pGRASS-Solver: A Parallel Iterative Solver for Scalable Power Grid Analysis Based on Graph Spectral Sparsification

Zhiqiang Liu, Wenjian Yu
Dept. Computer Science & Tech., BNRist, Tsinghua University, Beijing 100084, China
Email: liu-zq20@mails.tsinghua.edu.cn, yu-wj@tsinghua.edu.cn

***Abstract*—Due to the rapid advance of the integrated circuit technology, power grid analysis usually imposes a severe computational challenge, where linear equations with millions or even billions of unknowns need to be solved. Recent graph spectral sparsification techniques have shown promising performance in accelerating power grid analysis. However, previous graph sparsification based iterative solvers are restricted by difficulty of parallelization. Existing graph sparsification algorithms are implemented under the assumption of serial computing, while factorization and backward/forward substitution of the sparsifier's Laplacian matrix are also hard to parallelize. On the other hand, partition based iterative methods which can be easily parallelized 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 for scalable power grid analysis by integrating graph sparsification techniques and partition based methods. We first propose a practically-efficient parallel graph sparsification algorithm. Then, domain decomposition method is leveraged to solve the sparsifier's Laplacian matrix. An efficient graph sparsification based parallel preconditioner is obtained, which not only leads to fast convergence but also enjoys ease of parallelization. Extensive experiments are carried out to demonstrate the superior efficiency of the proposed solver for large-scale power grid analysis, showing 5.2X speedup averagely over the state-of-the-art parallel iterative solver. Moreover, it solves a real-world power grid matrix with 0.36 billion nodes and 8.7 billion nonzeros within 23 minutes on a 16-core machine, which is 9.5X faster than the best result of sequential graph sparsification based solver.**

***Index Terms*—Graph spectral sparsification, iterative solver, power grid analysis, parallel computing, domain decomposition method, preconditioned conjugate gradient algorithm.**

I. Introduction

Accurate and efficient analysis of large-scale power grids is crucial for the modern very large-scale integrated (VLSI) circuits design. Large-scale power grid analysis requires solving linear equations with millions or even billions of unknowns, which is computationally challenging due to excessive time and memory consumption. Existing methods for power grid analysis include direct solvers, iterative solvers [1], [2] and other specialized methods such as the hierarchical matrix based method [3] and domain decomposition method (DDM) [4]–[8]. Direct methods, such as Cholesky or LU decomposition [9], [10], exactly solve the simulation problems but require much more memory to produce and store the matrix factors. On the other hand, iterative methods, such as the the Krylov subspace iterative methods [11] or algebraic multigrid (AMG) methods [12], usually have more favorable memory requirements thereby achieving more scalable performance for large problems. Among the most popular iterative methods, the preconditioned conjugate gradient (PCG) or generalized minimal residual (GMRES) algorithms leveraging recent graph spectral sparsification techniques have shown highly scalable performance for large circuit analysis tasks [13]–[16].

Graph spectral sparsification aims to find an ultra-sparse subgraph (called sparsifier) which can preserve the spectral properties of the original graph. In the past decades, spectral sparsification approaches have been extensively studied in both theory [17]–[20] and practice [14]–[16], [21]. GRASS proposed in [14], [15] is the first practically-efficient graph spectral sparsification algorithm , which leverages approximate dominant generalized eigenvectors for identifying and recovering spectrally-critical off-tree edges. It can produce high-quality spectral sparsifiers but strongly relies on the solution of Laplacian matrix, which can be computationally challenging for large problems. To overcome this difficulty, two different approaches were proposed in [16], [21]. SF-GRASS proposed in [21] leverages spectral graph coarsening and graph signal processing (GSP) techniques. feGRASS proposed in [16] is based on effective edge weight and spectral edge similarity. It can produce high-quality spectral sparsifiers in much shorter time than GRASS. Moreover, feGRASS based solver achieves better performance than GRASS based solver and also other PCG solvers such as AMG-PCG [16]. However, all these graph sparsification algorithms are implemented under the assumption of serial computing. Besides, graph sparsification based iterative solvers require factorization and backward/forward substitution of the sparsifier's Laplacian matrix, which are also hard to parallelize.

Partition based methods are another type of methods that are designed specifically for parallel computing. Domain decomposition method was first explored for parallel power grid analysis [4], [5], but forming and factorizing the dense Schur complement matrix can be even more time-consuming than solving the original equations. To overcome this difficulty, additive Schwarz method (ASM) was introduced in [6], [7], which utilizes overlapping domain decomposition to build a block-structure preconditioner. ASM can be easily parallelized and achieve scalable performance for large-scale problems but lacks a direct control of the relative condition number of the

978-1-6654-4507-8/21/$31.00 ©2021 IEEE

preconditioner. Recently, a new block Jacobi preconditioner was introduced in [22], which is based on partitioning the maximum spanning tree (MST) and then forming the block Jacobi preconditioner for parallel computing. It aims to integrate graph sparsification techniques and partition based methods to deliver good parallelism. However, the MST-guided method consumes more memory and requires more iteration steps to converge than graph sparsification based methods.

In this paper, we aim to develop a parallel iterative solver for large-scale power grid analysis based on graph sparsification. Our main contributions are summarized as follows.

1) We propose a practically-efficient parallel graph spectral sparsification algorithm called pGRASS, which is based on 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, which inherits the convergence property from graph sparsification techniques and the divide-and-conquer nature from DDM. It is also an explicit preconditioner which can be easily reused for the linear systems with multiple right-hand-sides.

3) Combining the two techniques, we have developed an efficient parallel iterative solver called pGRASS-Solver. Extensive experiments have been conducted to verify the efficiency of the proposed solver. The results show that pGRASS-Solver achieves an average 5.5X speedup over feGRASS based solver [16] and an average 5.2X speedup over the MST-guided method [22] for simulating 16 large-scale power grid benchmarks [23], [24]. Besides, pGRASS-Solver succeeds in solving a real-world power grid matrix with 0.36 billion nodes and 8.7 billion nonzeros within 23 minutes, which is 9.5X 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 8 billion nonzeros can be solved within half an hour on a 16-core machine.

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

II. Background

A. Graph Spectral Sparsification Based Iterative Solver

Consider a weighted, undirected graph $G = (V, E, w)$, where V and E are the sets of vertices (nodes) and edges, respectively. w is a positive weight function. We will use $w(e)$ or $w_{i,j}$ to denote the weight of the edge $e = (i, j)$. The Laplacian matrix of graph G is denoted by $L_G \in \mathbb{R}^{n\times n}$, where $n = |V|$.

$$L_G(i,j) = \begin{cases} -w_{i,j}, & (i,j) \in E \\ \sum_{(i,k)\in E} w_{i,k}, & i = j \\ 0, & \text{otherwise} . \end{cases} \quad (1)$$

In DC analysis the power grid is modeled as a resistive network which can be regarded as a weighted undirected graph. The Laplacian matrix excluding the row and column for ground node is the coefficient matrix of the linear equation system to be solved. It is a symmetric positive definite matrix, so that the PCG algorithm can be employed to solve the equation [11]:

$$Ax = b . \quad (2)$$

Here x denotes the unknown vector of node voltages. With B as the preconditioner, the PCG algorithm for solving (2) will find an ϵ-accurate solution in at most $O(\kappa(A, B)^{1/2} \log \frac{1}{\epsilon})$ iterations [25], where $\kappa(A, B)$ is the relative condition number of A and B .

Graph spectral sparsification aims to find an ultra-sparse subgraph P which is spectrally similar to the original graph G. Graph P is σ-spectrally similar to G if for any $u \in \mathbb{R}^n$ [26],

$$\frac{1}{\sigma} u^T L_P u \le u^T L_G u \le \sigma u^T L_P u , \quad (3)$$

where L_P is the Laplacian matrix of P. This infers that $\kappa(L_G, L_P) \le \sigma^2$, and taking L_P as the preconditioner the PCG algorithm will converge in at most $O(\sigma \log \frac{1}{\epsilon})$ iterations.

Among existing practically-efficient graph sparsification algorithms [14]–[16], [21], feGRASS [16] achieves a better tradeoff between runtime and quality of sparsifier. In this work, a parallel version of feGRASS will be presented in section III-B, so we briefly review the sequential feGRASS algorithm below.

feGRASS leverages the concept of effective edge weight to extract the maximum-effective-weight spanning tree (MEWST) as the backbone of sparsifier. *Effective edge weight* is defined as [16]:

$$W_{\text{eff}}(e) = w_{i,j} \times \frac{\log(\max\{deg(i), deg(j)\})}{dist(r,i) + dist(r,j)} , \quad (4)$$

where $deg(i)$ denotes the degree of vertex i, r is a root node of the tree, and $dist(r, i)$ denotes the unweighted distance between r and i which can be computed with breadth-first search (BFS). The average stretch of MEWST is usually lower than that of maximum-weight spanning tree (MWST), which infers that MEWST is spectrally more critical than MWST [16].

After obtaining a spectrally critical spanning tree, feGRASS calculates the spectral criticalities of all off-tree edges with $w_{i,j}R_P(i, j)$, where $R_P(i, j)$ denotes the effective resistance across i and j in P. Then off-tree edges are sorted by spectral criticalities. To further improve the quality of sparsifier, the drop of effective resistance during the edge-recovery procedure is inspected in feGRASS. For two off-tree edges $e_1 = (p, q)$

and $e_2 = (s,t)$, *spectral edge similarity* between them is defined as:

$$Similarity(e_1, e_2) = f_{p,q}^T L_P^+ f_{s,t} = f_{s,t}^T L_P^+ f_{p,q} \ , \quad (5)$$

where $f_{p,q} = f_p - f_q$, and f_p and f_q is the p-th and the q-th column of identity matrix, respectively. Spectral edge similarity reflects the drop of effective resistance across e_1 after recovering e_2. After adding an edge $e = (i,j)$ into P, the off-tree edges that have large similarity to e should be excluded during the subsequent edge-recovery steps. The off-tree edges that are similar to e can be obatined by running β-layer breadth-first search (BFS) from i and j respectively.

The algorithm flow of feGRASS is described in Algorithm 1.

Algorithm 1 feGRASS: fast and effective GRAph Spectral Sparsification [16]

Input: Graph $G = (V, E, w)$, the number of edges added to the spanning tree of G for producing the sparse graph: α.
Output: Sparse graph P, which is spectrally similar to G.

1: Run BFS to compute unweighted distances. Calculate effective edge weights via (4). Run the Kruskal algorithm to obtain MEWST T. Set $P = T$.
2: Compute effective resistances R_e of off-tree edges in T.
3: Sort off-tree edges by $w(e)R_e$ in descending order to get an edge list *OffTreeEgdes*.
4: **for** $k = 1$ to $|E| - |V| + 1$ **do**
5: **if** α edges have been added into P **then**
6: Break.
7: **end if**
8: Get edge $e =$*OffTreeEgdes*$[k]$.
9: **if** $e = (i,j)$ is not marked **then**
10: Add e into P.
11: Run β-layer BFS from i and j respectively. Mark the off-tree edges connecting the reached vertices in BFS and other vertices.
12: **end if**
13: **end for**

B. Domain Decomposition Method

Domain decomposition method (DDM) refers to the techniques of divide and conquer that have been primarily developed for solving partial differential equations. They are then generalized with concepts of graph partitioning. There are two standard ways of partitioning a graph [27]: vertex-based partitioning and edge-based partitioning. In this work, we focus on the vertex-based partitioning. Suppose the graph is partitioned into m subdomains. The nodes in each subdomain are classified into interior nodes and interface nodes, as shown in Fig. 1. After reordering the nodes, the equation (2) has blocked sparse structure:

$$\begin{bmatrix} A_1 & E_1 & O & O & \cdots & O & O \\ E_1^T & C_1 & O & F_{12} & \cdots & O & F_{1,m} \\ O & O & A_2 & E_2 & \cdots & O & O \\ O & F_{12}^T & E_2^T & C_2 & \cdots & O & F_{2,m} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ O & O & O & O & \cdots & A_m & E_m \\ O & F_{1,m}^T & O & F_{2,m}^T & \cdots & E_m^T & C_m \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \vdots \\ x_m \\ y_m \end{bmatrix} = \begin{bmatrix} f_1 \\ g_1 \\ f_2 \\ g_2 \\ \vdots \\ f_m \\ g_m \end{bmatrix} . \quad (6)$$

Here, we use O to denote zero matrix. x_i and y_i denote the unknowns on interior and interface nodes in the i-th subdomain, respectively. Matrices A_1, ..., A_m correspond to the interior nodes of m subdomains, and $C_1, ..., C_m$ correspond to the interface nodes. Matrices E_i reflect the connections between the interior nodes and the interface nodes in the i-th subdomain, and matrices $F_{i,j}$ reflect the connections between the interface nodes in the i-th subdomain and the j-th subdomain.

Through eliminating all interior nodes, (6) is transformed to:

$$Sy \equiv \begin{bmatrix} S_1 & F_{12} & \cdots & F_{1,m} \\ F_{12}^T & S_2 & \cdots & F_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ F_{1,m}^T & F_{2,m}^T & \cdots & S_m \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} g_1 - E_1^T A_1^{-1} f_1 \\ g_2 - E_2^T A_2^{-1} f_2 \\ \vdots \\ g_m - E_m^T A_m^{-1} f_m \end{bmatrix} , \quad (7)$$

where S is called overall Schur complement matrix and S_i satisfies:

$$S_i = C_i - E_i^T A_i^{-1} E_i \ . \quad (8)$$

After S is formed, we can solve (7) to obtain the interface variables y_i. Finally, solving the following equations to obtain the interior variables x_i:

$$A_i x_i = f_i - E_i y_i, \ i = 1, 2, ..., m. \quad (9)$$

Most steps in DDM can be easily parallelized. However, the efficiency of DDM is reduced as the number of interface nodes increases. When the number of interface nodes is large, forming and solving Schur complement matrix S can be even more time-consuming than solving the original matrix. On the other hand, reducing the number of interface nodes is also difficult as this requires advanced graph partitioning

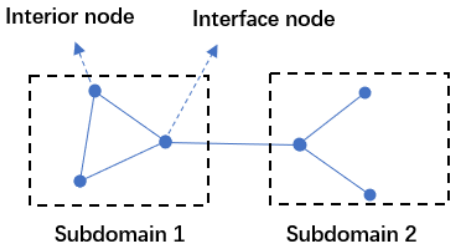

Fig. 1. A graph partitioned into 2 subdomains.

techniques. In this work, we address the problem by combining DDM with graph spectral sparsification. Instead of using DDM to solve Laplacian matrix of the orignal graph, we propose to use DDM to solve Laplacian matrix of the sparsifier. The subgraph is ultra-sparse, so the number of interface nodes can be very small. Thus, we obtain an efficient graph sparsification based parallel preconditioner.

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

In this section, we propose a parallel iterative solver for scalable power grid analysis, based on parallel graph spectral sparsification and domain decomposition method.

A. The Idea

Graph sparsification based PCG solver mainly includes three steps: (i) run graph spectral sparsification algorithm to obtain the spectrally similar ultra-sparse sparsifier, (ii) factorize the Laplacian matrix of sparsifier, (iii) run PCG algorithm with Cholesky factor of the sparsifier as the preconditioner. For a typical power grid analysis problem, the CPU time for the three steps is in the same order of magnitude, as reported in [16]. Take the case named "thupg10" in [24] as an example, feGRASS based PCG solver consumes 79.6s, 87.1s and 29.3s for the three steps respectively.

Existing graph sparsification based PCG solvers are fully serial. Consider the aforementioned three steps. For step (i), all the existing graph sparsification algorithms are implemented in serial. For step (ii) and step (iii), factorization and backward/forward substitution of the irregular ultra-sparse sparsifier are hard to parallelize. To address the first problem, we propose a parallel version of feGRASS algorithm, which is shown to be practically-efficient for large-scale power grid problems. To address the second issue, we observe that domain decomposition method can be efficient for solving ultra-sparse matrix because there are only a small amount of interface nodes. Thus, DDM can be leveraged to solve Laplacian matrix of the sparsifier. These two ideas will be presented in detail in the next two subsections, respectively.

B. Parallel Graph Spectral Sparsification

The sequential feGRASS algorithm (Algorithm 1) mainly includes four steps: (i) extract the maximum-effective-weight spanning tree (step 1 in Algorithm 1), (ii) calculate effective resistances of all off-tree edges (step 2 in Algorithm 1), (iii) sort off-tree edges (step 3 in Algorithm 1), (iv) recover off-tree edges (step 4-13 in Algorithm 1). Below we show how to parallelize each step to obtain a parallel graph sparsification algorithm.

Extracting the MEWST requires two basic graph algorithms: BFS and the Kruskal algorithm. Parallelization of these two algorithm has been extensively studied in parallel computing community. In this work, we just use the implementation in the *Problem Based Benchmark Suite (PBBS)* [28]. Parallel sorting is also well studied and we use the multiway mergesort implemented in C++ standard library.

To calculate effective resistances of all off-tree edges in parallel, we propose a simple but effective divide-and-conquer strategy. First partition the MEWST into a global subtree and m local subtrees, as shown in Fig. 2. Calculating effective resistance of one off-tree edge corresponds to *one query* for the distance between two endpoints of the edge. There are four types of queries, depending on locations of two endpoints, as shown in Fig. 2 with dotted lines. Consider calculating effective resistance of off-tree edge $e = (i, j)$. Let $R_{\mathit{eff}}(i, j)$ denote effective resistance of e, $T(i)$ 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 which vertex i belongs to, 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

$$R_{\mathit{eff}}(i, j) = D_0(i, j) \ . \tag{10}$$

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

$$R_{\mathit{eff}}(i, j) = D_{T(i)}(i, j) \ . \tag{11}$$

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

$$R_{\mathit{eff}}(i, j) = D_0(i, R_t(j)) + D_{T(j)}(R_t(j), j) \ . \tag{12}$$

If i and j belong to different local subtrees, then

$$\begin{aligned} R_{\mathit{eff}}(i, j) = {} & D_0(R_t(i), R_t(j)) + D_{T(i)}(R_t(i), i) \\ & + D_{T(j)}(R_t(j), j) \ . \end{aligned} \tag{13}$$

With (10) to (13), any query in original MEWST can be reduced to the queries in subtrees. The queries in one subtree can be computed efficiently using Tarjan's off-line least common ancestor (LCA) algorithm [29]. Then queries in different subtrees can be handled in parallel.

Recovering off-tree edges seems inherently serial, because whether to recover an off-tree edge depends on the decisions made earlier. To parallelize the edge-recovering phase, we first observe that for two off-tree edges $e_1 = (p, q)$ and $e_2 = (s, t)$,

$$\mathit{Similarity}(e_1, e_2) = f_{p,q}^T L_P^+ f_{s,t} \approx f_{p,q}^T L_T^+ f_{s,t} \ , \tag{14}$$

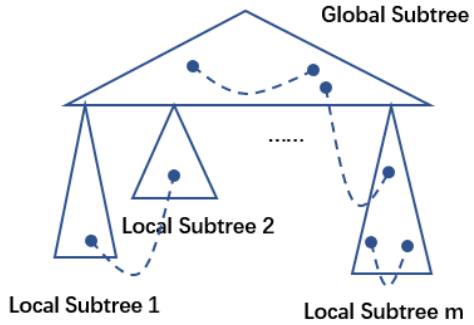

Fig. 2. A spanning tree, which is partitioned into a global subtree and m local subtrees.

where T denotes the spanning tree and P denotes the current subgraph. Eq. (14) infers that spectrally similar off-tree edges can be found by running β-layer BFS on the original spanning tree which is static during the edge-recovering procedure. Thus, a naive approach is computing spectrally similar edges for each off-tree edge in parallel first, storing them, then executing the sequential edge-recovering phase in Algorithm 1 except that spectrally similar edges are obtained by reading the data stored in advance. Because the main work in edge-recovering phase is computing spectrally similar edges, this strategy can achieve fairly good speedups.

However, storing spectrally similar edges of all off-tree edges may cause large amount of memory usage, making the aforementioned approach impractical. To address this issue, 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 in parallel, store them, execute the sequential edge-recovering procedure and then move to next block. The memory used for storing the spectrally similar edges can therefore be reused.

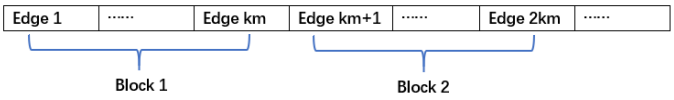

Fig. 3. Off-tree edges, divided into many blocks.

We combine the techniques proposed in this subsection to obtain a parallel graph spectral sparsification algorithm called pGRASS. It is described in Algorithm 2.

C. Solving the Sparsifier's Laplacian Matrix with DDM

Since the sparsifier constructed by graph sparsification is ultra-sparse, there are only a small amount of interface nodes when it is partitioned. The efficiency of domain decomposition method does not deteriorate for large-scale problems. So in this subsection, we propose to use domain decomposition method to solve the sparsifier's Laplacian matrix. Thus we obtain an efficient graph sparsification based parallel preconditioner.

The sparsifier is constructed by recovering α off-tree edges to the spanning tree. Consider partitioning the sparsifier into m subdomains, where we adopt the vertex-based partitioning. First partition the spanning tree into m subdomains, which can be done by simply removing $m-1$ edges. This results in $2(m-1)$ interface nodes. Then recover α off-tree edges to the spanning tree. One recovered edge results in at most two more interface nodes. So the number of interface nodes is at most $2(m-1)+2\alpha$. For example, if α is set to $0.02|V|$, which is a typical value in our experiment, the number of interface nodes does not exceed $2(m-1)+0.04|V| \approx 0.04|V|$.

Forming the Schur complement matrix in (7) can be parallelized, because S_i can be calculated using (8) in parallel. However, calculating S_i using (8) directly is time-consuming

Algorithm 2 pGRASS: parallel GRAph Spectral Sparsification

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

Output: Sparse graph P, which is spectrally similar to G.

```
Run parallel BFS to compute unweighted distances. Calculate effective edge weights via (4) in parallel. Run the parallel Kruskal algorithm to obtain MEWST T. Set P = T.
Partition T into a global subtree and m local subtrees. Use (10) to (13) to convert distance queries in T to distance queries in those subtrees. Run Tarjan's LCA algorithm for each subtree in parallel.
Sort off-tree edges by w(e)R_e in descending order in parallel.
Divide off-tree edges into many blocks such that each block except the last one contains km edges.
for each block do
    for each edge e = (i, j) in the current block in parallel do
        if e is not marked then
            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.
        end if
    end for
    for each edge e in the current block do
        if α edges have been added into P then
            Return.
        end if
        if e is not marked then
            Add e into P.
            Mark the spectrally similar edges of e.
        end if
    end for
end for
```

because the operation $A_i^{-1}E_i$ requires solving A_i many times. We show below how S_i can be calculated more efficiently.

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

$$\begin{bmatrix} P_iA_iP_i^T & P_iE_i \\ E_i^TP_i^T & C_i \end{bmatrix} = \begin{bmatrix} L_{11} & O \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ O & L_{22}^T \end{bmatrix} , \quad (15)$$

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

From (15) we can get:

$$P_iA_iP_i^T = L_{11}L_{11}^T , \quad (16)$$

$$P_iE_i = L_{11}L_{21}^T , \quad (17)$$

and

$$C_i = L_{21}L_{21}^T + L_{22}L_{22}^T \ . \tag{18}$$

Substituting (16) to (18) into (8), we have:

$$\begin{aligned} S_i &= C_i - E_i^T A_i^{-1} E_i \\ &= L_{21}L_{21}^T + L_{22}L_{22}^T \\ &- (P_i^T L_{11} L_{21}^T)^T (P_i^T L_{11} L_{11}^T P_i)^{-1} P_i^T L_{11} L_{21}^T \\ &= L_{22}L_{22}^T \ . \end{aligned} \tag{19}$$

Eq. (19) infers that S_i can be calculated by multiplying the submatrices in the Cholesky factor in (15). Besides, there is no need to factorize A_i again because the Cholesky factor of A_i is already obtained in (16).

After Schur complement matrix S is formed and factorized, we obtain an efficient parallel preconditioner where the preconditioner equation is solved with DDM as described in Algorithm 3. It is actually an explicit preconditioner, because the Cholesky factors of S and A_i are constructed explicitly and can be easily reused.

The method described here inherits the convergence property from graph sparsification techniques and the divide-and-conquer nature from DDM. The MST-guided method proposed in [22] is another method aiming at integrating graph sparsification techniques and partition based methods. It partitions the maximum spanning tree and then recovers all inner-partition off-tree edges to form a block Jacobi preconditioner. There are two major differences between the both methods. Firstly, the MST-guided method discards some spectrally critical inter-partition edges, while our method fully inherits the convergence property from graph sparsification techniques and thus leads to faster convergence of iterative solution. Secondly, the MST-guided method recovers all inner-partition edges forming 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.

Algorithm 3 Parallel Solution of the Laplacian-Matrix Equation (6) Obtained with DDM

Input: Laplacian-matrix equation provided in the form of (6). Cholesky factors of A_i, and S defined by (7) and (8).

Output: Solution x_i and y_i of (6).

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 (6).
3: **end for**
4: Use the Cholesky factor of S to solve (7).
5: **for** each subdomain i in parallel **do**
6: Use the Cholesky factor of A_i to solve (9) for x_i.
7: **end for**

D. The Overall Algorithm

The overall flow of the proposed parallel iterative solver (pGRASS-Solver) is described below.

1. Run the pGRASS algorithm (Algorithm 2) parallelly to obtain the sparsifier.
2. With DDM, partition the sparsifier into m subdomains.
3. Form the sparsifier's Laplacian matrix in form of (6), parallelly.
4. Factorize each subdomain matrix in the form of (15), parallelly. Then, compute S_i for each subdomain in parallel using (19), $i = 1, \cdots, m$.
5. Factorize the overall Schur complement matrix S in (7).
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 3.

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

IV. Experimental Results

We first validate the ideas proposed in section III-B and section III-C respectively. Then, the overall pGRASS-Solver is compared with the feGRASS based PCG solver [16] and the MST-guided method [22]. We have implemented pGRASS-Solver, feGRASS based PCG solver [16] and the MST-guided method [22] in C++. For Cholesky factorization, we use the state-of-the-art direct sparse solver CHOLMOD [9], [10]. For graph partitioning, we use the widely adopted graph partitioner METIS [30]. 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.

A. Validation of the Parallel Graph Sparsification

To validate the ideas proposed in section III-B, we compare the pGRASS algorithm (Algorithm 2) with the feGRASS algorithm (Algorithm 1). 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^{-3} with the sparsifier as preconditioner. pGRASS is executed in parallel using 16 threads while feGRASS is executed serially. The runtime (denoted by T_s), the relative condition number (denoted by κ) and the number of iteration steps (denoted by N_i) are listed in Table I.

From the results we see that, pGRASS algorithm (Algorithm 2) achieves 6.1X speedups over sequential feGRASS (Algorithm 1) algorithm on average. For large-scale cases, the speedup is up to 8.2X. 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 (14).

B. Validation of DDM Based Sparsifier Factorization

To validate the idea that DDM can be efficient for the ultra-sparse sparsifier, we compare two scenarios where DDM is

TABLE I
COMPARISON BETWEEN PARALLEL AND SEQUENTIAL GRAPH SPARSIFICATION ALGORITHMS FOR THE TEST CASES IN [23] AND [24]. (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	17	0.10	49.7	13	3.5
ibmpg4	1.0E6	4.1E6	0.39	59.2	2	0.11	64.2	3	3.5
ibmpg5	1.1E6	4.3E6	0.45	136	20	0.12	131	20	3.8
ibmpg6	1.7E6	6.6E6	0.75	226	32	0.18	197	31	4.2
ibmpg7	1.5E6	6.2E6	0.63	134	17	0.14	66.8	13	4.5
ibmpg8	1.5E6	6.2E6	0.64	132	17	0.15	72.3	14	4.3
thupg1	5.0E6	2.1E7	4.13	183	13	0.68	198	13	6.1
thupg2	8.9E6	3.9E7	8.13	277	14	1.25	209	13	6.5
thupg3	1.2E7	5.1E7	13.1	233	13	1.69	203	13	7.8
thupg4	1.5E7	6.6E7	17.0	211	12	2.19	214	12	7.8
thupg5	1.9E7	8.5E7	24.3	225	12	3.06	213	12	7.9
thupg6	2.4E7	1.1E8	25.3	216	13	3.73	208	12	6.8
thupg7	2.8E7	1.2E8	34.0	231	12	4.42	234	13	7.7
thupg8	4.0E7	1.8E8	50.8	223	12	6.85	297	13	7.4
thupg9	5.2E7	2.2E8	65.3	234	12	7.94	232	12	8.2
thupg10	6.0E7	2.6E8	79.6	242	12	9.72	267	12	8.2
Average	-	-	-	-	-	-	-	-	6.1

used for solving the original Laplacian matrix and for solving the sparsifier's Laplacian matrix. The dimension of Schur complement matrix (denoted by N_{Sch}) and the number of nonzeros (denoted by NNZ_{Sch}) are recorded. Table II also lists the time for forming Schur complement matrix (denoted by T_{Sch}) and the time for factorizing it (denoted by T_{Schfac}). "-" means that it takes more than 3600 seconds.

TABLE II
COMPARISON BETWEEN DDM FOR THE ORIGINAL MATRIX AND FOR THE PRECONDITIONER MATRIX. (TIME IN UNIT OF SECOND)

Case	DDM for original				DDM for sparsifier			
	N_{Sch}	NNZ_{Sch}	T_{Sch}	T_{Schfac}	N_{Sch}	NNZ_{Sch}	T_{Sch}	T_{Schfac}
ibmpg3	1.1E4	7.4E6	3.52	2.61	1.0E3	5.9E4	0.05	0.01
ibmpg4	1.3E4	1.1E7	6.46	7.66	1.1E3	6.7E4	0.06	0.01
ibmpg5	0.9E4	4.5E6	1.51	1.50	548	2.0E4	0.06	0.002
ibmpg6	0.8E4	4.0E6	1.82	1.38	555	2.0E4	0.10	0.02
ibmpg7	1.4E4	1.3E7	7.24	20.6	1.1E3	7.3E4	0.10	0.01
ibmpg8	1.4E4	1.3E7	7.17	10.9	1.1E3	7.3E4	0.10	0.01
thupg1	2.8E4	4.6E7	25.5	1326	3.4E3	7.4E5	0.39	0.12
thupg2	3.8E4	9.0E7	72.4	-	4.5E3	1.3E6	0.71	0.31
thupg3	4.4E4	1.2E8	111	-	5.2E3	1.8E6	1.02	0.49
thupg4	4.9E4	1.5E8	157	-	6.0E3	2.4E6	1.37	0.71
thupg5	5.7E4	2.0E8	240	-	6.7E3	3.0E6	1.97	0.95
thupg6	6.5E4	2.6E8	391	-	7.2E3	3.4E6	2.29	1.00
thupg7	6.8E4	3.0E8	417	-	8.1E3	4.2E6	2.96	1.51
thupg8	8.7E4	4.7E8	712	-	9.7E3	6.2E6	4.98	3.49
thupg9	8.8E4	4.8E8	1043	-	1.1E4	8.0E6	6.63	2.40
thupg10	9.9E4	6.2E8	1506	-	1.2E4	9.0E6	7.99	5.59

From the results we see that, utilizing DDM for solving the original linear equations is impractical for large-scale problems because forming and factorizing Schur complement matrix can be extremely time-consuming. When DDM is leveraged to solve the sparsifier's Laplacian matrix, the dimension of Schur complement matrix is reduced by about an order of magnitude, and the number of nonzeros is reduced by about two orders of magnitude. Thus Schur complement matrix can be formed and factorized much more efficiently. Take the case named "thupg10" as an example, it only takes 7.99 seconds to form Schur complement matrix in pGRASS-Solver, which is 188X faster than that in traditional parallel DDM.

C. Comparison with Two Recent PCG Solvers

We compare pGRASS-Solver with two recent work: feGRASS based solver [16] and the MST-guided method [22]. feGRASS based solver is sequential while the other two are parallel. The relative tolerance of PCG algorithm is set to 10^{-3}. For both graph sparsification algorithms, $2\%|V|$ off-tree edges are recovered. The performance of three solvers for the test cases in [23] and [24] is listed in Table III. 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 based solver, 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-Solver, 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 the MST-guided method and pGRASS-Solver, the time for graph partitioning is not included in the factorization time as in [22]. Sp_1 and Sp_2 denote the speedup ratios of pGRASS-Solver over feGRASS based solver and the MST-guided method respectively.

Compared with feGRASS based solver, pGRASS-Solver shows more than 6X improvement in factorization time for all test cases. This shows that, forming and factorizing Schur complement matrix for the sparsifier are highly efficient using the techniques described in section III-C. For the iteration phase, only about 3X speedups are obtained, which is due to the high memory access to computation ratio of sparse matrix computation routines. Overall, pGRASS-Solver can achieve 5.5X speedups on average with respect to sequential feGRASS based solver for the total time.

Compared with the parallel MST-guided method [22], pGRASS-Solver achieves an average 5.2X speedup for the total time. For the factorization phase, although both methods partition the matrix into 16 subdomain matrices and factorize each subdomain matrix in a single thread, pGRASS-Solver shows great improvement in factorization time. This is because the subdomain matrices in pGRASS-Solver are ultra-sparse after graph sparsification and can be factorized more efficiently. pGRASS-Solver also shows great improvement in iteration time, which is due to the following two reasons. Firstly, 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. Secondly, the preconditioner in pGRASS-Solver can be solved less costly because of fewer nonzeros in Cholesky factors, which leads to faster single-step iteration. We also note that pGRASS-Solver is more memory-efficient than the MST-

TABLE III
PERFORMANCE OF THREE PCG SOLVERS FOR THE TEST CASES IN [23] AND [24]. (TIME IN UNIT OF SECOND)

Case	feGRASS-PCG [16]					MST-Guided [22]				pGRASS-Solver					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
ibmpg3	0.35	0.62	0.43	17	1.40	0.56	0.82	40	1.38	0.10	0.06	0.14	13	0.30	4.7	4.6
ibmpg4	0.39	0.69	0.07	2	1.15	1.44	0.73	28	2.17	0.11	0.07	0.06	3	0.24	4.8	9.0
ibmpg5	0.45	0.72	0.69	20	1.86	0.31	0.89	48	1.20	0.12	0.06	0.25	20	0.43	4.3	2.8
ibmpg6	0.75	1.21	1.65	32	3.61	0.51	1.42	42	1.93	0.18	0.12	0.70	31	1.00	3.6	1.9
ibmpg7	0.63	1.11	0.90	17	2.64	1.44	1.13	28	2.57	0.14	0.11	0.29	13	0.54	4.9	4.8
ibmpg8	0.64	1.10	0.84	17	2.58	1.64	1.17	30	2.81	0.15	0.11	0.24	14	0.50	5.2	5.6
thupg1	4.13	6.06	2.51	13	12.7	4.72	2.97	21	7.69	0.68	0.51	1.14	13	2.33	5.5	3.3
thupg2	8.13	11.4	4.90	14	24.4	9.61	5.58	21	15.2	1.25	1.02	1.79	13	4.06	6.0	3.7
thupg3	13.1	15.3	6.00	13	34.4	15.2	7.36	21	22.6	1.69	1.51	2.33	13	5.53	6.2	4.1
thupg4	17.0	19.9	7.25	12	44.2	20.7	10.2	23	30.9	2.19	2.08	2.80	12	7.07	6.3	4.4
thupg5	24.3	25.6	9.79	12	59.7	32.0	13.9	20	45.9	3.06	2.92	3.40	12	9.38	6.4	4.9
thupg6	25.3	31.4	12.0	13	68.7	47.6	14.0	18	61.6	3.73	3.29	4.49	12	11.5	6.0	5.4
thupg7	34.0	38.7	13.5	12	86.2	76.7	17.3	15	94.0	4.42	4.47	4.69	13	13.6	6.3	6.9
thupg8	50.8	55.6	20.6	12	127	102	22.0	14	124	6.85	8.47	6.51	13	21.8	5.8	5.7
thupg9	65.3	72.7	25.0	12	163	202	33.1	15	235	7.94	9.03	8.88	12	25.9	6.3	9.1
thupg10	79.6	87.1	29.3	12	196	227	33.0	13	260	9.72	13.6	10.4	12	33.7	5.8	7.7
Average	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**5.5**	**5.2**

Sp_1 and Sp_2 denote the speedup ratios of pGRASS-Solver over feGRASS-PCG and the MST-Guided method, respectively.

guided method. Take the case "thupg10" as an example. MST-guided method consumes 62 GB memory, while pGRASS-Solver only uses 18 GB.

D. Results on a Real-World Power Grid with 360 million Nodes and 4.2 Billion Edges

A power grid circuit for the design of a flat panel display (with 2160×3840 pixels) from [16] is tested. It includes 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 two methods, $2\%|V|$ off-tree edges are recovered to construct the ultra-sparse sparsifier. pGRASS-Solver is executed with 8 and 16 threads respectively. The results are listed in Table IV. We note that the feGRASS based solver with $2\%|V|$ off-tree edges recovered runs faster than the results reported in [16]. Using 16 threads, pGRASS produces the sparsifier in 755 seconds, which is 11X faster than feGRASS algorithm. As for the total time, pGRASS-Solver achieves 9.5X speedups over feGRASS based PCG solver. As far as we know, it is the first time that a power grid matrix containing more than 8 billion nonzeros is solved within half an hour on a 16-core machine.

TABLE IV
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-Solver(m=8)	1098	227	728	69	2053 (34 mins)
pGRASS-Solver(m=16)	755	138	477	69	1370 (23 mins)
Speedup	11.1	4.4	8.5	-	9.5

V. CONCLUSIONS

This paper presents a practically-efficient parallel iterative solver (pGRASS-Solver) for large-scale power grid analysis problems. We first propose a parallel graph sparsification algorithm, based on a divide-and-conquer strategy to compute effective resistances and a parallel edge-recovering technique. Then, domain decomposition method is utilized to solve the sparsifier's Laplacian matrix. Therefore, we obtain a graph sparsification based parallel preconditioner, which combines the advantages of both graph sparsification technique and domain decomposition method. Experimental results on 16 large-scale power grid benchmarks show that an average 5.5X speedup is gained over the sequential feGRASS based solver [16] and an average 5.2X speedup is gained over the parallel MST-guided method [22], on a 16-core machine. For a real-world power grid matrix with 360 million nodes and 8.7 billion nonzeros, pGRASS-Solver succeeds in solving it within 23 minutes, which is 9.5X faster than the best sequential method. It is the first time that a power grid matrix with more than 8 billion nonzeros can be solved within half an hour on a 16-core machine.

VI. ACKNOWLEDGMENT

This work is supported by National Key R&D Program of China (2019YFB2205002), NSFC under grant No. 62090025, and BNRist under grant No. BNR2019ZS01001.

REFERENCES

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

[2] 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 Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1653–1666, 2016.

[3] J. M. Silva, J. R. Phillips, and L. M. Silveira, "Efficient simulation of power grids," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 10, pp. 1523–1532, 2010.

[4] M. Zhao, R. Panda, S. Sapatnekar, and D. Blaauw, "Hierarchical analysis of power distribution networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 2, pp. 159–168, 2002.

[5] Q. Zhou, K. Sun, K. Mohanram, and D. Sorensen, "Large power grid analysis using domain decomposition," in *Proceedings of the Design Automation Test in Europe Conference*, vol. 1, 2006, pp. 1–6.

[6] K. Sun, Q. Zhou, K. Mohanram, and D. C. Sorensen, "Parallel domain decomposition for simulation of large-scale power grids," in *IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 54–59.

[7] T. Yu, Z. Xiao, and M. D. F. Wong, "Efficient parallel power grid analysis via Additive Schwarz Method," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2012, pp. 399–406.

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

[9] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: CHOLMOD, supernodal sparse cholesky factorization and update/downdate," *ACM Transactions on Mathematical Software*, vol. 35, no. 3, p. 22, 2008.

[10] T. Davis, "SuiteSparse." [Online]. Available: http://faculty.cse.tamu.edu/davis/suitesparse.html

[11] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: Building blocks for iterative methods*. SIAM, 1994.

[12] 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.

[13] X. Zhao, L. Han, and Z. Feng, "A performance-guided graph sparsification approach to scalable and robust SPICE-accurate integrated circuit simulations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1639–1651, 2015.

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

[15] ——, "GRASS: graph spectral sparsification leveraging scalable spectral perturbation analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4944–4957, 2020.

[16] Z. Liu, W. Yu, and Z. Feng, "feGRASS: fast and effective graph spectral sparsification for scalable power grid analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2021. [Online]. Available: 10.1109/TCAD.2021.3060647

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

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

[19] M. B. Cohen, R. Kyng, G. L. Miller, J. W. Pachocki, R. Peng, A. B. Rao, and S. C. Xu, "Solving SDD linear systems in nearly $m \log^{1/2} n$ time," in *Proc. ACM STOC*, 2014, p. 343–352.

[20] D. A. Spielman and N. Srivastava, "Graph sparsification by effective resistances," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1913–1926, 2011.

[21] Y. Zhang, Z. Zhao, and Z. Feng, "SF-GRASS: solver-free graph spectral sparsification," in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–8.

[22] 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.

[23] "IBM power grid benchmarks." [Online]. Available: https://web.ece.ucsb.edu/~lip/PGBenchmarks/ibmpgbench.html

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

[25] 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.

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

[27] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Society for Industrial and Applied Mathematics, 2003.

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

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

[30] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.