergesort implemented in C++ standard library.

Effective resistance calculation with respect to a tree can be parallelized using a simple yet effective divide-and-conquer strategy. Note that computing the effective resistance of one off-tree edge corresponds to *one query* for the distance between two nodes of the edge. First partition the MEWST into a global subtree and m local subtrees, as shown in Fig. 2. Those distance queries can be classified into four types, depending on locations of two nodes, as shown in Fig. 2 with dotted lines. Let $R_{\text{eff}}(i,j)$ denote effective resistance of $e=(i,j)$, $T(i)$ denote the subtree which vertex i belongs to ($T(i)=0$ means vertex i belongs to the global subtree), $R_t(i)$ denote the root vertex of subtree $T(i)$, and $D_l(i,j)$ denote the distance between vertex i and j in the subtree with index l. If both i and j belong to the global subtree, then

$$R_{\text{eff}}(i,j) = D_0(i,j). \tag{16}$$

If both i and j belong to the same local subtree, then

$$R_{\text{eff}}(i,j) = D_{T(i)}(i,j). \tag{17}$$

If i belongs to the global subtree and j belongs to some local subtree, then

$$R_{\text{eff}}(i,j) = D_0(i,R_t(j)) + D_{T(j)}(R_t(j),j). \tag{18}$$

If i and j belong to different local subtrees, then

$$\begin{aligned} R_{\text{eff}}(i,j) = {} & D_0(R_t(i),R_t(j)) + D_{T(i)}(R_t(i),i) \\ & + D_{T(j)}(R_t(j),j). \end{aligned} \tag{19}$$

With (16)–(19), any distance query in the original MEWST can be reduced to queries in subtrees. Then the queries in different subtrees can be handled independently, which is embarrassingly parallel. The queries in one subtree can be computed efficiently leveraging Tarjan's off-line least common ancestor (LCA) algorithm [34].

To parallelize the edge-recovering stage, fisrt note that the main work in this stage is to find spectrally similar edges of each off-tree edge. This operation seems inherently serial because the spectral similarity defined in (8) relies on the latest subgraph $\mathcal{P}$. To overcome this issue, we first observe that for two off-tree edges $e_1=(p,q)$ and $e_2=(s,t)$

$$\text{Similarity}(e_1,e_2) = f_{p,q}^T L_{\mathcal{P}}^{+} f_{s,t} \approx f_{p,q}^T L_{\mathcal{T}}^{+} f_{s,t} \tag{20}$$

where $\mathcal{T}$ denotes the spanning tree and $\mathcal{P}$ denotes the latest subgraph. Equation (20) infers that one can obtain spectrally similar edges of each edge by running β-layer BFS on the spanning tree. The tree is static during the edge-recovering procedure, so a naive approach is to compute spectrally similar edges for each off-tree edge parallelly in advance. Then, one can run the sequential edge-recovering phase in feGRASS [2] except that spectrally similar edges are obtained by reading the data stored before.

However, storing spectrally similar edges of all off-tree edges may consume a large amount of memory, making the aforementioned approach impractical. Another drawback of the naive approach is that it introduces extra work, because there is no need to compute spectrally similar edges for those edges which have been excluded by previous recovered edges. To address these issues, we first divide the off-tree edges into many blocks, as shown in Fig. 3. Each block contains $k \times m$ edges, where m denotes the number of threads and k denotes a constant integer (we set k to 100 in our experiment). Within each block, we compute spectrally similar edges of each edge

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.

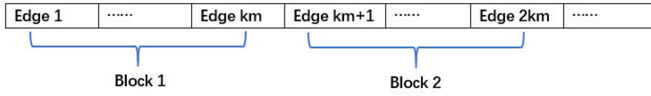

Fig. 3. Off-tree edges, divided into many blocks.

Algorithm 1 pGRASS: parallel GRAph Spectral Sparsification

Input: Graph $\mathcal{G} = (V, E, w)$, the number of edges added to the spanning tree of $\mathcal{G}$ for producing the sparse graph: α, the number of threads m.

Output: Sparse graph $\mathcal{P}$, which is spectrally similar to $\mathcal{G}$.

1: Run parallel BFS to compute unweighted distances. Calculate effective edge weights via (7) in parallel. Run the parallel Kruskal algorithm to obtain MEWST $\mathcal{T}$. Set $\mathcal{P} = \mathcal{T}$.
2: Partition $\mathcal{T}$ into a global subtree and m local subtrees. Use (16)–(19) to convert distance queries in $\mathcal{T}$ to distance queries in those subtrees. Run Tarjan's LCA algorithm for each subtree in parallel.
3: Sort off-tree edges by $w(e)R_e$ in descending order in parallel.
4: Divide off-tree edges into many blocks such that each block except the last one contains km edges.
5: **for** each block **do**
6:   **for** each edge $e = (i, j)$ in the current block in parallel **do**
7:     **if** e is not marked **then**
8:       Run β-layer BFS from i and j respectively. Store the off-tree edges connecting the reached vertices in BFS and other vertices as spectrally similar edges of e.
9:     **end if**
10:   **end for**
11:   **for** each edge e in the current block **do**
12:     **if** α edges have been added into $\mathcal{P}$ **then**
13:       Return.
14:     **end if**
15:     **if** e is not marked **then**
16:       Add e into $\mathcal{P}$.
17:       Mark the spectrally similar edges of e.
18:     **end if**
19:   **end for**
20: **end for**

in parallel, store them, execute the sequential edge-recovering procedure and then move to the next block. The memory for storing the spectrally similar edges can, therefore, be reused. This approach can also reduce the extra work introduced by parallelization because it only processes the edges which have not been excluded by the edges in previous blocks.

Combining the aforementioned techniques, we obtain a parallel graph spectral sparsification algorithm called pGRASS. It is described in Algorithm 1.

C. Solving the Sparsifier's Laplacian Matrix With DDM

In this section, we propose to leverage DDM to solve the sparsifier's Laplacian matrix. Because the sparsifier is ultra-sparse, the number of interface nodes is small when it is partitioned, resulting in a small Schur complement matrix which can be solved efficiently. Thus, we can obtain an efficient parallel preconditioner based on graph sparsification and DDM.

To compute the overall Schur complement matrix in (13), each local Schur complement matrix S_i can be computed using (14) in parallel. However, computing S_i using (14) directly is expensive because it requires solving A_i many times. Below we show how to compute S_i more efficiently.

Algorithm 2 Parallely Solving the Laplacian-Matrix Equation (12) with DDM

Input: Laplacian-matrix equation provided in the form of (12). Cholesky factors of A_i, and S defined by (13) and (14).

Output: Solution x_i and y_i of (12).

1: **for** each subdomain i in parallel **do**
2:   Use the Cholesky factor of A_i to calculate $b_i = g_i - E_i^T A_i^{-1} f_i$, where g_i, E_i and f_i are defined in (12).
3: **end for**
4: Use the Cholesky factor of S to solve (13).
5: **for** each subdomain i in parallel **do**
6:   Use the Cholesky factor of A_i to solve (15) for x_i.
7: **end for**

Suppose the subdomain matrix, including both interior nodes and interface nodes, is factorized in the following way:

$$\begin{bmatrix} P_i A_i P_i^T & P_i E_i \\ E_i^T P_i^T & C_i \end{bmatrix} = \begin{bmatrix} L_{i,11} & O \\ L_{i,21} & L_{i,22} \end{bmatrix} \begin{bmatrix} L_{i,11}^T & L_{i,21}^T \\ O & L_{i,22}^T \end{bmatrix} \quad (21)$$

where P_i denotes permutation matrix which can be computed using any fill-in reducing matrix reordering technique. Note that only the interior nodes are reordered.

From (21), we can get

$$P_i A_i P_i^T = L_{i,11} L_{i,11}^T \quad (22)$$

$$P_i E_i = L_{i,11} L_{i,21}^T \quad (23)$$

and

$$C_i = L_{i,21} L_{i,21}^T + L_{i,22} L_{i,22}^T. \quad (24)$$

Substituting (22)–(24) into (14), we have

$$\begin{aligned} S_i &= C_i - E_i^T A_i^{-1} E_i \\ &= L_{i,21} L_{i,21}^T + L_{i,22} L_{i,22}^T \\ &\quad - \left(P_i^T L_{i,11} L_{i,21}^T\right)^T \left(P_i^T L_{i,11} L_{i,11}^T P_i\right)^{-1} P_i^T L_{i,11} L_{i,21}^T \\ &= L_{i,22} L_{i,22}^T. \end{aligned} \quad (25)$$

Equation (25) infers that S_i can be computed by multiplying the submatrices in the Cholesky factor in (21). Besides, DDM requires factorizing A_i and the Cholesky factor of A_i is already obtained in (22).

After the Schur complement matrix S is formed and factorized, we obtain an efficient parallel preconditioner. It means that at each PCG iteration, the preconditioner equation is solved with DDM as described in Algorithm 2. Because the Cholesky factors of S and A_i are constructed and stored explicitly, the resultant preconditioner can be easily reused for linear systems with multiple right-hand sides.

It is interesting to compare our method with the MST-guided method proposed in [1], since both methods aim to combine the convergence property from graph sparsification techniques and the divide-and-conquer nature from partition-based approaches. The MST-guided method partitions the MST and then recovers all inner-partition edges to form a block Jacobi preconditioner. There are two major differences between the two methods. First, the MST-guided method discards some spectrally critical interpartition edges, while our

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.

method fully inherits the convergence property from graph sparsification techniques and, thus, leads to faster convergence of the iterative solution. Second, the MST-guided method recovers all inner-partition edges leading to a denser preconditioner, while our method retains the sparsity of the sparsifier, which results in shorter factorization time and iteration time along with less memory cost.

D. Overall Algorithmic Flow

The proposed parallel iterative solver which combines graph sparsification and DDM is called *pGRASS-Solver*. In this section, we have employed a naive DDM and we name the resulted solver *pGRASS-Solver1*, whose algorithmic flow is described as follows.

1) Run the pGRASS algorithm (Algorithm 1) parallelly to obtain the sparsifier.
2) With DDM, partition the sparsifier into m subdomains.
3) Form the sparsifier's Laplacian matrix in form of (12), parallelly.
4) Factorize each subdomain matrix in the form of (21), parallelly. Then, compute S_i for each subdomain in parallel using (25), $i = 1, \ldots, m$.
5) Factorize the overall Schur complement matrix S in (13).
6) Run the PCG algorithm with the sparsifier's Laplacian matrix as the preconditioner, where in each iteration step the preconditioner equation is solved with Algorithm 2.

In this algorithm flow, there are three major stages: 1) graph sparsification (step 1); 2) factorization of the preconditioner (steps 2–5); and 3) the PCG iteration (step 6). Most computations are well parallelized. Only steps 2 and 5 (during the factorization of the preconditioner) are executed serially. Because the sparifier is an ultrasparse graph, the both steps consume a small fraction of the overall runtime.

IV. More Efficient Factorization and Solution of the Sparsifier's Laplacian Matrix

To further improve the efficiency of the proposed solver, two techniques are presented in this section. We focus on reducing the cost for factorizing and solving the sparsifier's Laplacian matrix. We first present a variant of DDM, which is based on PCF, to reduce the cost for solving the preconditioner matrix without any additional overhead. Then, an efficient Schur complement matrix sparsification approach is proposed. The sparsified Schur complement matrix (SSCM) can be factorized and solved much more efficiently, with only marginal additional cost for sparsification and negligible increase in the number of PCG iterations. Thus, the efficiency of the proposed solver can be further improved. These two techniques will be presented in detail in the next two sections.

A. Parallel Partial Cholesky Factorization

If the preconditioner equation is solved with DDM, as in Algorithm 2, the submatrices A_i are solved twice in each PCG iteration step: one is for forming the right-hand side of the Schur complement equation $b_i = g_i - E_i^T A_i^{-1} f_i$, the other is for calculating the interior variables $x_i = A_i^{-1}(f_i - E_i y_i)$. To reduce the cost of applying the preconditioner, we present a more efficient way to solve the preconditioner equation.

We can write (21) in a slightly different way

$$\begin{bmatrix} P_i A_i P_i^T & P_i E_i \\ E_i^T P_i^T & C_i \end{bmatrix} = \begin{bmatrix} L_{i,11} & O \\ L_{i,21} & I \end{bmatrix} \begin{bmatrix} I & O \\ O & S_i \end{bmatrix} \begin{bmatrix} L_{i,11}^T & L_{i,21}^T \\ O & I \end{bmatrix} \tag{26}$$

where $S_i = L_{i,22} L_{i,22}^T$. Puting all these equations together leads to

$$\begin{bmatrix} P_1A_1P_1^T & P_1E_1 & O & O & \cdots & O & O \\ E_1^TP_1^T & C_1 & O & F_{1,2} & \cdots & O & F_{1,m} \\ O & O & P_2A_2P_2^T & P_2E_2 & \cdots & O & O \\ O & F_{1,2}^T & E_2^TP_2^T & C_2 & \cdots & O & F_{2,m} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ O & O & O & O & \cdots & P_mA_mP_m^T & P_mE_m \\ O & F_{1,m}^T & O & F_{2,m}^T & \cdots & E_m^TP_m^T & C_m \end{bmatrix} = LS_{\text{aug}}L^T \tag{27}$$

where L and S_{aug} satisfy

$$L = \begin{bmatrix} L_{1,11} & O & O & O & \cdots & O & O \\ L_{1,21} & I & O & O & \cdots & O & O \\ O & O & L_{2,11} & O & \cdots & O & O \\ O & O & L_{2,21} & I & \cdots & O & O \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ O & O & O & O & \cdots & L_{m,11} & O \\ O & O & O & O & \cdots & L_{m,21} & I \end{bmatrix} \tag{28}$$

$$S_{\text{aug}} = \begin{bmatrix} I & O & O & O & \cdots & O & O \\ O & S_1 & O & F_{1,2} & \cdots & O & F_{1,m} \\ O & O & I & O & \cdots & O & O \\ O & F_{1,2}^T & O & S_2 & \cdots & O & F_{2,m} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ O & O & O & O & \cdots & I & O \\ O & F_{1,m}^T & O & F_{2,m}^T & \cdots & O & S_m \end{bmatrix}. \tag{29}$$

Equation (27) is called PCF where only interior variables are eliminated. It implies that the preconditioner can be applied by solving L, S_{aug}, and L^T successively. Note that L is a block diagonal matrix, so L and L^T can be easily solved using forward/backward substitution, which is inherently parallel due to the blocked structure of L. S_{aug} is the overall Schur complement matrix S defined in (13) augmented by identity matrices so solving S_{aug} is equivalent to solving S. This approach is mathematically equivalent to Algorithm 2 but computationally different from that. To analyze time complexity of these two approaches, let $nnz(A)$ denote the number of nonzeros in matrix A and $fac(S)$ denote the Cholesky factor of the overall Schur complement matrix S. If Algorithm 2 is used, the cost for applying the preconditioner is

$$\widetilde{\text{Cost}}_1 = \Sigma_i \big(4nnz\big(L_{i,11}\big) + 2nnz(E_i)\big) + 2nnz(\text{fac}(S)). \tag{30}$$

Under the assumption of parallel computing, the cost becomes

$$\text{Cost}_1 = \max_i \big(4nnz\big(L_{i,11}\big) + 2nnz(E_i)\big) + 2nnz(\text{fac}(S)). \tag{31}$$

If the PCF described in this section is used, the cost is

$$\text{Cost}_2 = \max_i \big(2nnz\big(L_{i,11}\big) + 2nnz\big(L_{i,21}\big)\big) + 2nnz(\text{fac}(S)). \tag{32}$$

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.

It is observed that $nnz(L_{i,21})$ is usually much smaller than $nnz(L_{i,11})$. So solving the preconditioner equation using the techniques described in this section can be more efficient than solving it using Algorithm 2.

B. Sparsification of the Overall Schur Complement Matrix

Although the cost of applying the preconditioner has been reduced from (31) to (32), it still needs to solve the dense Schur complement matrix S. This is expensive in the scenarios where the sparsifier is denser and the size of the Schur complement matrix is larger. It should be noted that there are many applications, such as transient simulation of power grids, where linear equations with the same or similar coefficient matrices need to be solved for many times. For these applications, the time for PCG iteration dominates the overall performance so it is often preferred to recover more off-tree edges to reduce the number of PCG iterations, resulting in large and dense Schur complement matrices.

To improve the efficiency of pGRASS-Solver for these scenarios, we present a spectral sparsification approach which is customized for Schur complement matrices in this section. We have modified the effective resistance-based sampling method [22] to utilize the structural properties of the Schur complement matrix. Note that in the Schur complement matrix S (13), the diagonal blocks S_i are dense while the other blocks are sparse. Thus, instead of sparsifying the overall Schur complement matrix S directly, we propose to sparsify each dense block S_i separately. After those dense blocks are sparsified, they are put together with the other sparse off-diagonal blocks to form the overall SSCM. Let $\tilde{S}_i$ denote the sparsifier for S_i, then the overall SSCM $\tilde{S}$ is

$$\tilde{S} = \begin{bmatrix} \tilde{S}_1 & F_{12} & \cdots & F_{1,m} \\ F_{12}^T & \tilde{S}_2 & \cdots & F_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ F_{1,m}^T & F_{2,m}^T & \cdots & \tilde{S}_m \end{bmatrix}. \tag{33}$$

Each local Schur complement matrix S_i can be written as a non-negative diagonal matrix D_i plus a graph Laplacian matrix $L_{\mathcal{G}_i}$

$$S_i = D_i + L_{\mathcal{G}_i} \tag{34}$$

where the corresponding graph is denoted as $\mathcal{G}_i$. We use the effective resistance-based sampling method [22] to sparsify $\mathcal{G}_i$ to obtain $\mathcal{P}_i$, then the local SSCM can be written as follows:

$$\tilde{S}_i = D_i + L_{\mathcal{P}_i}. \tag{35}$$

The effective resistance-based method [22] samples each edge e with probability proportional to $w_e R_e$, where w_e and R_e denote weight of edge e and effective resistance across the two endpoints of edge e, respectively. For an edge $e = (p, q)$ in $\mathcal{G}_i$, its effective resistance can be computed as follows:

$$R_{p,q} = f_{p,q}^T S_i^{-1} f_{p,q} \tag{36}$$

where $f_{p,q} = f_p - f_q$, and f_p and f_q is the pth and the qth column of identity matrix, respectively. Suppose the dimension of S_i is n_i. We can assume there are $O(n_i^2)$ edges in $\mathcal{G}_i$ because it is dense. Note that S_i is obtained by multiplying $L_{i,22}$ and $L_{i,22}^T$ (25) so there is no need to perform Cholesky factorization on S_i again. Then, the effective resistance becomes

$$R_{p,q} = f_{p,q}^T \left(L_{i,22} L_{i,22}^T\right)^{-1} f_{p,q} = \left(L_{i,22}^{-1} f_{p,q}\right)^T L_{i,22}^{-1} f_{p,q}. \tag{37}$$

$L_{i,22}^{-1}$ can be computed by solving n_i linear equations where the coefficient matrix is $L_{i,22}$ and it takes $O(n_i^3)$ time. After $L_{i,22}^{-1}$ is computed, it takes $O(n_i)$ time to compute the effective resistance of one edge using (37). So computing effective resistances for all edges takes $O(n_i^3)$ time. Note that n_i is the number of interface nodes in the ith subdomain, which is usually very small. So effective resistances can be computed efficiently. We summarize the above Schur complement matrix sparsification algorithm as Algorithm 3.

Algorithm 3 Sparsification of the Schur Complement Matrix

Input: Schur complement matrix S defined by (13) and (14), a parameter α to determine the number of samples.
Output: Sparsified Schur complement matrix $\tilde{S}$ defined by (33).

1: **for** each subdomain i in parallel **do**
2: Construct the diagonal matrix D_i and the graph $\mathcal{G}_i$ using S_i as in (34).
3: Compute $L_{i,22}^{-1}$, where $L_{i,22}$ is defined in (21).
4: Compute effective resistance $R_{p,q}$ for each edge (p, q) in $\mathcal{G}_i$ using (37).
5: Compute sampling probability $Prob_{p,q}$ for each edge: $Prob_{p,q} = \frac{w_{p,q} R_{p,q}}{\sum_{(s,t)\in G_i} w_{s,t} R_{s,t}}$.
6: Set $M_i = \alpha n_i \log n_i$ and $\mathcal{P}_i = \phi$, where n_i is the dimension of S_i.
7: **for** $l = 1$ to M_i **do**
8: Choose a random edge $(j, k) \in \mathcal{G}_i$ with replacement, according to probability $Prob_{j,k}$.
9: Add edge (j, k) to $\mathcal{P}_i$ with weight $\frac{w_{j,k}}{M_i Prob_{j,k}}$.
10: **end for**
11: Set $\tilde{S}_i = D_i + L_{\mathcal{P}_i}$.
12: **end for**
13: Form $\tilde{S}$ as in (33).

Instead of solving the dense Schur complement matrix in each PCG iteration step, our new approach solves the SSCM so the cost of applying the preconditioner can be largely reduced. We remark that it may bring errors and affect the quality of the preconditioner. However, as shown by experimental results in the next section, there is almost no increase in the number of PCG iteration steps, which is because the spectral properties are well preserved using the proposed sparsification approach. Moreover, because the proposed spectral sparsification approach is highly efficient and it takes much less time to factorize the SSCM than to factorize the original dense Schur complement matrix, the cost of constructing the preconditioner can also be reduced.

Compared with sparsifying S with feGRASS [2] or pGRASS, the proposed Algorithm 3 can find sparsifiers $\tilde{S}$ preserving the spectral properties of S better, leading to less impact on the preconditioning quality and fewer increase in PCG iterations. On the other hand, the time complexity of Algorithm 3 is tolerable because the dimension of the Schur complement matrix is much smaller than the original matrix.

The details of approximately solving the Laplacian matrix of the sparsifier is presented in Algorithm 4. With this method,

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.

Algorithm 4 Parallely Solving the Laplacian-Matrix Equation (12) With PCF and the Sparsification

Input: Laplacian-matrix equation provided in the form of (12), partial Cholesky factors defined in (26)–(28) and Cholesky factor of sparsified Schur complement matrix.

Output: Approximate solution x_i and y_i of (12).

1: **for** each subdomain i in parallel **do**

2: Solve the following linear equations using forward substitution, based on (26).

$$\begin{bmatrix} L_{i,11} & O \\ L_{i,21} & I \end{bmatrix} \begin{bmatrix} z_i \\ b_i \end{bmatrix} = \begin{bmatrix} f_i \\ g_i \end{bmatrix}. \quad (39)$$

3: **end for**

4: Use the Cholesky factor of sparsified Schur complement matrix to solve the following linear equations.

$$\begin{bmatrix} \tilde{S}_1 & F_{12} & \cdots & F_{1,m} \\ F_{12}^T & \tilde{S}_2 & \cdots & F_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ F_{1,m}^T & F_{2,m}^T & \cdots & \tilde{S}_m \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}. \quad (40)$$

5: **for** each subdomain i in parallel **do**

6: Solve the following linear equations using backward substitution.

$$\begin{bmatrix} L_{i,11}^T & L_{i,21}^T \\ O & I \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} z_i \\ y_i \end{bmatrix}. \quad (41)$$

7: **end for**

the cost of applying the preconditioner is

$$\text{Cost}_3 = \max_i \left(2nnz\left(L_{i,11}\right) + 2nnz\left(L_{i,21}\right)\right) + 2nnz\left(\text{fac}\left(\tilde{S}\right)\right). \quad (38)$$

Note that $\tilde{S}$ is much sparser than S, so the cost of solving the preconditioner equations can be reduced largely.

C. Algorithm Description of pGRASS-Solver2

The parallel iterative solver proposed in this section also combines parallel graph sparsification and DDM. Instead of employing naive DDM as in pGRASS-Solver1, it leverages PCF and SSCM, which can be seen as a variant of DDM. We name the resulted solver *pGRASS-Solver2*, whose algorithmic flow is described below.

1) Run the pGRASS algorithm (Algorithm 1) parallelly to obtain the sparsifier.
2) With DDM, partition the sparsifier into m subdomains.
3) Form the sparsifier's Laplacian matrix in form of (12), parallelly.
4) Factorize each subdomain matrix in the form of (21), parallelly. Then, compute S_i for each subdomain in parallel using (25), $i = 1, \ldots, m$.
5) Sparsify the overall Schur complement matrix using Algorithm 3. Factorize the overall SSCM $\tilde{S}$ defined in (33).
6) Run the PCG algorithm with the sparsifier's Laplacian matrix as the preconditioner, where in each iteration step the preconditioner equation is solved with Algorithm 4.

Note that pGRASS-Solver2 differs from pGRASS-Solver1 only in steps 5 and 6. These two steps of pGRASS-Solver2 are more efficient than those of pGRASS-Solver1, especially, for the cases where the Schur complement matrix is large. For DC simulation of power grids, all these steps are executed once and the Schur complement matrix is usually small, so the performance of these two solvers are similar. For transient simulation of power grid, step 1 through step 5 are executed once while step 6 is executed for many times, so the runtime of step 6 dominates the overall performance. On the other hand, it is preferred to recover more off-tree edges to accelerate PCG iterations, resulting in the larger Schur complement matrix. Therefore, pGRASS-Solver2 can be more efficient than pGRASS-Solver1 for the transient analysis problem of power grid.

V. Experimental Results

We first validate the proposed techniques, including parallel graph sparsification, parallel preconditioner matrix factorization, and parallel preconditioner matrix solution, respectively. Then, pGRASS-Solver1 and pGRASS-Solver2 are compared with the feGRASS-based PCG solver [2] and the MST-guided method [1] for power grid DC analysis. After that the experimental results on transient analysis are presented. We have implemented pGRASS-Solver1, pGRASS-Solver2, feGRASS-based PCG solver [2], and the MST-guided method [1] in C++ with MKL [35]. Notice that a binary version of feGRASS has been shared on [36]. For the Cholesky factorization, we use the direct sparse solver CHOLMOD [11], [12]. For graph partitioning, we use the widely adopted graph partitioner METIS [37]. All experiments are carried out on a machine with two 8-core Intel Xeon E5-2630 Processors and 512-GB RAM. Thread-level parallelism is realized by OpenMP with 16 threads. In all experiments, the wall-clock runtime is reported. Unless stated explicitly, the relative tolerance for PCG termination is set to 10^{-6}.

A. Validation of the Parallel Graph Sparsification

To validate the ideas proposed in Section III-B, we compare the pGRASS algorithm (Algorithm 1) with the feGRASS algorithm [2]. For both algorithms, $2\%|V|$ off-tree edges are recovered. The quality of sparsifier is reflected by the relative condition number and the number of iteration steps that PCG converges to a relative tolerance of 10^{-6} with the sparsifier as preconditioner. pGRASS is executed in parallel using 16 threads while feGRASS is executed serially. The results are listed in Table I.

From the results we see that, pGRASS algorithm (Algorithm 1) achieves 6.1× speedups over the sequential feGRASS algorithm on average. For large-scale cases, the speedup is up to 8.2×. This validates the effectiveness of the ideas proposed in Section III-B. Note that there is a little difference in relative condition number and the number of iteration steps between two graph sparsification algorithms, which is caused by the approximation in (20).

B. Validation of DDM-Based Techniques for Factorizing Preconditioner Matrix

To validate the proposed techniques for preconditioner matrix factorization, we compare three methods for factorizing

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.

TABLE I
COMPARISON BETWEEN PARALLEL AND SEQUENTIAL GRAPH SPARSIFICATION ALGORITHMS FOR THE TEST CASES IN [28] AND [29]. (TIME IN UNIT OF SECOND)

Case	$\|V\|$	NNZ	feGRASS			pGRASS			Speedup
			T_s	κ	N_i	T_s	κ	N_i	
ibmpg3	0.9E6	3.7E6	0.35	99.4	49	0.10	49.7	38	3.5
ibmpg4	1.0E6	4.1E6	0.39	59.2	26	0.11	64.2	26	3.5
ibmpg5	1.1E6	4.3E6	0.45	136	59	0.12	131	54	3.8
ibmpg6	1.7E6	6.6E6	0.75	226	82	0.18	197	81	4.2
ibmpg7	1.5E6	6.2E6	0.63	134	53	0.14	66.8	43	4.5
ibmpg8	1.5E6	6.2E6	0.64	132	54	0.15	72.3	49	4.3
thupg1	5.0E6	2.1E7	4.13	183	59	0.68	198	59	6.1
thupg2	8.9E6	3.9E7	8.13	277	62	1.25	209	61	6.5
thupg3	1.2E7	5.1E7	13.1	233	62	1.69	203	60	7.8
thupg4	1.5E7	6.6E7	17.0	211	59	2.19	214	59	7.8
thupg5	1.9E7	8.5E7	24.3	225	61	3.06	213	59	7.9
thupg6	2.4E7	1.1E8	25.3	216	61	3.73	208	61	6.8
thupg7	2.8E7	1.2E8	34.0	231	61	4.42	234	61	7.7
thupg8	4.0E7	1.8E8	50.8	223	62	6.85	297	60	7.4
thupg9	5.2E7	2.2E8	65.3	234	61	7.94	232	61	8.2
thupg10	6.0E7	2.6E8	79.6	242	62	9.72	267	62	8.2
Average	-	-	-	-	-	-	-	-	6.1

T_s, κ and N_i are the time for graph sparsification, the relative condition number and the number of PCG iterations, respectively.

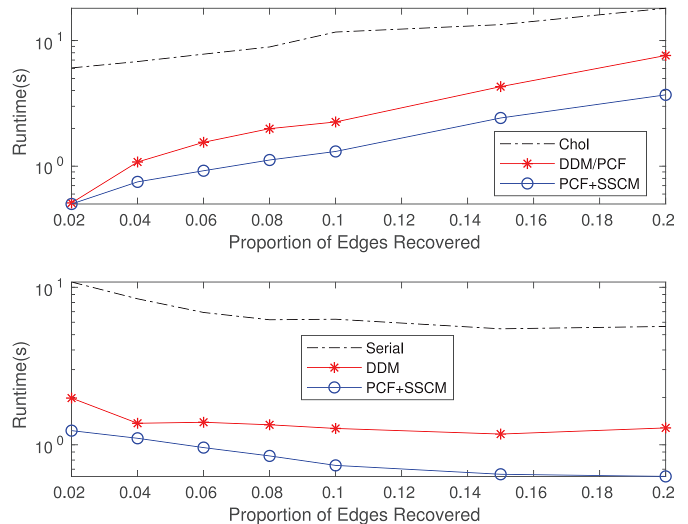

Fig. 4. Time for preconditioner matrix factorization (up) and the time for PCG iteration (down) on case thupg1 with different proportions of off-tree edges recovered.

the sparsifier's Laplacian matrix: 1) Cholesky factorization; 2) naive DDM; and 3) DDM with SSCM. Note that naive DDM and PCF are identical in the matrix factorization phase. Cholesky factorization is executed using a single thread while the other two methods are executed using 16 threads. We compare these methods under two scenarios where $2\%|V|$ and $10\%|V|$ off-tree edges are recovered, respectively. The parameter α in Algorithm 3 is set to 10. The results are listed in Table II.

Naive DDM works well for the scenarios where the preconditioner is sparser, but it may take even more time than the Cholesky factorization for the cases where the preconditioner is denser. The reason is that the size of the Schur complement matrix grows as more off-tree edges are recovered and factorizing the large Schur complement matrix can be extremely time consuming. Things are different after the Schur complement matrix is sparsified. With the proposed Schur complement matrix sparsification approach (Algorithm 3), the number of nonzeros can be reduced largely and the Schur complement matrix can be factorized more efficiently. Moreover, we can also see that it takes only a little time to sparsify the Schur complement matrix. Therefore, the total time for preconditioner matrix factorization T_f is drastically reduced, especially, for the cases where more off-tree edges are recovered. To show the results more clearly, we have plotted the factorization time for case "thupg1" with different percentages of off-tree edges in Fig. 4.

C. Validation of the DDM-Based Techniques for Solving Preconditioner Matrix

To validate the proposed techniques for solving the preconditioner matrix, we compare four PCG solvers where the preconditioner matrix is solved in different ways. In this section, we focus on the iteration phase of these solvers. For serial feGRASS-PCG [2], the preconditioner matrix is solved using serial backward/forward substitution. For pGRASS-Solver1, it is solved using DDM (Algorithm 2). For pGRASS-Solver2, it is solved using PCF along with SSCM (Algorithm 4). We also record the performance of the PCG solver in which the preconditioner matrix is solved solely using PCF. It differs from pGRASS-Solver2 only in that it solves the original Schur complement matrix while pGRASS-Solver2 solves the SSCM. As in the last section, we also compare these solvers under two scenarios where $2\%|V|$ and $10\%|V|$ off-tree edges are recovered, respectively. The results are listed in Table III.

pGRASS-Solver1 equipped with naive DDM achieves more than $6\times$ speedups over the sequential solver for the cases with fewer edges recovered, but the parallel speedup decreases as more edges are added into the sparsifier. Replacing naive DDM with PCF can reduce the preconditioning cost without any additional overhead. If the SSCM instead of the original one is solved, the preconditioning cost can be further reduced, with only negligible increase in PCG iteration steps. These results demonstrate that both the techniques of PCF and SSCM are beneficial for reducing the preconditioning cost T_{pre}, thus, reducing the PCG iteration time T_i. As in the last section, we have also plotted the PCG iteration time for case thupg1 with different percentages of off-tree edges in Fig. 4.

D. Results on DC Analysis of Power Grids

In this section, power grid DC analysis is considered. Cases are from two well-known power grid benchmarks [28], [29]. We compare pGRASS-Solver1 and pGRASS-Solver2 with two recent work: 1) feGRASS-based solver [2] and 2) the MST-guided method [1]. feGRASS-based solver is sequential while the other three are parallel. The relative tolerance of the PCG algorithm is set to 10^{-3}. For all graph sparsification algorithms, $2\%|V|$ off-tree edges are recovered. The performance of the four solvers for the test cases in [28] and [29] is listed in Table IV. For the MST-guided method, pGRASS-Solver1

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.

TABLE II
RESULTS ON FACTORIZING THE PRECONDITIONER MATRIX. (TIME IN UNIT OF SECOND)

Case	2%$\|V\|$ off-tree edges recovered										10%$\|V\|$ off-tree edges recovered									
	Chol	DDM/PCF					PCF+SSCM				Chol	DDM/PCF					PCF+SSCM			
	T_f	T_{sc}	T_{scf}	T_f	n_{sc}	nnz_{sc}	T_{spar}	T_{scf}	T_f	nnz_{sc}	T_f	T_{sc}	T_{scf}	T_f	n_{sc}	nnz_{sc}	T_{spar}	T_{scf}	T_f	nnz_{sc}
ibmpg3	0.62	0.05	0.01	0.06	1.0E3	5.9E4	0.01	0.002	0.06	7.7E3	0.75	0.10	0.02	0.12	2.0E3	2.5E5	0.04	0.004	0.14	2.2E4
ibmpg4	0.69	0.06	0.01	0.07	1.1E3	6.7E4	0.01	0.002	0.07	8.6E3	1.13	0.17	0.14	0.31	3.0E3	5.8E5	0.06	0.01	0.24	3.7E4
ibmpg5	0.72	0.06	0.002	0.06	548	2.0E4	0.006	0.001	0.06	4.5E3	1.04	0.12	0.01	0.13	1.6E3	1.5E5	0.02	0.002	0.14	1.6E4
ibmpg6	1.21	0.10	0.02	0.12	555	2.0E4	0.006	0.001	0.11	4.2E3	1.78	0.18	0.04	0.22	1.8E3	2.1E5	0.03	0.004	0.21	1.8E4
ibmpg7	1.11	0.10	0.01	0.11	1.1E3	7.3E4	0.02	0.002	0.12	9.5E3	2.00	0.27	0.20	0.47	3.5E3	8.1E5	0.08	0.01	0.36	3.9E4
ibmpg8	1.10	0.10	0.01	0.11	1.1E3	7.3E4	0.02	0.002	0.12	9.3E5	2.07	0.22	0.22	0.44	3.8E3	9.5E5	0.09	0.01	0.32	4.7E4
thupg1	6.06	0.39	0.12	0.51	3.4E3	7.4E5	0.07	0.04	0.50	5.5E4	11.7	0.99	1.26	2.25	7.5E3	3.6E6	0.22	0.10	1.31	1.3E5
thupg2	11.4	0.71	0.31	1.02	4.5E3	1.3E6	0.10	0.05	0.86	7.7E4	24.0	2.33	2.35	4.68	1.0E4	6.4E6	0.48	0.17	2.98	1.8E5
thupg3	15.3	1.02	0.49	1.51	5.2E3	1.8E6	0.13	0.06	1.21	9.0E4	33.7	3.44	4.69	8.13	1.2E4	8.9E6	0.84	0.23	4.51	2.1E5
thupg4	19.9	1.37	0.71	2.08	6.0E3	2.4E6	0.11	0.08	1.56	1.0E5	45.6	4.88	10.7	15.6	1.3E4	1.1E7	1.20	0.28	6.36	2.4E5
thupg5	25.6	1.97	0.95	2.92	6.7E3	3.0E6	0.16	0.11	2.24	1.2E5	63.2	6.57	17.7	24.3	1.5E4	1.4E7	1.62	0.29	8.48	2.7E5
thupg6	31.4	2.29	1.00	3.29	7.2E3	3.4E6	0.21	0.13	2.63	1.3E5	79.8	8.86	32.7	41.6	1.6E4	1.7E7	1.84	0.44	11.1	3.0E5
thupg7	38.7	2.96	1.51	4.47	8.1E3	4.2E6	0.27	0.13	3.36	1.4E5	105	12.2	175	187	1.8E4	2.2E7	3.27	0.49	16.0	3.4E5
thupg8	55.6	4.98	3.49	8.47	9.7E3	6.2E6	0.47	0.19	5.64	1.8E5	160	18.0	211	229	2.1E4	2.8E7	5.27	0.58	23.9	4.0E5
thupg9	72.7	6.63	2.40	9.03	1.1E4	8.0E6	0.69	0.19	7.51	2.0E5	214	26.2	558	584	2.4E4	3.6E7	5.61	0.53	32.3	4.6E5
thupg10	87.1	7.99	5.59	13.6	1.2E4	9.0E6	0.99	0.22	9.20	2.2E5	268	35.8	887	923	2.6E4	4.4E7	11.7	0.66	48.2	5.1E5

T_{spar} denotes the time for sparsifying Schur complement matrix with Algorithm 3. T_{sc} denotes the time for factorizing the subdomain matrix and forming Schur complement matrix. T_{scf} is the time for factorizing (sparsified) Schur complement matrix.

T_f denotes the total time of the stage of preconditioner matrix factorization. For DDM or PCF, $T_f = T_{sc} + T_{scf}$. For PCF+SSCM, $T_f = T_{sc} + T_{spar} + T_{scf}$. n_{sc} and nnz_{sc} are the dimension and the number of nonzeros of Schur complement matrix.

The T_{sc} and the n_{sc} of PCF+SSCM are same as those of DDM/PCF, so they are omitted.

TABLE III
RESULTS ON SOLUTION OF THE PRECONDITIONER MATRIX AND PCG ITERATION. (TIME IN UNIT OF SECOND)

Case	2%$\|V\|$ off-tree edges recovered												10%$\|V\|$ off-tree edges recovered											
	Serial			DDM (31)			PCF (32)			PCF+SSCM (38)			Serial			DDM (31)			PCF (32)			PCF+SSCM (38)		
	T_{pre}	N_i	T_i	T_{pre}	N_i	T_i	T_{pre}	N_i	T_i	T_{pre}	N_i	T_i	T_{pre}	N_i	T_i	T_{pre}	N_i	T_i	T_{pre}	N_i	T_i	T_{pre}	N_i	T_i
ibmpg3	0.59	49	1.26	0.09	38	0.17	0.05	38	0.13	0.05	41	0.14	0.63	46	1.32	0.09	23	0.16	0.05	23	0.11	0.05	28	0.12
ibmpg4	0.35	26	0.80	0.07	26	0.14	0.05	26	0.12	0.04	29	0.11	0.37	17	0.67	0.09	16	0.14	0.07	16	0.12	0.05	17	0.09
ibmpg5	0.90	59	1.97	0.15	54	0.27	0.08	54	0.20	0.07	55	0.20	0.77	39	1.54	0.14	35	0.23	0.08	35	0.18	0.07	40	0.18
ibmpg6	2.13	82	4.89	0.39	81	0.65	0.17	81	0.45	0.18	88	0.46	1.69	59	3.53	0.28	46	0.44	0.16	46	0.31	0.16	50	0.32
ibmpg7	1.07	53	2.48	0.19	43	0.31	0.08	43	0.21	0.09	45	0.24	1.11	33	2.04	0.31	26	0.46	0.14	26	0.30	0.11	29	0.28
ibmpg8	1.09	54	2.51	0.20	49	0.35	0.10	49	0.25	0.10	50	0.26	1.12	33	2.06	0.34	28	0.45	0.17	28	0.28	0.12	29	0.22
thupg1	4.66	59	10.8	1.11	59	1.98	0.51	59	1.34	0.43	59	1.23	3.16	29	6.27	0.86	29	1.27	0.54	29	0.97	0.33	29	0.74
thupg2	8.96	62	20.9	1.83	61	3.20	0.92	61	2.31	0.80	61	2.21	5.90	29	11.7	1.50	29	2.43	0.90	29	1.84	0.61	29	1.53
thupg3	11.8	62	27.5	2.51	60	4.50	1.31	60	3.30	1.00	60	2.97	7.86	29	15.4	2.06	29	3.27	1.23	29	2.45	0.81	29	2.02
thupg4	14.5	59	33.8	3.37	59	5.91	1.58	59	4.15	1.30	59	3.85	9.87	28	18.3	2.66	29	4.07	1.69	29	3.04	1.16	29	2.59
thupg5	19.1	61	44.6	3.83	59	6.75	2.12	59	5.08	1.65	59	4.60	13.2	29	25.6	3.64	28	5.49	2.16	28	3.97	1.29	28	3.10
thupg6	23.4	61	54.6	5.31	61	9.48	2.40	61	6.53	2.04	61	6.20	16.1	29	31.3	4.56	29	6.94	3.08	29	5.42	1.73	29	4.02
thupg7	28.4	61	65.5	5.99	61	10.7	3.00	61	7.75	2.38	61	7.09	19.7	29	37.9	6.88	29	9.34	5.62	29	8.12	2.67	29	5.24
thupg8	40.3	62	93.5	8.23	60	14.4	4.54	60	11.6	3.64	61	10.8	27.9	29	53.6	9.28	29	12.9	6.81	29	10.3	3.29	29	6.83
thupg9	51.9	61	119	11.3	61	19.8	5.45	61	13.9	4.67	61	13.1	38.4	30	72.4	16.6	30	22.0	12.2	30	17.5	5.05	30	10.4
thupg10	61.6	62	141	13.3	62	23.2	6.43	62	16.5	5.48	62	15.5	45.0	30	84.8	20.8	29	27.4	14.8	29	21.4	8.10	29	14.5

In the second row, the formula numbers for estimating the preconditioning cost are labeled. T_i denotes the time for PCG iteration and N_i denotes the number of PCG iteration steps. T_{pre} denotes the time for solving the preconditioner matrix, which is the part of T_i that we care most about.

and pGRASS-Solver2, the time for graph partitioning is not included in the factorization time as in [1].

Compared with the feGRASS-based solver, pGRASS-Solver1 achieves an average 6.7× speedup for the total time. With the techniques proposed in Section IV, the parallel speedup is further improved to 7.9×. The comparison between the proposed two solvers with the feGRASS-based solver has been presented in the last three sections so we will skip the details here.

Compared with the parallel MST-guided method [1], pGRASS-Solver1 (pGRASS-Solver2) achieves an average 5.9× (6.8×) speedup for the total time. For the factorization phase, although all methods partition the matrix into 16 subdomain matrices and factorize each subdomain matrix in a single thread, the proposed two solvers show great improvement in factorization time. This is because the subdomain matrices in the proposed solvers are ultrasparse after graph sparsification and can be factorized more efficiently. A great improvement in iteration time is also shown, which is due to the following two reasons. First, the relative condition number of graph sparsification-based preconditioners is lower than that of the MST-guided block Jacobi preconditioners, which results in fewer iteration steps. Second, the preconditioner in the proposed solvers can be solved less costly because of fewer nonzeros in Cholesky factors, which leads to faster single-step iteration. We also note that the proposed solvers are more memory-efficient than the MST-guided method. Take the case thupg10 as an example. The MST-guided method consumes 62-GB memory, while both pGRASS-Solver1 and pGRASS-Solver2 use only 18 GB.

A power grid circuit for the design of a flat panel display (with 2160 × 3840 pixels) from [2] is tested. It includes

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
RESULTS ON POWER GRID DC ANALYSIS. (TIME IN UNIT OF SECOND)

Case	feGRASS-PCG [2]					MST-Guided [1]				pGRASS-Solver1					Speedup		pGRASS-Solver2				Speedup	
	T_s	T_f	T_i	N_i	T_{tot}	T_f	T_i	N_i	T_{tot}	T_s	T_f	T_i	N_i	T_{tot}	Sp_1	Sp_2	T_f	T_i	N_i	T_{tot}	Sp_1	Sp_2
ibmpg3	0.35	0.62	0.43	17	1.40	0.56	0.63	40	1.19	0.10	0.06	0.09	13	0.25	5.6	4.8	0.06	0.07	13	**0.23**	6.1	5.2
ibmpg4	0.39	0.69	0.07	2	1.15	1.44	0.57	28	2.01	0.11	0.07	0.03	3	0.21	5.5	9.6	0.07	0.02	3	**0.20**	5.8	10.0
ibmpg5	0.45	0.72	0.69	20	1.86	0.31	0.58	48	0.89	0.12	0.06	0.13	20	0.31	6.0	2.9	0.06	0.10	21	**0.28**	6.6	3.2
ibmpg6	0.75	1.21	1.65	32	3.61	0.51	1.01	42	1.52	0.18	0.12	0.41	31	0.71	5.1	2.1	0.11	0.20	32	**0.49**	7.4	3.1
ibmpg7	0.63	1.11	0.90	17	2.64	1.44	0.84	28	2.28	0.14	0.11	0.16	17	0.41	6.4	5.6	0.12	0.10	17	**0.36**	7.3	6.3
ibmpg8	0.64	1.10	0.84	17	2.58	1.64	0.88	30	2.52	0.15	0.11	0.17	17	0.43	6.0	5.9	0.12	0.13	18	**0.40**	6.5	6.3
thupg1	4.13	6.06	2.51	13	12.7	4.72	2.20	21	6.92	0.68	0.51	0.55	13	1.74	7.3	4.0	0.50	0.41	14	**1.59**	8.0	4.4
thupg2	8.13	11.4	4.90	14	24.4	9.61	3.89	21	13.5	1.25	1.02	1.31	14	3.58	6.8	3.8	0.86	0.74	14	**2.85**	8.6	4.7
thupg3	13.1	15.3	6.00	13	34.4	15.2	6.47	21	21.7	1.69	1.51	1.15	13	4.35	7.9	5.0	1.21	0.81	13	**3.71**	9.3	5.9
thupg4	17.0	19.9	7.25	12	44.2	20.7	8.99	23	29.7	2.19	2.08	1.48	12	5.75	7.7	5.2	1.56	1.13	12	**4.88**	9.1	6.1
thupg5	24.3	25.6	9.79	12	59.7	32.0	9.93	20	41.9	3.06	2.92	1.68	12	7.66	7.8	5.5	2.24	1.23	12	**6.53**	9.1	6.4
thupg6	25.3	31.4	12.0	13	68.7	47.6	12.3	18	59.9	3.73	3.29	1.97	12	8.99	7.6	6.6	2.63	1.48	12	**7.84**	8.8	7.6
thupg7	34.0	38.7	13.5	12	86.2	76.7	12.7	15	89.4	4.42	4.47	2.95	12	11.8	7.3	7.6	3.36	2.02	12	**9.80**	8.8	9.1
thupg8	50.8	55.6	20.6	12	127	102	18.4	14	120	6.85	8.47	3.78	12	19.1	6.6	6.3	5.64	3.10	12	**15.6**	8.1	7.7
thupg9	65.3	72.7	25.0	12	163	202	32.7	15	235	7.94	9.03	4.90	12	21.9	7.4	10.7	7.51	4.02	12	**19.5**	8.4	12.1
thupg10	79.6	87.1	29.3	12	196	227	28.3	13	255	9.72	13.6	6.21	12	29.5	6.6	8.6	9.20	4.15	12	**23.1**	8.5	11.0
Average	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**6.7**	**5.9**	-	-	-	-	**7.9**	**6.8**

T_s, T_i and N_i denote the time for graph sparsification, the time for PCG iteration and the number of PCG iteration steps, respectively. T_{tot} denotes the total runtime. For feGRASS-PCG, T_f denotes the time for factorizing Laplacian matrix of sparsifier. For the MST-guided block Jacobi method, T_f denotes the time for factorizing subdomain matrices. For pGRASS-Solver1, T_f denotes the time for factorizing subdomain matrices, computing and factorizing Schur complement matrix (step 4 and step 5 in the algorithm flow described in section III-D). For pGRASS-Solver2, T_f corresponds to step 4 and step 5 in the algorithm flow described in section IV-C.

Sp_1 and Sp_2 denote the speedup ratios of pGRASS-Solver1 (or pGRASS-Solver2) over feGRASS-PCG and the MST-Guided method, respectively.

TABLE V
RESULTS ON A LARGER REAL-WORLD POWER GRID. (TIME IN UNIT OF SECOND)

Method	T_s	T_f	T_i	N_i	T_{tot}
feGRASS-PCG	8347	605	4036	66	12988 (3.6 hrs)
pGRASS-Solver1	755	138	451	69	1344 (22.4 mins)
Speedup	11.1	4.4	9.0	-	9.7
pGRASS-Solver2	755	110	322	69	1187 (19.8 mins)
Speedup	11.1	5.5	12.5	-	10.9

3.6×10^8 nodes and the resulting Laplacian matrix has 8.7 billion nonzeros. The MST-guided block Jacobi fails to handle this case due to excessive memory requirement. For the other three methods, $2\%|V|$ off-tree edges are recovered to construct the ultrasparse sparsifier. pGRASS-Solver1 and pGRASS-Solver2 are executed with 16 threads. The results are listed in Table V. We note that the feGRASS-based solver with $2\%|V|$ off-tree edges recovered runs faster than the results reported in [2]. Using 16 threads, pGRASS produces the sparsifier in 755 s, which is $11\times$ faster than the feGRASS algorithm. As for the total time, pGRASS-Solver1 and pGRASS-Solver2 achieve $9.7\times$ and $10.9\times$ speedups over feGRASS-based PCG solver, respectively. As far as we know, it is the first time that a power grid matrix containing more than eight billion nonzeros is solved within half an hour on a 16-core machine.

E. Results on Transient Analysis of Power Grids

In this section, power grid transient simulation is considered. Cases are from two well-known power grid benchmarks [28], [29]. For the cases from [29], capacitors and inductors with random values are added (similar to IBM PG benchmarks) and periodic pulse currents are generated at each current source for transient analysis. We compare direct solver CHOLMOD [12], feGRASS-based solver [2], pGRASS-Solver1 and pGRASS-Solver2 for this application. DC analysis is first performed, followed by repeatedly solving (3) toward time 5 ns with varied time steps for three iterative solvers and a fixed time step (10 ps, 500 time points totally) for the direct solver. The varied time steps are determined with the breakpoints of current sources, but restricted not to exceed 200 ps for error control. The resulted total number of time points is also recorded. Although the coefficient matrix changes with the time step, all the linear equations share the same preconditioner. For all these iterative solvers, $10\%|V|$ off-tree edges are recovered to form sparsifiers and the relative tolerance of PCG algorithms is set to 10^{-6}.

The results are listed in Table VI. Note that the T_{dc} reported, here, is larger than the T_{tot} reported in Table IV, because the relative tolerance of PCG is set to 10^{-6}, here, to produce a more accurate initial condition for transient simulation. From the results we see that the proposed pGRASS-Solver2 achieves $8.6\times$ speedups averagely over the sequential feGRASS-based solver. It is interesting to notice that pGRASS-Solver2 shows more than $2\times$ improvement on average over pGRASS-Solver1 for those transient analysis benchmarks, which is larger than the improvement in DC analysis shown in the last section. There are mainly two reasons. First, the overall performance of transient analysis is dominated by the time for PCG iteration. With the techniques proposed in Section IV, the cost for solving the preconditioner matrix can be reduced largely, with only negligible increase in the number of PCG iterations. Second, it is preferred to recover more off-tree edges to reduce the number of PCG iterations in transient analysis, leading to a larger Schur complement matrix whose factorization can be

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.

TABLE VI
RESULTS ON POWER GRID TRANSIENT ANALYSIS. (TIME IN UNIT OF SECOND)

Case	feGRASS-PCG [2]						pGRASS-Solver1						Direct Solver [12]				pGRASS-Solver2									Speedup		
	T_{dc}	T_s	T_f	T_{trans}	N_i	T_{tot}	T_{dc}	T_s	T_f	T_{trans}	N_i	T_{tot}	T_{dc}	T_f	T_{trans}	T_{tot}	T_{dc}	T_f	T_{trans}	$\#pts$	N_i	T_{tot}	$Err(mV)$	Rel(%)	Sp_1	Sp_2	Sp_3
ibmpg3t	2.23	0.32	0.45	97.8	39.6	101	0.33	0.13	0.11	14.0	39.9	14.6	20.2	18.7	80.2	119	0.30	0.10	9.76	81	41.7	**10.3**	2.2/1.2	0.12	9.8	1.4	11.6
ibmpg4t	1.88	0.38	0.63	20.8	13.2	23.7	0.32	0.17	0.24	4.60	14.0	5.33	36.8	35.0	107	179	0.29	0.17	2.96	37	15.7	**3.59**	0.13/0.05	0.01	6.6	1.5	49.9
ibmpg5t	3.14	0.44	0.65	160	49.0	164	0.45	0.19	0.16	20.2	34.4	21.0	8.85	7.76	57.2	73.8	0.38	0.14	12.9	81	37.8	**13.6**	0.93/0.56	0.05	12.1	1.5	5.4
ibmpg6t	6.85	0.77	0.99	270	53.5	279	0.95	0.25	0.22	28.7	37.0	30.1	8.58	6.41	77.3	92.3	0.75	0.21	19.4	81	39.8	**20.6**	1.6/0.99	0.09	13.5	1.5	4.8
thupg1t	21.0	4.56	7.34	236	20.4	269	3.17	0.74	2.08	51.0	20.3	57.0	221	213	378	812	2.41	1.06	25.0	48	20.3	**29.2**	1.3/0.13	0.07	9.2	2.0	27.8
thupg2t	40.4	10.5	15.6	442	20.2	509	5.47	1.41	5.07	110	20.1	122	357	344	657	1358	4.32	2.61	63.6	48	20.1	**71.9**	1.1/0.16	0.06	7.1	1.7	18.9
thupg3t	55.9	13.9	22.2	580	20.1	672	7.70	1.79	7.45	118	20.5	135	554	537	848	1939	5.87	4.04	78.1	48	20.6	**89.8**	1.3/0.13	0.07	7.5	1.5	21.6
thupg4t	70.7	18.4	30.6	775	20.6	895	10.2	2.30	13.3	166	21.1	192	1004	974	1203	3181	7.60	6.29	106	48	21.1	**122**	1.2/0.11	0.07	7.3	1.6	26.1
thupg5t	94.5	24.7	39.8	986	21.3	1145	12.7	3.07	15.9	194	20.8	226	1404	1356	1546	4306	9.90	7.62	123	48	20.8	**144**	1.4/0.13	0.08	8.0	1.6	29.9
thupg6t	111	28.1	53.3	1327	22.1	1519	16.5	3.52	59.5	262	21.9	342	1916	1871	1956	5743	12.6	10.0	161	48	21.9	**187**	1.2/0.12	0.07	8.1	1.8	30.7
thupg7t	138	36.6	70.9	1564	21.2	1810	19.6	4.66	115	336	20.8	475	4832	4738	3689	13259	14.9	14.6	171	48	20.8	**205**	1.2/0.14	0.07	8.8	2.3	64.7
thupg8t	200	54.6	111	2220	21.3	2586	29.7	7.60	583	549	21.1	1169	5021	4918	4597	14536	23.2	22.0	277	48	21.1	**330**	1.2/0.25	0.07	7.9	3.5	44.0
thupg9t	257	70.1	151	2752	21.5	3230	36.8	8.22	488	770	21.2	1303	10452	8117	6219	24788	28.6	29.2	433	48	21.2	**499**	2.4/0.58	0.13	6.5	2.6	49.7
thupg10t	308	86.9	213	3415	21.5	4023	46.5	10.6	1786	1320	21.3	3163	14941	13318	8647	36906	34.4	44.2	458	48	21.4	**547**	1.9/0.32	0.11	7.4	5.8	67.5
Average	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**8.6**	**2.2**	**32.4**

T_{dc}, T_s and T_f are the time for DC analysis, graph sparsification and preconditioner matrix factorization. T_{trans} denotes the time for all the subsequent transient steps. T_{tot} denotes the total runtime, i.e. $T_{tot} = T_{dc} + T_s + T_f + T_{trans}$. The T_s of pGRASS-Solver2 is the same as that of pGRASS-Solver1 so it is omitted. For the direct solver, T_f denotes the time for factorizing the coefficient matrix and $T_{tot} = T_{dc} + T_f + T_{trans}$.

In column $Err(mV)$, the maximum and average node voltage errors of pGRASS-Solver2 to the solution obtained by CHOLMOD are shown. Rel(%) is the maximum relative error to VDD voltage. N_i is the average number of PCG iterations per transient step and $\#pts$ is the number of time points. Sp_1, Sp_2 and Sp_3 denote the speedup ratios of pGRASS-Solver2 over feGRASS-PCG, pGRASS-Solver1 and CHOLMOD, respectively.

extremely costly. With the proposed Schur complement matrix sparsification approach, the cost for factorizing the preconditioner matrix can be reduced greatly. It is, especially, true for large-scale cases. For example, pGRASS-Solver2 achieves up to 5.8× speedups over pGRASS-Solver1 on the largest case named "thupg10t."

To verify the accuracy of the proposed iterative solver for transient simulation, the results derived from the direct solver CHOLMOD [12] are also listed in Table VI. The differences between the node voltages computed by pGRASS-Solver2 and CHOLMOD solver are denoted by *Err*. Both the maximum error and the average error are recorded. The results show that the node voltage error is less than 2.4 mV for all cases. As the supply voltage is 1.8 V, the relative error is below 0.13%, which is tolerable. The transient waveforms of node n0_20706300_8937900 and node n1_29561400_9521100 in case "ibmpg3t" are plotted in Fig. 5. They validate the accuracy of transient simulation using the proposed parallel iterative solver. Besides, we can also see that the proposed pGRASS-Solver2 achieves 32.4× speedups averagely over the sequential direct solver CHOLMOD.

To show the parallel scalability, we execute the two versions of pGRASS-Solver on the case named thupg1t with the different number of threads. To better demonstrate where the speedups come from, i.e., from algorithm improvement or parallel implementations, here, we just change the number of threads, while the number of partitions is kept constant (set to 16 for all thread numbers). So the total work under the different number of threads remains constant. The results are listed in Table VII. We see that single-thread pGRASS-Solver1 runs slower than feGRASS-PCG, which is because that pGRASS-Solver1 takes more total work to deliver good parallelism. With the techniques of PCF and SSCM, the total work of pGRASS-Solver2 is reduced, which is even smaller than that of the sequential feGRASS-PCG. We also notice that pGRASS-Solver2 achieves better parallel efficiency than pGRASS-Solver1. Using 16 threads, the parallel efficiency of

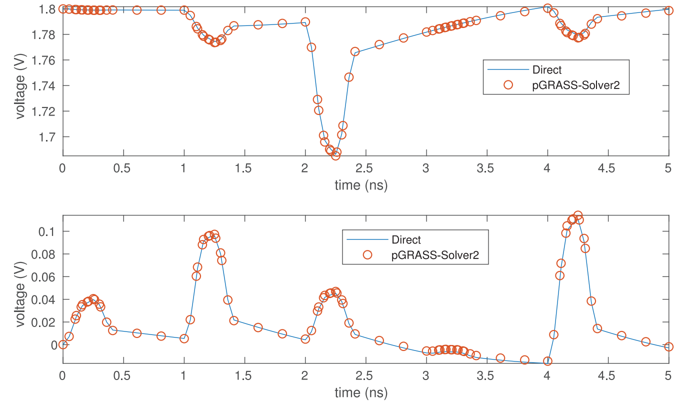

Fig. 5. Transient simulation results of a VDD node (up) and a GND node (down) in case ibmpg3t, obtained with direct equation solver and the proposed pGRASS-Solver2.

TABLE VII
RUNTIME OF TRANSIENT ANALYSIS ON THE CASE Thupg1t USING DIFFERENT NUMBER OF THREADS (TIME IN UNIT OF SECOND)

#Threads	1	2	4	8	16
feGRASS-PCG	269	-	-	-	-
pGRASS-Solver1	354	180	102	65.4	57.0
pGRASS-Solver2	231	125	65.3	41.6	29.2

pGRASS-Solver2 is 49.4% while that of pGRASS-Solver1 is only 38.8%. The reason is that solving the Schur complement matrix is the main bottleneck for parallelization and pGRASS-Solver2 employs the SSCM, which can be solved much more efficiently.

VI. CONCLUSION

This article presents a practically efficient parallel iterative solver (pGRASS-Solver) for large-scale power grid analysis problems. A parallel graph sparsification algorithm is first proposed, which employs a divide-and-conquer strategy to

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.

compute effective resistances and a parallel edge-recovering technique. Then, DDM is utilized to solve the sparsifier's Laplacian matrix. To further reduce the cost for factorization and solution of the sparsifier's Laplacian matrix, a variant of DDM which employs PCF and Schur complement matrix sparsification is proposed. Consequently, we obtain an efficient parallel preconditioner, which combines the advantages of both graph sparsification techniques and partition-based methods. For power grid DC analysis, experimental results show that an average 7.9× speedup is gained over the sequential feGRASS-based solver [2] and an average 6.8× speedup is gained over the parallel MST-guided method [1], on a 16-core machine. For power grid transient analysis, the proposed pGRASS-Solver achieves an average 8.6× speedup over the feGRASS-based solver and several tens times speedup over direct solver CHOLMOD [12]. For a real-world power grid matrix with 360 million nodes and 8.7 billion nonzeros, pGRASS-Solver succeeds in performing DC analysis on it within 20 min, which is 10.9× faster than the best sequential method.

References

[1] Y. Wang, W. Zhang, P. Li, and J. Gong, "Convergence-boosted graph partitioning using maximum spanning trees for iterative solution of large linear circuits," in *Proc. IEEE/ACM DAC*, 2017, pp. 1–6.

[2] Z. Liu, W. Yu, and Z. Feng, "feGRASS: Fast and effective graph spectral sparsification for scalable power grid analysis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 3, pp. 681–694, Mar. 2022.

[3] J. Yang, Z. Li, Y. Cai, and Q. Zhou, "PowerRush: An efficient simulator for static power grid analysis," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 10, pp. 2103–2116, Oct. 2014.

[4] K. Daloukas, N. Evmorfopoulos, P. Tsompanopoulou, and G. Stamoulis, "Parallel fast transform-based preconditioners for large-scale power grid analysis on graphics processing units (GPUs)," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1653–1666, Oct. 2016.

[5] J. M. Silva, J. R. Phillips, and L. M. Silveira, "Efficient simulation of power grids," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, no. 10, pp. 1523–1532, Oct. 2010.

[6] M. Zhao, R. Panda, S. Sapatnekar, and D. Blaauw, "Hierarchical analysis of power distribution networks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 2, pp. 159–168, Feb. 2002.

[7] Q. Zhou, K. Sun, K. Mohanram, and D. Sorensen, "Large power grid analysis using domain decomposition," in *Proc. Design Autom. Test Europe Conf.*, vol. 1, 2006, pp. 1–6.

[8] K. Sun, Q. Zhou, K. Mohanram, and D. C. Sorensen, "Parallel domain decomposition for simulation of large-scale power grids," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 2007, pp. 54–59.

[9] T. Yu, Z. Xiao, and M. D. F. Wong, "Efficient parallel power grid analysis via additive Schwarz method," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2012, pp. 399–406.

[10] G. Cui, W. Yu, X. Li, Z. Zeng, and B. Gu, "Machine-learning-driven matrix ordering for power grid analysis," in *Proc. Design, Autom. Test Europe Conf. Exhibition (DATE)*, 2019, pp. 984–987.

[11] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate," *ACM Trans. Math. Softw.*, vol. 35, no. 3, p. 22, 2008.

[12] T. Davis. "SuiteSparse." Accessed: 2022. [Online]. Available: http://faculty.cse.tamu.edu/davis/suitesparse.html

[13] R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadelphia, PA, USA: SIAM, 1994.

[14] Z. Zhu, B. Yao, and C.-K. Cheng, "Power network analysis using an adaptive algebraic multigrid approach," in *Proc. IEEE/ACM DAC*, 2003, pp. 105–108.

[15] C. Li, C. An, F. Yang, and X. Zeng, "ESPSim: An efficient scalable power grid simulator based on parallel algebraic multigrid," *ACM Trans. Design Autom. Electron. Syst.*, vol. 28, no. 1, p. 5, 2023. [Online]. Available: https://doi.org/10.1145/3529533

[16] X. Zhao, L. Han, and Z. Feng, "A performance-guided graph sparsification approach to scalable and robust SPICE-accurate integrated circuit simulations," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 10, pp. 1639–1651, Oct. 2015.

[17] Z. Feng, "Spectral graph sparsification in nearly-linear time leveraging efficient spectral perturbation analysis," in *Proc. IEEE/ACM DAC*, 2016, pp. 1–6.

[18] Z. Feng, "GRASS: Graph spectral sparsification leveraging scalable spectral perturbation analysis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 4944–4957, Dec. 2020.

[19] D. A. Spielman and S.-H. Teng, "Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems," *SIAM J. Matrix Anal. Appl.*, vol. 35, no. 3, pp. 835–885, 2014.

[20] I. Koutis, G. L. Miller, and R. Peng, "Approaching optimality for solving SDD linear systems," in *Proc. ACM FOCS*, 2010, pp. 235–244.

[21] J. Batson, D. Spielman, and N. Srivastava, "Twice-Ramanujan sparsifiers," in *Proc. ACM STOC*, 2009, pp. 255–262.

[22] D. A. Spielman and N. Srivastava, "Graph sparsification by effective resistances," *SIAM J. Comput.*, vol. 40, no. 6, pp. 1913–1926, 2011.

[23] J. Batson, D. A. Spielman, N. Srivastava, and S.-H. Teng, "Spectral sparsification of graphs: Theory and algorithms," *Commun. ACM*, vol. 56, no. 8, pp. 87–94, Aug. 2013.

[24] M. B. Cohen et al., "Solving SDD linear systems in nearly $m \log^{1/2} n$ time," in *Proc. ACM STOC*, 2014, pp. 343–352.

[25] Y. Zhang, Z. Zhao, and Z. Feng, "SF-GRASS: Solver-free graph spectral sparsification," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2020, pp. 1–8.

[26] Z. Liu and W. Yu, "Pursuing more effective graph spectral sparsifiers via approximate trace reduction," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, 2022, pp. 613–618.

[27] Z. Liu and W. Yu, "pGRASS-Solver: A parallel iterative solver for scalable power grid analysis based on graph spectral sparsification," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2021, pp. 1–9.

[28] S. R. Nassif. "IBM power grid benchmarks." Accessed: 2022. [Online]. Available: https://web.ece.ucsb.edu/ lip/PGBenchmarks/ibmpgbench.html

[29] J. Yang and Z. Li. "THU power grid benchmarks." Accessed: 2022. [Online]. Available: http://tiger.cs.tsinghua.edu.cn/PGBench/

[30] J. Yang, Z. Li, Y. Cai, and Q. Zhou, "PowerRush: Efficient transient simulation for power grid analysis," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2012, pp. 653–659.

[31] O. Axelsson and G. Lindskog, "On the rate of convergence of the preconditioned conjugate gradient method," *Numerische Mathematik*, vol. 48, no. 5, pp. 499–523, 1986.

[32] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: SIAM, 2003.

[33] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, "Internally deterministic parallel algorithms can be fast," in *Proc. ACM PPoPP*, 2012, pp. 181–192.

[34] H. N. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," in *Proc. ACM STOC*, 1983, pp. 246–251.

[35] "Intel oneAPI math kernel library." Accessed: 2022. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html

[36] "feGRASS: Fast and effective GRAph spectral sparsification." Accessed: 2022. [Online]. Available: http://numbda.cs.tsinghua.edu.cn/packages/feGRASSEXE.zip

[37] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.

Authorized licensed use limited to: Tsinghua University. Downloaded on August 30,2023 at 13:27:57 UTC from IEEE Xplore. Restrictions apply.