Nested Dissection Based Parallel Transient Power Grid Analysis on Public Cloud Virtual Machines

Jiawen Cheng, Zhiqiang Liu, Lingjie Li, Wenjian Yu*
Dept. Computer Science & Tech., BNRist, Tsinghua Univ., Beijing, China.
Email: cjw21@mails.tsinghua.edu.cn, yu-wj@tsinghua.edu.cn

***Abstract*—Accurate and efficient transient analysis of power grids (PGs) poses a large challenge of computation for nowadays integrated circuit design. In this work, we propose to leverage the public cloud computing to do PG transient analysis while preserving security. A multi-level distributed parallel LU factorization and forward/backward substitution approach based on nested dissection is then proposed to guarantee accuracy and robustness. Experimental results show that the proposed algorithm can achieve an average 2.06X speedup over NICSLU and 2.85X over conventional domain decomposition method based parallel approach. And, it exhibits good scalability with up to 6.0X parallel speedup on large-scale PGs with 4 cloud computer nodes.**

I. Introduction

Power grid (PG) analysis is increasingly important for designing and verifying the on-chip power delivery system for nowadays low-power integrated circuits (ICs). It includes two kinds of analysis: direct current (DC) analysis and transient analysis. The former regards the PG as a resistive network and ignores capacitors and inductors, while the latter takes them all into account. The major purpose of PG analysis is to compute the voltage drop (i.e. IR drop) at PG nodes under various current load assumptions which reflect different working modes of the circuit. Compared to DC analysis, the transient analysis is more useful to validating the PG design. Due to the increased number of circuit components, the decreased margin for IR drop and the demand of accurate analysis, the transient analysis usually consumes a large portion of time and poses a large challenge of computation for nowadays IC design.

For the transient analysis, a system of first-order ordinary differential equations (ODEs) derived from modified nodal analysis (MNA) need to be solved. With the implicit time integration method, e.g., backward Euler or trapezoidal rule, the problem is converted to the solution of linear equation systems at successive time points. Therefore, the direct solver or iterative solver for sparse linear equation system are employed for efficient PG analysis. The direct solver based method is suitable for transient analysis when the time step is fixed, which only performs the expensive LU or Cholesky factorization of the matrix once at the beginning. The direct sparse solvers which have been used for circuit simulation include SuperLU series [1], CHOLMOD [2], KLU [3], NICSLU [4] and CKTSO [5], etc.

The iterative solver based method is better suited for the cases with variable time steps because its efficiency is hardly affected by the change of coefficient matrices for different time points. It also consumes less memory so that it better scales to larger problems than the direct solver. Popular iterative solver employs the Krylov subspace iterative methods [6] with various preconditioning techniques like algebraic multigrid method [7], [8], incomplete LU factorization [9] and graph spectral sparsification [10], [11]. Recently, iterative solvers incorporating graph spectral sparsification techniques have demonstrated highly scalable performance for PG analysis. However, their efficiency relies on the quality of preconditioner, which cannot be robust enough on various problems. For achieving high accuracy during a long-time transient analysis, a tight convergence criterion of iterative solver is also required, which degrades the advantage of iterative method over the direct solver based method. Therefore, an efficient and robust method for transient analysis of large PGs is still desired.

Distributed parallel computing and cloud computing are the technology which can break the computation challenges faced by the time-consuming tasks in electronic design automation (EDA). Domain decomposition method (DDM) is a specialized method developed to explore the parallelism in circuit simulation and PG analysis [12], [13], [11]. Existing work mostly consider the DDM based iterative solver for achieving high parallelism for PG analysis. In [13], the parallel DDM is developed for the DC analysis of very large PGs on distributed computing platform. However, it is on a customized cluster with high-speed Infiniband networks.

Nowadays, public cloud computing platforms have become economic and available computing infrastructure facilitating various computation tasks. However, deploying EDA tools to them is not widely accepted due to the concern of security and actual benefit. In this work, we consider leveraging the public cloud computing to expedite the time-consuming transient PG analysis while preserving security and robustness. An efficient distributed parallel algorithm based on nested dissection, a variant of DDM based direct solver, is proposed to accelerate the transient analysis of large-scale PGs, at the small cost of cloud virtual machine rent. The major contributions of this work are as follows.

1) An idea of expediting transient PG analysis with public cloud computing is proposed. With the circuit netlist parsed locally and the matrices and the source waveforms uploaded to cloud computing platform, it guarantees the security.

2) A multi-level distributed parallel LU factorization algorithm and forward/backward substitution algorithm based on nested dissection is proposed. The relatively small size of the interfaces at each level allows for better parallelism. And, its nature of the direct solver guarantees accuracy and robustness.

3) Only a few vectors are transferred between the processes at each time point of forward/backward substitution, which yields a low communication volume. Therefore, the proposed

979-8-3503-9354-5/24/$31.00 ©2024 IEEE

algorithm can be deployed on a cloud virtual machine cluster without high-speed networks such as InfiniBand.

4) An efficient scheme is designed for updating the right-hand side (RHS) of the linear system in parallel at each time point. It takes advantage of the fact that processes on the same computer node can share memory. This scheme further reduces computation and communication overhead.

Experiments on transient analysis of 9 practical PGs are conducted on a public cloud computing platform with Ethernet. The results show that the proposed algorithm achieves an average 2.06X speedup compared to NICSLU on a single computer node and an average 2.84X speedup compared to DDM on multiple computer nodes. And, the proposed algorithm exhibits good scalability, with up to 6.0X speedup over the serial simulation on the large-scale PGs.

II. Preliminaries

A. Transient Analysis of Power Grid

In the transient analysis, the PG is modeled as a resistor-inductor-capacitor (RLC) network. Fig. 1 presents a schematic illustration of the detailed RLC model of IBM PG [14].

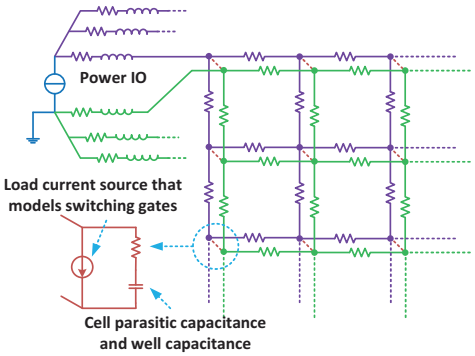


Fig. 1. The RLC model of IBM PG [15].

The ODE system derived from MNA can be formulated as

$$\boldsymbol{C}\dot{\boldsymbol{x}}(t) + \boldsymbol{G}\boldsymbol{x}(t) = \boldsymbol{B}\boldsymbol{u}(t), \tag{1}$$

where $\boldsymbol{C} \in \mathbb{R}^{n\times n}, \boldsymbol{G} \in \mathbb{R}^{n\times n}, \boldsymbol{B} \in \mathbb{R}^{n\times p}$ denote the capacitance and inductance matrix, conductance matrix, and input selector matrix, respectively. $\boldsymbol{u}(t) \in \mathbb{R}^p$ represents the vector of p input sources and $\boldsymbol{x}(t) \in \mathbb{R}^n$ represents the solution vector of node voltages and branch currents.

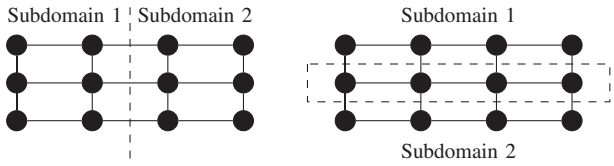
By applying the trapezoidal rule, one can convert (1) into the task of solving the following linear equation system at each time point.

$$(\frac{\boldsymbol{C}}{h}+\frac{\boldsymbol{G}}{2})\boldsymbol{x}(t+h) = (\frac{\boldsymbol{C}}{h}-\frac{\boldsymbol{G}}{2})\boldsymbol{x}(t)+\boldsymbol{B}\frac{\boldsymbol{u}(t+h)+\boldsymbol{u}(t)}{2}, \tag{2}$$

where h is the time step in transient analysis. We can further use $\boldsymbol{A}$ to denote $\frac{\boldsymbol{C}}{h}+\frac{\boldsymbol{G}}{2}$ and $\boldsymbol{b}$ to denote the RHS in (2). So, (2) is rewritten as $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. When h is fixed, matrix $\boldsymbol{A}$ is a constant at different time points, so that applying the direct solver is preferred as only one expensive matrix factorization is required.

B. Domain Decomposition Method

Domain decomposition method (DDM) refers to a series of divide-and-conquer techniques based on graph partitioning for solving linear equations [12]. In this work, we focus on the DDM aided direct solver for symmetric linear equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, instead of the DDM based iterative solver. The symmetric matrix $\boldsymbol{A}$ naturally corresponds to an undirected graph. As shown in Fig. 2, the DDM can be realized with two kinds of separators: edge separator and vertex separator. The edge separator is an edge subset satisfying that the removal of it makes the graph divided into m unconnected components of similar size called subdomains. The vertex separator is defined similarly. The separator nodes are also known as the interface.



(a) Edge separator (b) Vertex separator

Fig. 2. An example of partitioning a graph into two subdomains with (a) an edge separator or (b) a vertex separator.

The DDM with edge or vertex separator is widely recognized as the graph partition problem. Suppose the graph is partitioned into m subdomains. For any value of m, it can be efficiently solved with algorithms like METIS [16]. When $m = 2$, the DDM with vertex separator forms a dissection of vertices. When it is recursively executed for each subdomain, the nested dissection enables an effective reordering approach for sparse direct solver [17].

With DDM, the $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ problem can be rewritten as the equation with blocked sparse coefficient matrix:

$$\begin{pmatrix} \boldsymbol{D}_1 & & & \boldsymbol{F}_1 \\ & \boldsymbol{D}_2 & & \boldsymbol{F}_2 \\ & & \ddots & \vdots \\ & & \boldsymbol{D}_m & \boldsymbol{F}_m \\ \boldsymbol{E}_1 & \boldsymbol{E}_2 & \cdots \boldsymbol{E}_m & \boldsymbol{S} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_m \\ \boldsymbol{y} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}_1 \\ \boldsymbol{f}_2 \\ \vdots \\ \boldsymbol{f}_m \\ \boldsymbol{g} \end{pmatrix}, \tag{3}$$

where the internal nodes in each subdomain correspond to the diagonal blocks $\boldsymbol{D}_1, \boldsymbol{D}_2, \cdots, \boldsymbol{D}_m$, and $\boldsymbol{S}$ corresponds to the separator nodes. Connection matrices $\boldsymbol{E}_i = \boldsymbol{F}_i^T, i = 1, \cdots, m$ reflect the connections between the internal nodes and the separator nodes.

To simplify notation, we rewrite (3) as

$$\begin{pmatrix} \boldsymbol{D} & \boldsymbol{F} \\ \boldsymbol{E} & \boldsymbol{S} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{pmatrix}, \tag{4}$$

where $\boldsymbol{D}$ is a block diagonal matrix. By eliminating all internal nodes, (4) becomes

$$(\boldsymbol{S} - \boldsymbol{E}\boldsymbol{D}^{-1}\boldsymbol{F})\boldsymbol{y} = \boldsymbol{g} - \boldsymbol{E}\boldsymbol{D}^{-1}\boldsymbol{f}. \tag{5}$$

The dense matrix $\boldsymbol{S}-\boldsymbol{E}\boldsymbol{D}^{-1}\boldsymbol{F}$ is called the Schur complement matrix. Finally, the internal unknowns can be solved with

$$\boldsymbol{D}_i\boldsymbol{x}_i = \boldsymbol{f} - \boldsymbol{F}_i\boldsymbol{y}. \tag{6}$$

The opportunities for parallelism are solving $\boldsymbol{D}^{-1}\boldsymbol{F}$, $\boldsymbol{D}^{-1}\boldsymbol{f}$ and (6). As the number of subdomains m increases, the size of the interface also increase and solving (5) becomes a computational bottleneck. The above analysis also works for the DDM with edge separator, where the size of Schur complement is even larger. In [11], the DDM with edge separator is employed to parallelly solve the preconditioner matrix in the iterative solver based method. However, the DDM based parallel direct solver for PG analysis is less explored.

III. Methodology

In order not to expose the internal structure of circuit, its netlist should be parsed locally. Then, according to (2), the data uploaded to the public cloud computing platform are the matrices $\boldsymbol{C}, \boldsymbol{G}, \boldsymbol{B}$, the source waveforms $\boldsymbol{u}(t)$ and the initial state of $\boldsymbol{x}$. The coefficient matrix $\frac{\boldsymbol{C}}{h} + \frac{\boldsymbol{G}}{2}$ is symmetric but not necessarily positive definite. Therefore, an LU factorization based on nested dissection is performed on it, followed by a forward/backward substitution and an RHS updating at each time step.

A. The Idea of Using Nested Dissection

Since DDM forms a large dense Schur complement with many subdomains, we consider recursive partitioning of the matrix inspired by nested dissection. In this way, the size of the interface can be effectively reduced, which can improve the parallel efficiency. Specifically, consider a matrix partitioned into two subdomains and an interface. After reordering, its LU factorization should satisfy

$$\begin{pmatrix} \boldsymbol{D}_1 & & \boldsymbol{F}_1 \\ & \boldsymbol{D}_2 & \boldsymbol{F}_2 \\ \boldsymbol{E}_1 & \boldsymbol{E}_2 & \boldsymbol{S} \end{pmatrix} = \begin{pmatrix} \boldsymbol{L}_{11} & & \\ & \boldsymbol{L}_{22} & \\ \boldsymbol{L}_{31} & \boldsymbol{L}_{32} & \boldsymbol{L}_{33} \end{pmatrix} \begin{pmatrix} \boldsymbol{U}_{11} & & \boldsymbol{U}_{13} \\ & \boldsymbol{U}_{22} & \boldsymbol{U}_{23} \\ & & \boldsymbol{U}_{33} \end{pmatrix}. \tag{7}$$

Suppose the LU factorization of $\boldsymbol{D}_1, \boldsymbol{D}_2$ is known, then the unknowns in $\boldsymbol{L}$ and $\boldsymbol{U}$ factors can be solved by

$$\boldsymbol{L}_{31} = \boldsymbol{E}_1 \boldsymbol{U}_{11}^{-1}, \boldsymbol{L}_{32} = \boldsymbol{E}_2 \boldsymbol{U}_{22}^{-1}, \tag{8}$$

$$\boldsymbol{U}_{13} = \boldsymbol{L}_{11}^{-1} \boldsymbol{F}_1, \boldsymbol{U}_{23} = \boldsymbol{L}_{22}^{-1} \boldsymbol{F}_2, \tag{9}$$

and a LU factorization of the Schur complement

$$\boldsymbol{L}_{33}\boldsymbol{U}_{33} = \boldsymbol{S} - \boldsymbol{L}_{31}\boldsymbol{U}_{13} - \boldsymbol{L}_{32}\boldsymbol{U}_{23}. \tag{10}$$

The factorization of $\boldsymbol{D}_1, \boldsymbol{D}_2$ can be done recursively in the same manner, so that the coefficient matrix is reordered into the form of nest dissection. The opportunities for distributed parallelism are the factorization of $\boldsymbol{D}_1, \boldsymbol{D}_2$, the solution of (8) and the solution of (9). The factorization of the Schur complement is executed on a single process, but the cost is acceptable because of its relatively small size.

For forward substitution, the equations to be solved is

$$\begin{pmatrix} \boldsymbol{L}_{11} & & \\ & \boldsymbol{L}_{22} & \\ \boldsymbol{L}_{31} & \boldsymbol{L}_{32} & \boldsymbol{L}_{33} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \boldsymbol{y} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}_1 \\ \boldsymbol{f}_2 \\ \boldsymbol{g} \end{pmatrix}. \tag{11}$$

The solution can be obtained by solving

$$\boldsymbol{L}_{11}\boldsymbol{x}_1 = \boldsymbol{f}_1, \boldsymbol{L}_{22}\boldsymbol{x}_2 = \boldsymbol{f}_2, \boldsymbol{L}_{33}\boldsymbol{y} = \boldsymbol{g} - \boldsymbol{L}_{31}\boldsymbol{x}_1 - \boldsymbol{L}_{32}\boldsymbol{x}_2. \tag{12}$$

Because $\boldsymbol{L}_{11}, \boldsymbol{L}_{22}$ conform the nested dissection form, the solution of $\boldsymbol{x}_1, \boldsymbol{x}_2$ can be performed recursively in parallel. The solution of $\boldsymbol{x}_3$ is performed on a single process. For backward substitution, the solution can be obtained similarly.

B. Parallel Reordering and Factorization

To reorder the matrix into the nested dissection form, a multi-level parallel algorithm is adopted. Suppose the maximum nested level is l_m, then the total number of processes is 2^{l_m}. For the top level l_m, only process 0 partitions and reorders the entire matrix. For the next level $l_m - 1$, process 0 and process 2^{l_m-1} partition each of the two subdomains derived from the level l_m in parallel. The matrix reordering proceeds similarly for the remaining levels except level 0. For level l, processes whose numbers divide 2^l can perform matrix partitioning in parallel. For level 0, a permutation vector is computed for reducing fill-ins. Fig. 3 illustrates an example of task assignment for 3-level matrix reordering, where the number in each node indicates the number of the process executing the task.

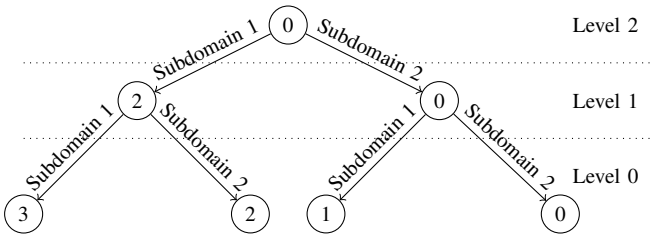


Fig. 3. An example of task assignment for 3-level matrix reordering.

The above matrix reordering algorithm is summarized as Algorithm 1. Given the maximum nested level l_m, the algorithm uses four vectors f, g, h, e of length $l_m + 1$ to record partition information of each level for subsequent factorization and solving, where f, g, h denote the offsets of subdomain 1, subdomain 2 and the interface in the entire matrix, respectively, and e denotes the end of the submatrix. The superscript (r) means the data resides in process r and the subscript l means the data describes information of level l.

Algorithm 1 Parallelly reorder $\boldsymbol{A}$ with the nested dissection approach

Input: Matrix $\boldsymbol{A}$, nested level l_m.

Output: Partition vectors $f^{(r)}, g^{(r)}, h^{(r)}, e^{(r)}$, reordered matrix $\boldsymbol{A}$.

$\boldsymbol{A}^{(0)} \leftarrow \boldsymbol{A}$.
for each process r **do**
  Initialize 4 integer vectors $f^{(r)}, g^{(r)}, h^{(r)}, e^{(r)}$, each of length $l_m + 1$ and filled with 0.
  Initialize an empty local permutation vector $\boldsymbol{p}^{(r)}$.
  for $l = l_m$ to 1 **do**
    if r divides 2^l **then**
      Compute the vertex separator of $\boldsymbol{A}^{(r)}$ to obtain vectors P_1, P_2, S, which stand for the indices of the separated parts and the vertex separator, respectively.
      Prepend S to $\boldsymbol{p}^{(r)}$.
      $g_l^{(r)} \leftarrow f_l^{(r)} + |P_1|, h_l^{(r)} \leftarrow g_l^{(r)} + |P_2|$.
      $e_l^{(r)} \leftarrow f_l^{(r)} + \text{NROW}(\boldsymbol{A}^{(r)})$.
      Send matrix $\boldsymbol{A}^{(r)}(P_1, P_1)$ and integer $f_l^{(r)}$ to process $r + 2^{l-1}$. ▷ *send subdomain 1 to left child*
      $\boldsymbol{A}^{(r)} \leftarrow \boldsymbol{A}^{(r)}(P_2, P_2), f_{l-1}^{(r)} \leftarrow g_l^{(r)}$.
      Broadcast $f_l^{(r)}, g_l^{(r)}, h_l^{(r)}, e_l^{(r)}$ to process $r + 1, r + 2, \cdots, r + 2^l - 1$.
    else if r divides 2^{l-1} **then**
      $\boldsymbol{A}^{(r)}, f_{l-1}^{(r)} \leftarrow$ matrix and integer received from process $r - 2^{l-1}$. ▷ *receive subdomain 1 from parent*
  Compute fill-in reducing permutation vector $\boldsymbol{p}_f$ for $\boldsymbol{A}^{(r)}$.
  Prepend $\boldsymbol{p}_f$ to $\boldsymbol{p}^{(r)}$.
  $e_0^{(r)} \leftarrow f_0^{(r)} + \text{NROW}(\boldsymbol{A}^{(r)})$.
Gather $\boldsymbol{p}^{(r)}$ to all processes and obtain $\boldsymbol{p}$.
Reorder $\boldsymbol{A}$ with $\boldsymbol{p}$.

The reordered matrix corresponding to Fig. 3 is

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{D}^{(3)} & & \boldsymbol{F}_1^{(3)} & & & & \boldsymbol{F}_2^{(3)} \\ & \boldsymbol{D}^{(2)} & \boldsymbol{F}_1^{(2)} & & & & \boldsymbol{F}_2^{(2)} \\ \boldsymbol{E}_1^{(3)} & \boldsymbol{E}_1^{(2)} & \boldsymbol{S}_1^{(2)} & & & & \boldsymbol{F}_{2,1}^{(2)} \\ \xleftarrow{\text{offset of } P_1\ (f_1^{(0)})} & & & \boldsymbol{D}^{(1)} & & \boldsymbol{F}_1^{(1)} & \boldsymbol{F}_2^{(1)} \\ \xleftarrow{\text{offset of } P_2\ (g_1^{(0)})} & & & & \boldsymbol{D}^{(0)} & \boldsymbol{F}_1^{(0)} & \boldsymbol{F}_2^{(0)} \\ \xleftarrow{\text{offset of } S\ (h_1^{(0)})} & & & & & & \\ & & & \boldsymbol{E}_1^{(1)} & \boldsymbol{E}_1^{(0)} & \boldsymbol{S}_1^{(0)} & \boldsymbol{F}_{2,1}^{(0)} \\ \xleftarrow{\text{end of submatrix } (e_1^{(0)})} & & & & & & \\ \boldsymbol{E}_2^{(3)} & \boldsymbol{E}_2^{(2)} & \boldsymbol{E}_{2,1}^{(2)} & \boldsymbol{E}_2^{(1)} & \boldsymbol{E}_2^{(0)} & \boldsymbol{E}_{2,1}^{(0)} & \boldsymbol{S}_2^{(0)} \end{pmatrix}, \quad (13)$$

where the meanings of $f_1^{(0)}, g_1^{(0)}, h_1^{(0)}, e_1^{(0)}$ are also illustrated in it. $\boldsymbol{D}^{(r)}$ corresponds to the internal nodes in level 0. $\boldsymbol{E}_1^{(r)}$ and $\boldsymbol{F}_1^{(r)}$ denote the connection matrices in level 1. $\boldsymbol{S}_1^{(r)}$ corresponds to the separator nodes in level 1, which is owned by only half of the processes. For level 2 or higher, the connection matrices are reinterpreted as block matrices for subsequent factorizing. For example, $\boldsymbol{E}_2^{(0)}, \cdots, \boldsymbol{E}_2^{(3)}$ and $\boldsymbol{F}_2^{(0)}, \cdots, \boldsymbol{F}_2^{(3)}$ in (13) denote the blocks corresponding to the internal nodes, i.e., they have the same row or column ranges as $\boldsymbol{D}^{(1)}, \cdots, \boldsymbol{D}^{(4)}$, respectively. $\boldsymbol{E}_{2,1}^{(0)}, \boldsymbol{E}_{2,1}^{(2)}$ and $\boldsymbol{F}_{2,1}^{(0)}, \boldsymbol{F}_{2,1}^{(2)}$ in (13) denote the blocks corresponding to the separator nodes in level 1, i.e., they have the same row or column ranges as $\boldsymbol{S}_1^{(0)}, \boldsymbol{S}_1^{(2)}$, respectively.

After reordered, the matrix is factorized level by level from the bottom up. The $\boldsymbol{L}$ and $\boldsymbol{U}$ factors of (13) satisfy the form of (14) and (15), respectively.

$$\boldsymbol{L} = \begin{pmatrix} \begin{pmatrix} \boldsymbol{L}_0^{(3)} & & \\ & \boldsymbol{L}_0^{(2)} & \\ \boldsymbol{L}_1^{(3)} & \boldsymbol{L}_1^{(2)} & \tilde{\boldsymbol{L}}_1^{(2)} \end{pmatrix} & & \\ & \begin{pmatrix} \boldsymbol{L}_0^{(1)} & & \\ & \boldsymbol{L}_0^{(0)} & \\ \boldsymbol{L}_1^{(1)} & \boldsymbol{L}_1^{(0)} & \tilde{\boldsymbol{L}}_1^{(0)} \end{pmatrix} & \\ \boldsymbol{L}_2^{(2)} & \boldsymbol{L}_2^{(0)} & \tilde{\boldsymbol{L}}_2^{(0)} \end{pmatrix}. \quad (14)$$

$$\boldsymbol{U} = \begin{pmatrix} \begin{pmatrix} \boldsymbol{U}_0^{(3)} & & \boldsymbol{U}_1^{(3)} \\ & \boldsymbol{U}_0^{(2)} & \boldsymbol{U}_1^{(2)} \\ & & \tilde{\boldsymbol{U}}_1^{(2)} \end{pmatrix} & & \boldsymbol{U}_2^{(2)} \\ & \begin{pmatrix} \boldsymbol{U}_0^{(1)} & & \boldsymbol{U}_1^{(1)} \\ & \boldsymbol{U}_0^{(0)} & \boldsymbol{U}_1^{(0)} \\ & & \tilde{\boldsymbol{U}}_1^{(0)} \end{pmatrix} & \boldsymbol{L}_2^{(0)} \\ & & \tilde{\boldsymbol{U}}_2^{(0)} \end{pmatrix}. \quad (15)$$

The task assignment remains as shown in Fig. 3. For $\boldsymbol{D}^{(r)}$, process r in level 0 performs LU factorization on it and obtains $\boldsymbol{L}_0^{(r)}, \boldsymbol{U}_0^{(r)}$, so it can be done completely in parallel. Then, each process solves for the connection matrices $\boldsymbol{L}_1^{(r)}, \boldsymbol{U}_1^{(r)}$ with (8) and (9) in parallel. For level 1, each odd-numbered process r sends $\boldsymbol{L}_1^{(r)}\boldsymbol{U}_1^{(r)}$ to process $r-1$. Then, each even-numbered process r in level 1 solves for the separator nodes $\tilde{\boldsymbol{L}}_1^{(r)}, \tilde{\boldsymbol{U}}_1^{(r)}$ with (10) in parallel. For level 2 or higher, the connection matrices are in the form of nested dissection, thus they are likewise solved in the multi-level way. That is, the blocks corresponding to the internal nodes are solved first, and then the blocks corresponding to the separators are solved level by level. At each level, several blocks are transferred between the processes similarly and concatenated to form the connection matrices residing in a single process. Solving for the separator nodes in level 2 or higher is the same as level 1.

The above LU factorization algorithm is summarized as Algorithm 2. Lines 2 and 3 describe the factorization of $\boldsymbol{D}^{(r)}$ in level 0. Lines 5 to 9 describe solving for the blocks of $\boldsymbol{L}_l^{(r)}, \boldsymbol{U}_l^{(r)}$ corresponding to the internal nodes. Lines 10 to 20 describe solving for the blocks of $\boldsymbol{L}_l^{(r)}, \boldsymbol{U}_l^{(r)}$ corresponding to the separator nodes. Lines 22 to 28 describe the factorization of the separator nodes in level l. Note that lines 7 and 9 solve lower triangular linear systems with multiple sparse RHS, which can be done efficiently with Gilbert/Peierls' algorithm [18]. The parallelism comes from irrelevance of solving each column of the RHS. For task scheduling, an atomic integer is used to balance the workload of each thread.

Algorithm 2 Distributed parallel LU factorization $\boldsymbol{A} = \boldsymbol{LU}$

Input: Partition vectors $f^{(r)}$, $g^{(r)}, h^{(r)}, e^{(r)}$, matrix $\boldsymbol{A}$, level l_m, .

Output: Factors $\boldsymbol{L}, \boldsymbol{U}$ distributed on each process.

1: **for** each process r **do**
2: $\quad \boldsymbol{D}^{(r)} \leftarrow \boldsymbol{A}(f_0^{(r)} : e_0^{(r)}, f_0^{(r)} : e_0^{(r)})$.
3: $\quad$ Perform LU factorization $\boldsymbol{D}^{(r)} = \boldsymbol{L}_0^{(r)}\boldsymbol{U}_0^{(r)}$.
4: $\quad$ **for** $l = 1$ to l_m **do**
5: $\quad\quad$ ▷ *solve for the internal nodes in* $\boldsymbol{L}_l^{(r)}, \boldsymbol{U}_l^{(r)}$ ◁
6: $\quad\quad \boldsymbol{E}_l^{(r)} \leftarrow \boldsymbol{A}(h_l^{(r)} : e_l^{(r)}, f_0^{(r)} : e_0^{(r)})$.
7: $\quad\quad$ Solve $\boldsymbol{L}_l^{(r)}\boldsymbol{U}_0^{(r)} = \boldsymbol{E}_l^{(r)}$ for $\boldsymbol{L}_l^{(r)}$.
8: $\quad\quad \boldsymbol{F}_l^{(r)} \leftarrow \boldsymbol{A}(f_0^{(r)} : e_0^{(r)}, h_l^{(r)} : e_l^{(r)})$.
9: $\quad\quad$ Solve $\boldsymbol{L}_0^{(r)}\boldsymbol{U}_l^{(r)} = \boldsymbol{F}_l^{(r)}$ for $\boldsymbol{U}_l^{(r)}$.
10: $\quad\quad$ ▷ *solve for the separator nodes in* $\boldsymbol{L}_l^{(r)}, \boldsymbol{U}_l^{(r)}$ ◁
11: $\quad\quad$ **for** $k = 1$ to $l-1$ **do**
12: $\quad\quad\quad$ **if** r divides 2^k **then**
13: $\quad\quad\quad\quad$ $\boldsymbol{M}_1, \boldsymbol{L}', \boldsymbol{M}_2, \boldsymbol{U}' \leftarrow$ matrices received from process $r + 2^{k-1}$.
14: $\quad\quad\quad\quad \boldsymbol{E}_{l,k}^{(r)} \leftarrow \boldsymbol{A}(h_l^{(r)} : e_l^{(r)}, h_k^{(r)} : e_k^{(r)})$.
15: $\quad\quad\quad\quad$ Solve $\boldsymbol{S}_1\tilde{\boldsymbol{U}}_k^{(r)} = \boldsymbol{E}_{l,k}^{(r)} - \boldsymbol{L}_l^{(r)}\boldsymbol{U}_k^{(r)} - \boldsymbol{M}_1$ for $\boldsymbol{S}_1$.
16: $\quad\quad\quad\quad \boldsymbol{F}_{l,k}^{(r)} \leftarrow \boldsymbol{A}(h_k^{(r)} : e_k^{(r)}, h_l^{(r)} : e_l^{(r)})$.
17: $\quad\quad\quad\quad$ Solve $\tilde{\boldsymbol{L}}_k^{(r)}\boldsymbol{S}_2 = \boldsymbol{F}_{l,k}^{(r)} - \boldsymbol{L}_k^{(r)}\boldsymbol{U}_l^{(r)} - \boldsymbol{M}_2$ for $\boldsymbol{S}_2$.
18: $\quad\quad\quad\quad \boldsymbol{L}_l^{(r)} \leftarrow [\boldsymbol{L}'\ \boldsymbol{L}_l^{(r)}\ \boldsymbol{S}_1], \boldsymbol{U}_l^{(r)} \leftarrow \begin{bmatrix} \boldsymbol{U}' \\ \boldsymbol{U}_l^{(r)} \\ \boldsymbol{S}_2 \end{bmatrix}$.
19: $\quad\quad\quad$ **else if** r divides 2^{k-1} **then**
20: $\quad\quad\quad\quad$ Send matrices $\boldsymbol{L}_l^{(r)}\boldsymbol{U}_k^{(r)}, \boldsymbol{L}_l^{(r)}, \boldsymbol{L}_k^{(r)}\boldsymbol{U}_l^{(r)}, \boldsymbol{U}_l^{(r)}$ to process $r - 2^{k-1}$.
21: $\quad\quad$ ▷ *solve for* $\tilde{\boldsymbol{L}}_l^{(r)}, \tilde{\boldsymbol{U}}_l^{(r)}$ ◁
22: $\quad\quad$ **if** r divides 2^l **then**
23: $\quad\quad\quad$ $\boldsymbol{M} \leftarrow$ matrix received from process $r + 2^{l-1}$.
24: $\quad\quad\quad \boldsymbol{S}_l^{(r)} \leftarrow \boldsymbol{A}(h_l^{(r)} : e_l^{(r)}, h_l^{(r)} : e_l^{(r)})$.
25: $\quad\quad\quad \boldsymbol{S} \leftarrow \boldsymbol{S}_l^{(r)} - \boldsymbol{L}_l^{(r)}\boldsymbol{U}_l^{(r)} - \boldsymbol{M}$.
26: $\quad\quad\quad$ Perform LU factorization $\boldsymbol{S} = \tilde{\boldsymbol{L}}_l^{(r)}\tilde{\boldsymbol{U}}_l^{(r)}$.
27: $\quad\quad$ **else if** r divides 2^{l-1} **then**
28: $\quad\quad\quad$ Send matrix $\boldsymbol{L}_l^{(r)}\boldsymbol{U}_l^{(r)}$ to process $r - 2^{l-1}$.

C. Parallel Forward/Backward Substitution

With the distributed factors $\boldsymbol{L}$ and $\boldsymbol{U}$ obtained by Algorithm (2), a lower triangular system $\boldsymbol{Ly} = \boldsymbol{b}$ and an upper triangular

system $\boldsymbol{Ux} = \boldsymbol{y}$ are solved per time point. The lower triangular system corresponding to (14) is

$$\begin{pmatrix} \begin{pmatrix} \boldsymbol{L}_0^{(3)} & & \\ & \boldsymbol{L}_0^{(2)} & \\ \boldsymbol{L}_1^{(3)} & \boldsymbol{L}_1^{(2)} & \tilde{\boldsymbol{L}}_1^{(2)} \end{pmatrix} & & \\ & \begin{pmatrix} \boldsymbol{L}_0^{(1)} & \\ & \boldsymbol{L}_0^{(0)} \\ \boldsymbol{L}_1^{(1)} & \boldsymbol{L}_1^{(0)} \; \tilde{\boldsymbol{L}}_1^{(0)} \end{pmatrix} & \\ \boldsymbol{L}_2^{(2)} & & \boldsymbol{L}_2^{(0)} \; \tilde{\boldsymbol{L}}_2^{(0)} \end{pmatrix} \begin{pmatrix} \boldsymbol{y}_0^{(3)} \\ \boldsymbol{y}_0^{(2)} \\ \boldsymbol{y}_1^{(2)} \\ \boldsymbol{y}_0^{(1)} \\ \boldsymbol{y}_0^{(0)} \\ \boldsymbol{y}_1^{(0)} \\ \boldsymbol{y}_2^{(0)} \end{pmatrix} = \begin{pmatrix} \boldsymbol{b}_0^{(3)} \\ \boldsymbol{b}_0^{(2)} \\ \boldsymbol{b}_1^{(2)} \\ \boldsymbol{b}_0^{(1)} \\ \boldsymbol{b}_0^{(0)} \\ \boldsymbol{b}_1^{(0)} \\ \boldsymbol{b}_2^{(0)} \end{pmatrix}. \tag{16}$$

A multi-level parallel algorithm based on the task assignment illustrated in Fig. 3 is proposed for forward/backward substitution of the triangular equations. The forward substitution is bottom-up. Each process r in level 0 solves for the segment $\boldsymbol{y}_0^{(r)}$ with a single-process forward substitution $\boldsymbol{L}_0^{(r)}\boldsymbol{y}_0^{(r)} = \boldsymbol{b}_0^{(r)}$ in parallel. For the remaining levels, processes whose numbers divide 2^l in level l solve for the segment $\boldsymbol{y}_l^{(r)}$ with

$$\tilde{\boldsymbol{L}}_l^{(r)}\boldsymbol{y}_l^{(r)} = \boldsymbol{b}_l^{(r)} - \boldsymbol{L}_l^{(r)}\boldsymbol{y}_l^{(r)} - \boldsymbol{L}_l^{(r+2^{l-1})}\boldsymbol{y}_l^{(r+2^{l-1})}. \tag{17}$$

The above algorithm is summarized as Algorithm 3. Line 8 is the implementation of (17). The algorithm for backward substitution is very similar to the algorithm for forward substitution. The only difference is that the backward substitution is top-down, so we omit it here for brevity. It can be deduced that each process r depends on the RHS segment $\boldsymbol{b}(f_0^{(r-1)} : f_0^{(r)})$ for forward/backward substitution and owns the solution vector segment $\boldsymbol{x}(f_0^{(r-1)} : f_0^{(r)})$ afterwards.

Algorithm 3 Distributed parallel forward substitution $\boldsymbol{Ly} = \boldsymbol{b}$

Input: Partition vectors $f^{(r)}, g^{(r)}, h^{(r)}, e^{(r)}$, factors $\boldsymbol{L}$ distributed on each process, RHS vector $\boldsymbol{b}$, nested level l_m.
Output: Solution $\boldsymbol{y}$ distributed on each process.

1: **for** each process r **do**
2: ▷ *solve for the internal nodes in* $\boldsymbol{y}$ ◁
3: Solve $\boldsymbol{L}_0^{(r)}\boldsymbol{y}^{(r)}(f_0^{(r)} : e_0^{(r)}) = \boldsymbol{b}^{(r)}(f_0^{(r)} : e_0^{(r)})$.
4: ▷ *solve for the separator nodes in* $\boldsymbol{y}$ ◁
5: **for** $l = 1$ to l_m **do**
6: **if** r divides 2^l **then**
7: $\boldsymbol{m}, \boldsymbol{y}' \leftarrow$ vectors received from process $r + 2^{l-1}$.
8: Solve $\tilde{\boldsymbol{L}}_l^{(r)}\boldsymbol{y}^{(r)}(h_l^{(r)} : e_l^{(r)}) = \boldsymbol{b}^{(r)}(h_l^{(r)} : e_l^{(r)}) - \boldsymbol{L}_l^{(r)}\boldsymbol{y}^{(r)}(g_l^{(r)} : h_l^{(r)}) - \boldsymbol{m}$.
9: $\boldsymbol{y}^{(r)}(f_l^{(r)} : g_l^{(r)}) \leftarrow \boldsymbol{y}'$.
10: **else if** r divides 2^{l-1} **then**
11: $\boldsymbol{y}' \leftarrow \boldsymbol{y}^{(r)}(f_l^{(r)} : g_l^{(r)})$.
12: Send vectors $\boldsymbol{L}_l^{(r)}\boldsymbol{y}', \boldsymbol{y}'$ to process $r - 2^{l-1}$.

Finally, the RHS $\boldsymbol{b}$ is updated for the next time point, which mainly consists of two time-consuming sparse matrix-vector multiplications (SpMV) $(\frac{\boldsymbol{C}}{h} - \frac{\boldsymbol{G}}{2})\boldsymbol{x}(t)$ and $\boldsymbol{B}\frac{\boldsymbol{u}(t+h)+\boldsymbol{u}(t)}{2}$. An efficient scheme illustrated in Fig. 4 is designed for computing a general SpMV $\boldsymbol{v} = \boldsymbol{Mu}$ in parallel. Note that each process r depends on the RHS segment $\boldsymbol{b}^{(r)}$ and owns the solution vector segment $\boldsymbol{x}^{(r)}$. Therefore, $\boldsymbol{M}$ is split by rows and each process r own the block $\boldsymbol{M}^{(r)}$. The segments $\boldsymbol{u}^{(r)}$ are gathered to all processes, which derives the entire vector $\boldsymbol{u}$ identical on all processes. The segment $\boldsymbol{v}^{(r)}$ can be obtained by $\boldsymbol{M}^{(r)}\boldsymbol{u}$ where the coefficient matrix $\boldsymbol{M}^{(r)}$ contains much fewer rows. This scheme uses an `Allgatherv` operation, which brings a huge overhead when the number of processes is large. To tackle this issue, shared memory is allocated for the solution vector segments on the same computer node. This avoids needless communication between processes on the same computer node.

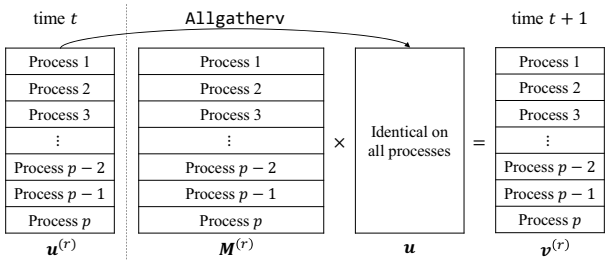


Fig. 4. An efficient scheme for parallel sparse matrix-vector multiplication.

To demonstrate the low communication volume of the proposed forward/backward substitution algorithm and RHS updating scheme, a theoretical analysis is given here. Suppose the size of the coefficient matrix is N, the nested level is l_m, the number of computer nodes is n, and the bitwidth of nonzeros is 8. For forward/backward substitution, several segments of the solution are transferred between the processes in level 1 to l_m, which add up to the entire vector and yields a communication volume of $8Nl_m$. For RHS updating, each computer node owns $1/n$ of the solution vector and gather them to all nodes, which yields a communication volume of $8N(n-1)$. Therefore, the total communication volume at each time point is $8N(l_m + n - 1)$. For a typical running configuration of 128 processes on 4 computer nodes, the communication volume is only 10 times of the solution vector data.

IV. Experimental Results

To validate the performance of the proposed algorithm, experiments are conducted on transient analysis of 4 middle-scale PGs (ibmpg3t-ibmpg6t) from the IBMPG benchmarks [14] and 5 large-scale PGs (thupg1t, thupg3t, thupg5t, thupg8t, thupg10t) derived from the THUPG benchmarks [19]. For all of the test cases from THUPG benchmarks, capacitors and inductors with random value are added and periodic trapezoidal pulse currents are used as the input sources. The time step is 10ps and the total number of time points is 500. We have implemented the proposed algorithm in C++ and compared it with edge separator based DDM and NICSLU [4], a high-performance parallel direct solver based on shared memory architecture. The three implementations take matrix files ($\boldsymbol{C}, \boldsymbol{G}, \boldsymbol{B}$ and the source waveforms $\boldsymbol{u}(t)$) as input and output the solutions at each time point. The initial state of $\boldsymbol{x}$ is set to all 0. Process-level parallelism is realized by OpenMPI [20] and thread-level parallel matrix operations are realized by Eigen library [21] and Intel Math Kernel Library (MKL) [22]. For graph partitioning and fill-in reducing, we adopt the widely used METIS library [16]. For NICSLU, we replace its built-in forward/backward substitution implementation with Intel MKL routines because the former is much slower. All experiments are conducted on a public cloud computing platform, where the computer nodes are connected by Ethernet. Each computer node has a 32-core Intel Xeon Platinum 8378A CPU and 256GB RAM. In all the experiments, the wall-clock runtime excluding loading matrices from hard disk is reported.

For the DDM and the proposed algorithm, each process is allocated two threads (i.e., two CPU cores).

The results on a single computer node are listed in Table I. Because the scale of thupg8t and thupg10t is too large, all three implementations fail to complete the simulation due to out of memory. From the results we see that the proposed algorithm achieves an average 2.06X speedup over NICSLU and an average 1.43X speedup over the DDM.

TABLE I
RESULTS OF TRANSIENT POWER GRID ANALYSIS ON A SINGLE COMPUTER NODE (TIME IN UNIT OF SECOND)

Case	N_{th}	NICSLU	DDM			Proposed				
		T_s	T	T_f	T	N_S	T_f	T	$\overline{N}_S$	Sp1	Sp2
ibmpg3t $N=1.0$E6 $nnz=4.5$E6	4	74.4	47.8	9.07	24.8	0	8.88	25.9	0	1.84	0.96
	8		40.7	5.60	18.5	2325	6.08	16.3	561	2.49	1.13
	16		32.2	5.43	23.1	6288	4.25	13.1	377	2.45	1.76
	32		34.8	5.47	27.3	10860	3.83	10.6	261	3.30	2.58
ibmpg4t $N=1.2$E6 $nnz=5.8$E6	4	94.5	56.2	12.6	34.4	0	12.3	34.5	0	1.63	1.00
	8		43.1	7.82	24.9	3334	9.13	25.0	723	1.72	0.99
	16		37.3	6.51	27.2	6410	5.55	16.4	335	2.28	1.66
	32		41.9	8.43	39.1	13112	4.79	14.6	353	2.86	2.67
ibmpg5t $N=1.6$E6 $nnz=6.4$E6	4	75.5	50.1	11.1	32.5	540	11.43	39.2	0	1.28	0.83
	8		40.1	6.87	23.4	2469	6.87	23.6	330	1.70	0.99
	16		36.0	5.78	23.7	4670	5.78	16.7	204	2.15	1.41
	32		43.8	5.25	26.9	9388	5.25	13.9	294	3.15	1.93
ibmpg6t $N=2.4$E6 $nnz=9.7$E6	4	103	68.9	12.8	40.0	0	12.8	39.2	0	1.76	1.02
	8		53.2	7.64	27.8	1864	7.64	30.4	266	1.75	0.91
	16		48.3	5.23	23.7	4178	5.23	22.9	296	2.11	1.04
	32		61.6	4.51	22.6	7309	4.51	20.6	167	2.99	1.10
thupg1t $N=5.3$E6 $nnz=2.3$E7	4	458	276	66	176	2953	92.5	193	1401	1.43	0.92
	8		221	87.1	185	8792	66.0	142	1404	1.55	1.30
	16		187	69.4	166	15064	44.6	95.4	661	1.96	1.74
	32		200	51.6	170	25981	36.5	80.5	697	2.49	2.12
thupg3t $N=1.2$E7 $nnz=5.4$E7	4	1243	761	213	487	4646	323	610	2270	1.25	0.80
	8		587	374	611	14110	316	493	2268	1.19	1.24
	16		539	218	453	23615	168	300	994	1.80	1.51
	32		599	188	496	41340	138	252	1034	2.38	1.97
thupg5t $N=2.0$E7 $nnz=8.8$E7	4	2193	1932	533	1013	6777	626	1061	3234	1.82	0.95
	8		1203	511	915	17513	425	695	2445	1.73	1.32
	16		1058	482	891	30558	327	539	1576	1.96	1.65
	32		1211	522	1068	51244	263	445	1210	2.72	2.40
Average		-	-	-	-	-	-	-	-	2.06	1.43

N_{th} denotes the total number of threads used. T_s, T, T_f denote the time of 1-thread serial simulation, parallel simulation and matrix factorization, respectively. N_S denotes the size of interface in DDM. $\overline{N}_S$ denotes the average size of interfaces at level 1 in the proposed algorithm. Sp1, Sp2 denote the speedup ratios of the proposed algorithm over NICSLU and DDM, respectively.

The results on multiple computer nodes (virtual machines) are listed in Table II. Because NICSLU is based on shared memory architecture, it is removed for this comparison. The experiments with 64 and 128 threads are carried out on 2 and 4 computer nodes, respectively. Each computer node is allocated 16 processes. The DDM and the proposed algorithm both fail to complete the simulation of thupg10t on 2 computer nodes with 64 threads due to out of memory. From the results we see that the proposed algorithm achieves a 2.84X speedup averagely over the DDM, which is larger than the 1.43X speedup on a single computer node and proves its better parallel scalability.

From Table I and II we see that the size of Schur complement is reduced with the nested dissection. The communication volume of forward/backward substitution and RHS updating per time point is consistent with the theoretical analysis in Section III-C, which indicates the low communication overhead of the proposed algorithm. And, the communication volume of matrix reordering and factorization is only a small fraction of the total communication volume.

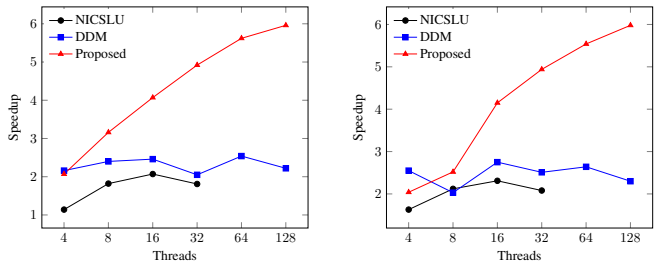
In all the experiments, the results of the proposed algorithm have no error (in double-precision floating-point arithmetic) with respect to the standard solutions obtained by NICSLU, which proves its accuracy. To further demonstrate the parallel

TABLE II
RESULTS OF TRANSIENT POWER GRID ANALYSIS ON MULTIPLE COMPUTER NODES (TIME IN UNIT OF SECOND, COMMUNICATION VOLUME IN UNIT OF GIGABYTE)

Case	N_{th}	DDM			Proposed					
		T_f	T	N_S	T_f	T	$\overline{N}_S$	V_f	V_{tr}	Sp
ibmpg3t	64	6.67	36.0	17508	3.96	10.9	162	0.50	0.06	3.29
	128	10.2	56.1	26646	4.16	11.5	126	0.93	0.07	4.88
ibmpg4t	64	8.15	45.5	19966	4.47	13.6	196	0.65	0.07	3.35
	128	13.1	64.8	32215	5.01	14.0	172	1.19	0.09	4.63
ibmpg5t	64	5.57	30.8	14865	5.57	13.9	159	0.76	0.09	2.22
	128	10.5	51.3	24295	5.84	14.4	138	1.11	0.12	3.57
ibmpg6t	64	5.73	30.4	14441	5.73	18.2	163	0.77	0.13	1.67
	128	9.98	48.4	22794	6.15	18.9	111	1.50	0.18	2.56
thupg1t	64	40.2	175	38348	31.8	75.0	324	3.14	0.31	2.33
	128	47.8	212	58838	33.2	78.3	315	5.55	0.38	2.71
thupg3t	64	142	472	60959	116	224	516	7.78	0.68	2.10
	128	197	541	91613	108	208	489	13.5	0.88	2.60
thupg5t	64	291	863	77062	224	391	763	12.9	1.13	2.21
	128	371	987	117742	209	368	594	22.3	1.47	2.68
thupg8t $N=4.1$E7 $nnz=1.8$E8	64	859	2714	106472	806	1175	1049	27.4	2.33	2.31
	128	801	2875	142819	750	1089	874	47.1	2.97	2.64
thupg10t $N=6.3$E7 $nnz=2.7$E8	128	1239	4375	181934	1058	1695	1034	73.0	4.57	2.58
Average		-	-	-	-	-	-	-	-	2.84

The notations of $N_{th}, T_f, T, N_S, \overline{N}_S$ and Sp are the same as Table I. V_f denotes the communication volume of matrix reordering and factorization and V_{tr} denotes the average communication volume of forward/backward substitution and RHS updating per time point.

scalability of the proposed algorithm, we plot the speedup ratios of the three implementations with different number of threads over 1-thread serial simulation on thupg3t and thupg5t in Fig. 5. From it we see that the speedup ratios of NICSLU and DDM are basically unchanged with the increase of the number of threads, and that of DDM even decreases with 128 threads. In contrast, the speedup ratio of the proposed algorithm increases with the number of threads and achieves up to about 6.0X with 128 threads. The reason is that the proposed algorithm effectively reduces the size of Schur complement and has a low communication volume.



(a) thupg3t (b) thupg5t

Fig. 5. The speedup ratios of three implementations with different number of threads over 1-thread serial simulation on thupg3t and thupg5t.

Finally, we do a rough cost analysis. In the cloud computing platform, the rental price of the virtual machine we use is 1.62$/h. For the transient simulation of thupg8t, using 4 computer nodes costs extra (1089×4-1175×2)×1.62/3600=0.90$ for virtual machine rent than using 2 computer nodes. On the other hand, this reduces 1175-1089=86s of the simulation time which means saving 86s time of the IC designer. This is beneficial as it corresponds to saving 0.96$ providing that the human cost of IC designer is about 40$/h. For larger cases, using the multiple nodes in cloud computing would be the required choice for accurate transient analysis due to its excessive memory cost.

V. Conclusions

In this work, we propose to leverage the public cloud computing for transient PG analysis while ensuring security. A multi-level distributed parallel LU factorization and forward/backward substitution approach based on nested dissection with low communication overhead is then proposed to guarantee accuracy and robustness. And, an efficient scheme is designed for RHS updating of the linear system to further exploit parallelism. These make the proposed algorithm suitable to be deployed on public cloud computing platforms without InfiniBand. Experiments on transient analysis of 9 practical PGs show that the proposed algorithm achieves an average 2.06X speedup over NICSLU on a single computer node and an average 2.84X speedup over DDM on multiple computer nodes. Moreover, the proposed algorithm exhibits good scalability, with up to 6.0X speedup over the serial simulation on the large-scale power grids.

VI. Acknowledgment

This work is partially supported by National Key R&D Program of China (2021ZD0114703), and partially supported by the National Natural Science Foundation of China (62204141).

References

[1] X. S. Li, "An overview of SuperLU: algorithms, implementation, and user interface," *ACM Trans. Mathmatical Software*, vol. 31, no. 3, pp. 302–325, 2005.

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

[3] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Trans. Mathmatical Software*, vol. 37, no. 3, pp. 1–17, 2010.

[4] X. Chen, Y. Wang, and H. Yang, *Parallel Sparse Direct Solver for Integrated Circuit Simulation*. Spinger, 2017.

[5] X. Chen, "Numerically-stable and highly-scalable parallel LU factorization for circuit simulation," in *Proc. ICCAD*, 2022, pp. 1–9.

[6] 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.

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

[8] C. Li, C. An, F. Yang, and X. Zeng, "ESPSim: An efficient scalable power grid simulator based on parallel algebraic multigrid," *ACM Trans. Design Automation of Electronic Systems*, vol. 28, no. 1, pp. 1–31, 2022.

[9] L. Li, Z. Liu, K. Liu, S. Shen, and W. Yu, "Parallel incomplete LU factorization based iterative solver for fixed-structure linear equations in circuit simulation," in *Proc. ASPDAC*, 2023, p. 339–345.

[10] 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, 2020.

[11] Z. Liu and W. Yu, "pGRASS-Solver: a parallel iterative solver for scalable power grid analysis based on graph spectral sparsification," in *Proc. ICCAD*, 2021.

[12] K. Sun, Q. Zhou, K. Mohanram, and D. C. Sorensen, "Parallel domain decomposition for simulation of large-scale power grids," in *Proc. ICCAD*, 2007, pp. 54–59.

[13] T. Yu, Z. Xiao, and M. D. F. Wong, "Efficient parallel power grid analysis via additive schwarz method," in *Proc. ICCAD*, 2012, pp. 399–406.

[14] S. R. Nassif, "Power grid analysis benchmarks," in *Proc. ASPDAC*, 2008, pp. 376–381.

[15] J. Yang, Z. Li, Y. Cai, and Q. Zhou, "PowerRush: efficient transient simulation for power grid analysis," in *Proc. ICCAD*, 2012, pp. 653–659.

[16] 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.

[17] B. Hendrickson and E. Rothberg, "Improving the run time and quality of nested dissection ordering," *SIAM Journal on Scientific Computing*, vol. 20, no. 2, pp. 468–489, 1998.

[18] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 5, pp. 862–874, 1988.

[19] J. Yang and Z. Li, "THU Power Grid Benchmarks," http://tiger.cs.tsinghua.edu.cn/PGBench/, 2023.

[20] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: goals, concept, and design of a next generation MPI implementation," in *Proc. the 11th European PVM/MPI Users' Group Meeting*, 2004, pp. 97–104.

[21] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[22] Intel Inc., "Accelerate Fast Math with Intel® oneAPI Math Kernel Library," https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html, 2023.