# Chapter 2
# Related Work

Parallel circuit simulation has been a popular research topic for several decades since the invention of SPICE. Researchers have proposed a large amount of parallelization techniques for SPICE-like circuit simulation [1]. In this chapter, we will comprehensively review state-of-the-art studies on parallel circuit simulation techniques. Before that, we would like to briefly introduce classifications of these parallel techniques. Based on different points of view, parallel circuit simulation techniques can also have different classifications. From the implementation platform point of view, parallel circuit simulation techniques can be classified into software techniques and hardware techniques. Hardware techniques include field-programmable gate array (FPGA)- and graphics processing unit (GPU)-based acceleration approaches. For software techniques, from the domain of parallel processing point of view, they can be further classified into direct parallel methods, parallel circuit-domain techniques, and parallel time-domain techniques. From the algorithm level of parallel processing point of view, there are intra-algorithm and inter-algorithm parallel techniques.

## 2.1 Direct Parallel Methods

According to the simulation flow shown in Fig. 1.2, the most straightforward way to parallelize SPICE-like circuit simulators is to parallelize every step in the SPICE simulation flow. Basically, the following major steps in the SPICE simulation flow can be parallelized: netlist parsing and simulation setup, matrix pre-analysis, device model evaluation, sparse direct solver, matrix/RHS load, and time node control. However, as explained in Sect. 1.2, some steps are quite sequential and difficult to parallelize. In addition, steps before entering SPICE iterations (i.e., netlist parsing, simulation setup, and matrix pre-analysis) are executed only once, so their performance is insensitive to the overall performance. According to the percentage of the runtime, one may only focus on the parallelization of device model evaluation and

© Springer International Publishing AG 2017

X. Chen et al., *Parallel Sparse Direct Solver for Integrated Circuit Simulation*,
DOI 10.1007/978-3-319-53429-9_2

the sparse direct solver, which are the two most time-consuming components in the SPICE flow. Such simulation techniques can be called direct parallel methods as they are straightforward to implement in existing SPICE-like simulation tools. This is also the conventional parallelization method adopted by many commercial products. As explained in Sect. 1.2, the parallel efficiency of device model evaluation can be close to 100% but the parallel efficiency of other steps, especially the sparse direct solver, cannot be as high as expected. This means that, the overall parallel efficiency is mainly limited by the poor scalability of those steps that cannot be efficiently parallelized. A detailed description of direct parallel methods is presented in an early publication [2]. It gives several methods to improve the parallel efficiency for the matrix/RHS load step using multiple locks or barriers.

In fact, for direct parallel methods, people pay more attention to the parallelization of the sparse direct solver, due to its high runtime percentage and high difficulty of parallelization. In what follows, we will review existing techniques for parallel direct and iterative matrix solutions.

### *2.1.1 Parallel Direct Matrix Solutions*

SPICE-like circuit simulators typically use sparse LU factorization to solve linear systems. Although there are many popular software packages that implement parallel sparse LU factorization algorithms, most of the efforts have been made for general-purpose sparse linear system solving in recent years, while very few studies are carried out specially for circuit simulation problems. The main difficulty in parallelization of sparse direct solvers for SPICE-like circuit simulation problems comes from the highly sparse and irregular nature of circuit matrices. Unstructured and irregular sparse operations must be processed with load balance. In addition, the parallelization overheads associated with a small number of floating-point operations (FLOP) for each task must be well controlled. On the other hand, the sparsity offers a new opportunity to parallelize sparse direct methods, as multiple rows or columns may be computed simultaneously. In direct methods, the numerical factorization step usually spends much more time than forward/backward substitutions, so people's interest mainly focuses on parallelization of the numerical factorization step.

State-of-the-art popular sparse direct solvers include the SuperLU series (SuperLU, SuperLU_MT and SuperLU_Dist) [3–7], the Unsymmetric Multifrontal Package (UMFPACK) [8], KLU [9], the Parallel Sparse Direct Solver (PARDISO) [10–12], Multifrontal Massively Parallel Sparse Direct Solver (MUMPS) [13–15], the Watson Sparse Matrix Package (WSMP) [16], etc. Among these solvers, only KLU is specially design for circuit simulation applications. However, KLU is purely sequential. According to the fundamental algorithms adopted by these solvers, they can be classified into two main categories: dense submatrix-based methods and non-dense-submatrix-based methods. The basic idea of dense submatrix-based solvers is to collect and reorganize arithmetic operations of sparse nonzero elements

into regular dense matrix operations, such that the basic linear algebra subprogram (BLAS) [17] and/or the linear algebra package (LAPACK) [18] can be invoked to deal with dense submatrices. These solvers can be further classified into two categories: supernodal methods and multifrontal methods.

#### 2.1.1.1 Supernodal Methods

A supernode is generally defined as a set of successive rows or columns of **U** or **L** with triangular diagonal block full and the same structure in the rows or columns below or on the right side of the diagonal block [3, 6]. Row- and column-order supernodes with the same row and column indexes can also be combined to form a single supernode, as illustrated in Fig. 2.1. Supernodes can be treated as dense submatrices for storage and computation, such that both the computational efficiency and cache performance can be improved. To efficiently process dense matrix computations, vendor-optimized BLAS and/or LAPACK is usually required.

To explore parallelism from the sparsity, a task graph which is a direct acyclic graph (DAG) is usually used to represent the data dependence in sparse LU factorization. In SuperLU_MT [5, 7], the task graph is named elimination tree (ET) [19]. SuperLU_MT is based on the sparse left-looking algorithm developed by Gilbert and Peierls [20], which is named G-P algorithm, and utilizes column-order supernodes. Based on the dependence represented by the ET, SuperLU_MT uses a pipelined supernodal algorithm to schedule the parallel LU factorization. Due to the fact that partial pivoting can interchange the order of rows so that an exact column-level dependence graph cannot be determined before factorization, the concept of ET contains all potential column-level dependence regardless of the actual pivot choices, which means that it is a toplimit of the column-level dependence. An example of the ET is shown in Fig. 2.2. The use of the ET enables a static scheduling graph that can be determined before factorization, but the overhead is that the ET overdetermines the column-level dependence and contains much redundant dependence.

PARDISO [10–12] also utilizes supernodes to realize a parallel LU factorization algorithm, but its strategy is quite different from SuperLU. The authors of PARDISO

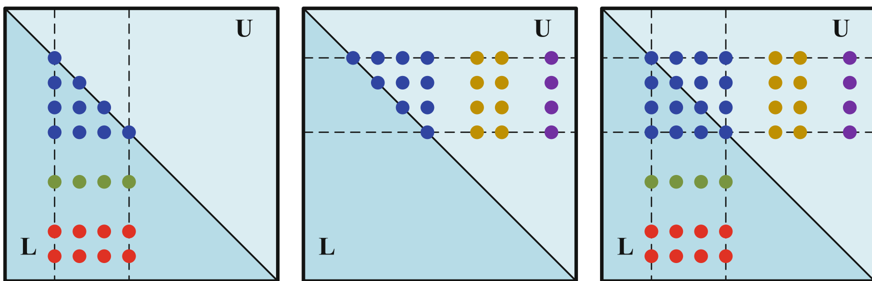
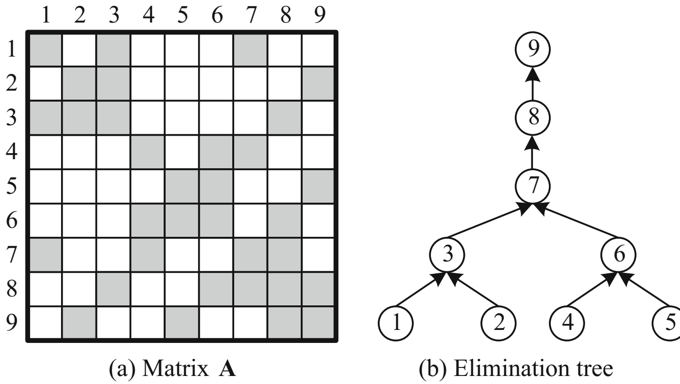



**Fig. 2.1** Examples of supernode



(a) Matrix **A** (b) Elimination tree

**Fig. 2.2** Example of elimination tree

have developed a parallel left-right-looking algorithm [11] which is associated with a complete block supernode diagonal pivoting method, where rows and columns of a diagonal supernode can be interchanged without affecting the task dependence graph. Such a strategy enables a complete static task dependence graph that represents the dependence exactly without any redundance, but the overhead is that it can sometimes lead to unstable solutions so an iterative refinement is required after forward/backward substitutions for PARDISO. PARDISO further explores the parallel scalability by a two-level dynamic scheduling [12]. According to the comparison of different sparse solvers presented in [21], PARDISO is one of highest performance sparse solver for general sparse matrices.

#### 2.1.1.2 Multifrontal Methods

The main purpose of the multifrontal [22] technique is somewhat similar to that of the supernodal technique, but the basic theory and implementation are quite different. The multifrontal technique factorizes a sparse matrix with a sequence of dense frontal matrices, each of which corresponds to one or more steps of the LU factorization. We use the example shown in Fig. 2.3 to demonstrate the basic idea of the multifrontal method. The first pivot, say element (1, 1), is selected, and then the first frontal matrix is constructed by collecting all the nonzero elements that will contribute to the elimination of the first pivot row and column by the right-looking algorithm, as shown in Fig. 2.3b. The frontal matrix is then factorized by a dense right-looking-like pivoting operation, resulting in the factorized frontal matrix shown in Fig. 2.3c. As can be seen, the computations of the frontal matrix can be done by dense kernels such as BLAS so the performance can be enhanced. After eliminating the first pivot, the second pivot, say element (3, 2), is selected. A new frontal matrix is constructed by collecting all the contributing elements that are from the original matrix and the previous frontal matrix, as shown in Fig. 2.3d. It is then also factorized and the

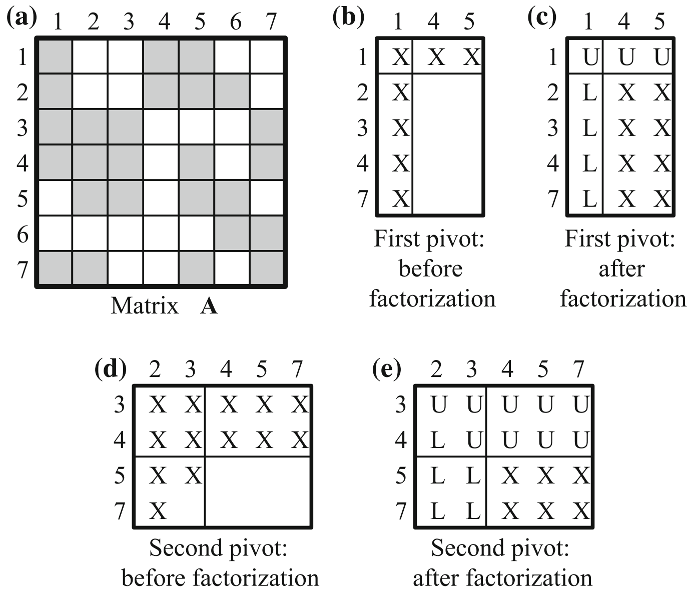



**Fig. 2.3** Illustration of the multifrontal method [23]

resulting frontal matrix is shown in Fig. 2.3e. The same procedure will be continued until the LU factors are complete. The multifrontal technique can also be combined with the supernodal technique to further improve the performance by simultaneously processing multiple frontal matrices with the identical pattern.

There are several levels of parallelism in the multifrontal algorithm [14]. First, one can also use the ET to schedule the computational tasks, such that independent frontal matrices can be processed concurrently. This is a task-level parallelism. Second, if a frontal matrix is large, it can be factorized by a parallel BLAS so this is a data-level parallelism. Third, the factorization of the dense node at the root position of the ET can be factorization by a parallel LAPACK.

Many software packages are based on the multifrontal technique. UMFPACK [8] is an implementation of the multifrontal method to solve sparse linear systems. Although the solver itself is purely sequential, its parallelism can be simply explored by invoking parallel BLAS. MUMPS [13–15] is a multifrontal-based distributed sparse direct solver. WSMP [16] is a collection of various algorithms to solve sparse linear systems that can be executed both in sequential and parallel. For sparse unsymmetric matrices, it adopts the multifrontal algorithm.

#### 2.1.1.3 Non-Submatrix-Based Methods

Unlike the supernodal or multifrontal algorithm, this category of methods do not form any dense submatrices during sparse LU factorization. A representative solver is KLU [9], which is an improved implementation of the G-P sparse left-looking algorithm [20]. As circuit matrices are generally extremely sparse, it is difficult to form big dense submatrices during sparse LU factorization, and, thus, this type of solvers is considered to be more suitable for circuit simulation applications, as supported by the test results of KLU.

A multi-granularity parallel LU factorization algorithm has been proposed in [24]. However, it can only be applied to symmetric matrices. Actually, in nonlinear circuit simulation, the matrix is usually unsymmetric, so symmetric LU factorization is useless. In addition, for symmetric matrices, Cholesky factorization [25] is about twice as efficient than LU factorization.

ShyLU [26], developed by the Sandia National Laboratory, is a two-level hybrid sparse linear solver. The first level hybrid comes from the combined direct and iterative algorithms. The matrix is partitioned into four blocks, i.e.,

$$\mathbf{A} = \begin{bmatrix} \mathbf{D} & \mathbf{C} \\ \mathbf{R} & \mathbf{G} \end{bmatrix} \tag{2.1}$$

where $\mathbf{D}$ and $\mathbf{G}$ are square and $\mathbf{D}$ is a non-singular block-diagonal matrix. $\mathbf{D}$ can be easily factorized by a sparse LU factorization, and then an approximate Schur complement [27] is calculated, i.e.,

**Algorithm 1** Algorithm of ShyLU [26].

1: Factorize $\mathbf{D}$ by sparse LU factorization

2: Compute approximate Schur complement:

$$\bar{\mathbf{S}} \approx \mathbf{G} - \mathbf{R}\mathbf{D}^{-1}\mathbf{C}$$

3: Solve

$$\mathbf{D}\mathbf{z} = \mathbf{b}_1$$

4: Solve

$$\mathbf{S}\mathbf{x}_2 = \mathbf{b}_2 - \mathbf{R}\mathbf{z}$$

using iterative methods where $\bar{\mathbf{S}}$ is used as the pre-conditioner. $\mathbf{S}$ is the exact Schur complement but it does not need to be explicit formed

5: Solve

$$\mathbf{D}\mathbf{x}_1 = \mathbf{b}_1 - \mathbf{C}\mathbf{x}_2$$

$$\bar{\mathbf{S}} \approx \mathbf{G} - \mathbf{R}\mathbf{D}^{-1}\mathbf{C}. \tag{2.2}$$

The approximate Schur complement serves as a pre-conditioner to solve the linear equations corresponding to the right-bottom block using iterative methods. For a linear system

$$\begin{bmatrix} \mathbf{D} & \mathbf{C} \\ \mathbf{R} & \mathbf{G} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix}, \tag{2.3}$$

ShyLU solves it using the algorithm shown in Algorithm 1. The second level hybrid comes from the combined multi-machine and multi-core parallelism. ShyLU has also been tested in SPICE-like circuit simulation. According to very limited results [28], the performance of ShyLU in circuit simulation, especially the speedup over KLU, is not so remarkable (the speedup over KLU is only $20\times$ using 256 cores for a particular circuit).

Till now, very few sparse linear solvers are specially designed for circuit simulation applications, and very few public results of sparse linear solvers are reported for circuit matrices. We believe that a comprehensive comparison and investigation between various algorithms of sparse linear solvers on circuit matrices from different applications can provide lots of new insights and guidelines to the development of sparse linear solvers for circuit simulation.

### *2.1.2 Parallel Iterative Matrix Solutions*

Compared with direct methods, iterative methods can significantly reduce the memory requirement as they are executed almost in-place. Iterative methods are also quite easy to parallelize as the core operation is just sparse matrix-vector multiplication (SpMV). There are a great number of parallel SpMV implementations on modern multi-core CPUs, many-core GPUs, or reconfigurable FPGAs [29–34]. However, in fact, there are very few researches that have investigated iterative methods for solving linear systems in SPICE-like circuit simulation applications. Commercial general-purpose circuit simulators rarely use iterative methods, mainly due to the convergence and robustness issues of iterative methods. To improve the convergence, iterative methods require good pre-conditioners, which should have the following two properties. First, the pre-conditioner should approximate the matrix very well to ensure good convergence. Second, the inverse of the pre-conditioner should be cheap to compute to reduce the runtime of the linear solver. In most cases, we do not need to explicitly calculate the inverse but the equivalent implicit computations should also be cheap. For parallel iterative methods, many research efforts have been carried out on how to build robust pre-conditioners, as iterative methods themselves are straightforward to parallelize.

An example of a pre-conditioned linear system can be simply expressed as follows:

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}, \tag{2.4}$$

where $\mathbf{M}$ is the pre-conditioner. $\mathbf{M}$ is selected such that solving the linear system of Eq. (2.4) by iterative methods can converge much faster than solving the original linear system $\mathbf{Ax} = \mathbf{b}$. If $\mathbf{M}$ is exactly $\mathbf{A}$, then the left side of Eq. (2.4) is exactly the identity matrix so it can be trivially solved. However, if we have obtained the exact $\mathbf{A}^{-1}$, it is equivalent to that we have already solved the original linear system. In other words, it is unnecessary to compute the exact inverse. On the contrary, the pre-conditioner should be selected such that it can approximate the matrix as exactly as possible with a very cheap method.

In mathematics, pre-conditioner techniques can be classified into two main categories: incomplete factorization per-conditioner and approximate inverse pre-conditioner [35]. Incomplete factorization tries to find an approximate factorization of the matrix, i.e.,

$$\mathbf{A} \approx \tilde{\mathbf{L}}\tilde{\mathbf{U}}. \tag{2.5}$$

Typically, the approximate factors are obtained from LU factorization by dropping small values under a given threshold. Tradeoffs can be explored between the number of fill-ins in the approximate factors, i.e., the approximate accuracy, and the computational cost of the pre-conditioner. The approximate inverse pre-conditioner tries to calculate a sparse matrix $\mathbf{M}$ which minimizes the Frobenius norm of the following residual:

$$F(\mathbf{M}) = ||\mathbf{I} - \mathbf{AM}||_F^2. \tag{2.6}$$

Researchers have proposed some iterative algorithms that can efficiently calculate the sparse approximate inverse matrix $\mathbf{M}$ [35].

Based on the theory of these pre-conditioners, a few parallel pre-conditioners have been developed for circuit simulation problems [36–38]. A common feature of these early work is that they treat the pre-conditioner and the iterative solve as a black-box and do not utilize any information from circuit simulation.

In SPICE-like circuit simulation, there is another opportunity to apply pre-conditioners for iterative solvers to solve linear systems. Due to the quadratic convergence of the Newton–Raphson method, the matrix values change slow during SPICE iterations, especially when the Newton–Raphson iterations are converging. This property provides us an opportunity to utilize the LU factors in a certain iteration to serve as a pre-conditioner for subsequent iterations which are solved by sequential or parallel generalized minimal residual (GMRES) methods [39–41]. Compared with the previous approaches that apply additional pre-conditioners, the computational cost of the pre-conditioner can be almost ignored in these methods, as computation of the pre-conditioner, i.e., the complete LU factorization, is an inherent step in circuit simulation. Another advantage is that the pre-conditioner can be used in multiple iterations if the matrix values change very slow. However, due to the sensitivity of iterative methods on matrix values, it is difficult to judge when the pre-conditioner is invalid. To overcome this problem, the nonlinear device models are piecewise linearized, and once nonlinear devices change their operating regions, the pre-conditioner is required to update [39–41].

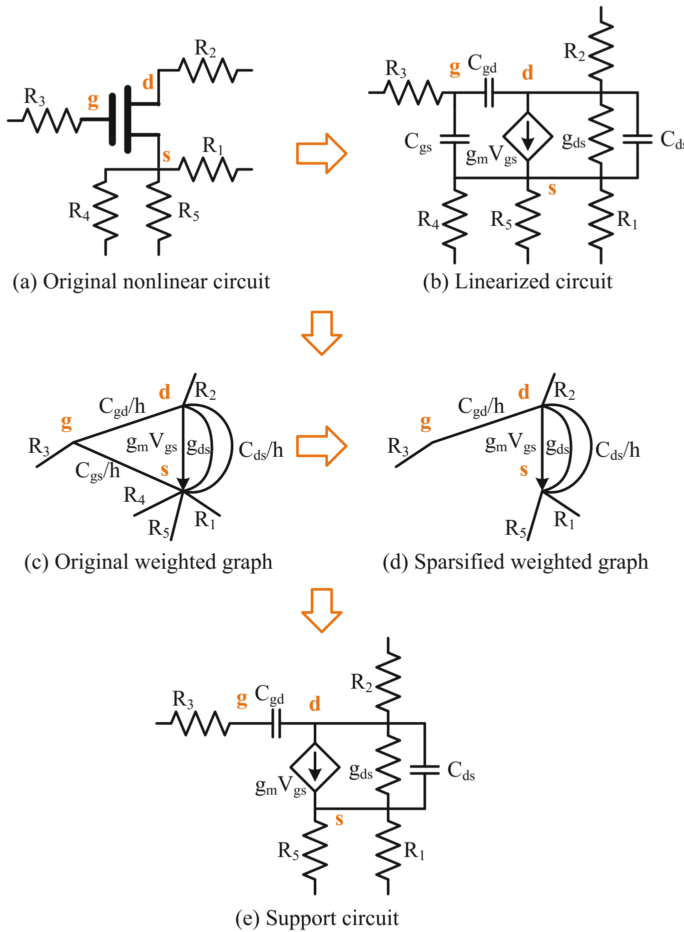



**Fig. 2.4** Example of support circuit pre-conditioner [42–44]

The above pre-conditioners are purely based on the matrix information, completely ignoring the circuit-level information. In other words, they are pure matrix-based methods. Another type of pre-conditioners utilizes circuit-level information named supper circuit pre-conditioner [42–44], which is based on the support graph and graph sparsification theories [45]. The basic idea is to extract a highly sparsified circuit network which is called a support circuit that is very close to the original circuit, so that matrix factorization for the support circuit can be quickly done almost in linear time, and can be served as the pre-conditioner for GMRES. Figure 2.4 shows an example of the creation of the support circuit pre-conditioner.

The Sandia National Laboratory has proposed another type of pre-conditioner for SPICE-like circuit simulation [46]. It first partitions the circuit into several blocks and then uses the block Jacobi pre-conditioner for the GMRES solver. This approach

fails on some circuits so its applicability in real SPICE-like circuit simulation needs further investigation.

A common problem with pre-conditioned iterative methods in SPICE-like circuit simulation is the universality. Although existing researches have shown that the proposed approaches can work well for the circuits they have tested, unlike direct methods, it cannot guarantee that these approaches can also work well for any circuit. All of the existing iterative methods in circuit simulation are likely ad hoc approaches, and, hence, more universality should be explored.

## 2.2 Domain Decomposition

The concept of domain decomposition has different meanings under various contexts. Generally speaking, domain decomposition can be described as a method that solves a large problem by partitioning the problem into multiple small subproblems and then solving these subproblems separately. From the circuit point of view, to realize parallel simulation, a natural idea is to partition the circuit into multiple subcircuits such that each subcircuit can be solved independently, if the boundary condition is properly formulated at either circuit level or matrix level. Actually, domain decomposition is widely used in modern parallel circuit simulation tools, especially in fast SPICE simulation techniques. There are basically several types of methods in domain decomposition-based parallel simulation techniques: parallel bordered-block-diagonal (BBD)-form matrix solutions, parallel multilevel Newton methods, parallel Schwarz methods, and parallel waveform relaxation methods.

### *2.2.1 Parallel BBD-Form Matrix Solutions*

This type of methods is more like a matrix-level technique rather than a domain decomposition technique. However, building a BBD-form matrix requires to partition the circuit, and the performance of solving the BBD-form matrix strongly depends on the quality of the partition, so we put this type of methods in domain decomposition instead of direct parallel methods.

Figure 2.5 illustrates how to create the BBD form by circuit partitioning. The circuit is partitioned into $K$ non-overlapped subdomains, in which one subdomain contains all the interface nodes and the other subdomains are subcircuits. After such a partitioning, the matrix created by MNA naturally have a BBD form, where there are $K-1$ diagonal block matrices $\mathbf{D}_1, \ldots, \mathbf{D}_{K-1}$, $K-1$ bottom-border block matrices $\mathbf{R}_1, \ldots, \mathbf{R}_{K-1}$, $K-1$ right-border block matrices $\mathbf{C}_1, \ldots, \mathbf{C}_{K-1}$, and a right-bottom block matrix $\mathbf{G}$. The diagonal blocks correspond to the internal equations of all the subcircuits. The border blocks correspond to all the connections between subcircuits and interface nodes. The right-bottom block corresponds to the internal equations of interface nodes. LU factorization of a BBD-form matrix is based on the Schur

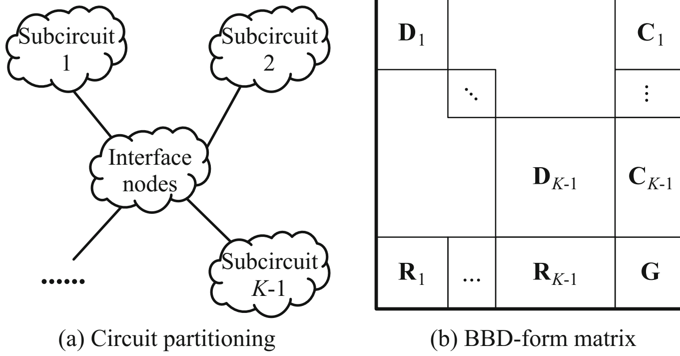



**Fig. 2.5** Illustration of how to create the BBD form by circuit partitioning

---

**Algorithm 2** LU factorization of a BBD-form matrix.

---

1: Factorize $K-1$ diagonal blocks

$$\mathbf{D}_1 = \mathbf{L}_1\mathbf{U}_1, \ldots, \mathbf{D}_{K-1} = \mathbf{L}_{K-1}\mathbf{U}_{K-1}$$

2: Update $K-1$ bottom-border blocks

$$\mathbf{R}_1 = \mathbf{R}_1\mathbf{U}_1^{-1}, \cdots, \mathbf{R}_{K-1} = \mathbf{R}_{K-1}\mathbf{U}_{K-1}^{-1}$$

3: Update $K-1$ right-border blocks

$$\mathbf{C}_1 = \mathbf{L}_1^{-1}\mathbf{C}_1, \cdots, \mathbf{C}_{K-1} = \mathbf{L}_{K-1}^{-1}\mathbf{C}_{K-1}$$

4: Accumulate updates to the right-bottom block

$$\mathbf{G} = \mathbf{G} - \sum_{k=1}^{K-1} \mathbf{R}_k\mathbf{C}_k$$

5: Factorize the right-bottom block

$$\mathbf{G} = \mathbf{L}_K\mathbf{U}_K$$

---

complement theory [27]. The factorization process can be described by Algorithm 2. There are several opportunities to parallelize the factorization of a BBD-form matrix. First, factorizations of diagonal blocks are completely independent so they can be trivially parallelized. In addition, factorization of each diagonal block can also be parallelized. The same conclusion also holds for the updates to all the border blocks.

Second, accumulation to the right-bottom block can be partially parallelized. Third, factorization of the right-bottom block can also be parallelized.

Constructing the BBD-form matrix can be achieved by either matrix-level methods or circuit-level methods. Existing approaches often create the BBD-form matrix by partitioning the circuit [47–50], although pure matrix-level methods also exist [51–53]. From the circuit design point of view, large circuits are usually designed hierarchically and structured. This will greatly help reduce the difficulty of partitioning the circuit. In fact, matrix-level methods create the BBD-form matrix by creating a network based on the symbolic pattern of the matrix, and then partitioning the network into subdomains.

A few practical issues should be considered when implementing parallel BBD-form matrix solutions. First, the right-bottom block can be a severe bottleneck in the parallel solver, as it can be quite dense and dominates the overall computational time. In fact, not only factorizing the right-bottom block can be expensive, accumulating updates to that block can also be time-consuming. The reason is that the accumulation cannot be efficiently parallelized as multiple different submatrices may update the same position, which requires a lock for each nonzero element in the right-bottom block. As the size of the right-bottom block depends on the number of interface nodes, a high-quality partitioning is required. Second, load balance is big problem. As can be seen, the size of each diagonal block depends on the size of each subcircuit after circuit partitioning. In practice, the size of different circuit modules can vary much so it is difficult to obtain equal-sized subcircuits. If we force to get a partition with equal-sized subcircuits, the number of cut-off links will be significantly large, i.e., the right-bottom block will be large. One solution is to partition the circuit into a number of small subcircuits such that load balance can be achieved by dynamic scheduling. However, this method implies that the circuit is quite large and has many submodules, which is not always true in practice.

### *2.2.2 Parallel Multilevel Newton Methods*

The above BBD-form matrix solutions are still matrix-level approaches but not real circuit-level approaches. The idea can also be extended for solving nonlinear equations by the concept of the multilevel Newton technique [54–57]. Multilevel Newton methods are actually algorithm-level methods but they are operated at the circuit level.

The basic idea of multilevel Newton methods can be described as follows. Each subdomain is first solved separately using the Newton–Raphson method with a given boundary condition, and then the top-level nonlinear equation is solved by integrating the updated solutions from all the subdomains. The two levels of Newton–Raphson iterations are repeated until all the boundary conditions are converged. Multilevel Newton methods can be formulated as follows. After the circuit is partitioned into $K$ subdomains in which one subdomain contains the interface nodes, we have $K$ equations to describe the whole system

$$f_i(\mathbf{x}_i, \mathbf{u}) = 0, i = 1, 2, \ldots, K - 1 \tag{2.7}$$

$$g(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{K-1}, \mathbf{u}) = 0, \tag{2.8}$$

where $\mathbf{x}_i$ is the unknown vector of subdomain $i$ ($i = 1, 2, \ldots, K - 1$), and $\mathbf{u}$ is the unknown vector corresponding to the interface nodes. Equation (2.7) is the local nonlinear equation of subdomain $i$, and Eq. (2.8) corresponds to the top-level nonlinear equation. Equations (2.7) and (2.8) are solved hierarchically by multilevel Newton methods. First, an inner Newton–Raphson iterations loop is performed to solve each local equation Eq. (2.7) under a fixed boundary condition $\mathbf{u}$ until convergence. After that, an outer Newton–Raphson iterations loop is performed to solve the top-level global equation Eq. (2.8) based on the solutions received from all the local equations. The two levels of Newton–Raphson iterations loop will be repeated until all the solutions, i.e., $\mathbf{x}_i$ and $\mathbf{u}$, are converged.

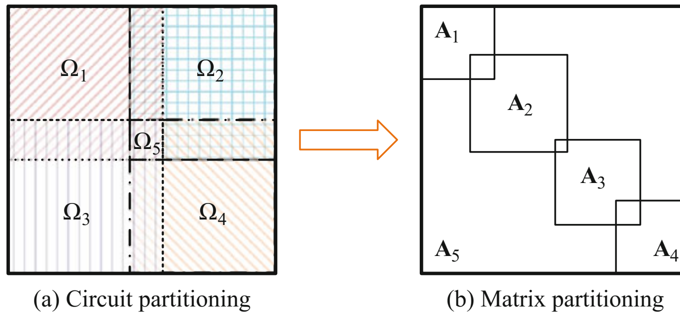
In addition to that multilevel Newton methods can be easily parallelize, there is another unique advantage for multilevel Newton methods. In general cases, the quadratic convergence property of the Newton–Raphson method is still retained in multilevel Newton methods. In the mean time, the overall computational cost can be significantly reduced, as the Newton–Raphson method for each subcircuit can be converged quickly due to the small size of each subcircuit. Consequently, the performance improvement of parallel multilevel Newton methods comes from two aspects: one is the improved fundamental algorithm and the other is the parallelism.

### *2.2.3 Parallel Schwarz Methods*

The above parallel BBD-form matrix solutions and parallel multilevel Newton methods are both master-slave approaches, in which the master may be a severe bottleneck. To resolve this bottleneck, Schwarz methods can be adopted [58]. Different from the above nonoverlapping partition methods, in Schwarz methods, the circuit is partitioned into multiple overlapped subdomains.

A parallel simulation approach using the Schwarz alternating procedure has been proposed in [59, 60]. A circuit can be partitioned into $K - 1$ nonlinear subdomains $\Omega_1, \Omega_2, \ldots, \Omega_{K-1}$ and a linear subdomain $\Omega_K$. This is equivalent to partition the matrix $\mathbf{A}$ into $K - 1$ overlapped submatrices $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_{K-1}$ corresponding to all the nonlinear subdomains and a background matrix $\mathbf{A}_K$ corresponding to the overlaps of subdomains $\Omega_1, \Omega_2, \ldots, \Omega_{K-1}$ and the linear subdomain $\Omega_K$, as illustrated in Fig. 2.6. After partitioning, linear systems during SPICE simulation is solved by the Schwarz alternating procedure, as shown in Algorithm 3, in which all the subdomains are solved in parallel.

Compared with parallel BBD-form matrix solutions and parallel multilevel Newton methods, the main advantage of parallel Scharwz methods is that, parallel Scharwz methods do not belong to the master–slave parallelization framework but they only involve point-to-point communications, potentially resulting in better




(a) Circuit partitioning (b) Matrix partitioning

**Fig. 2.6** Illustration of overlapped circuit partitioning and its corresponding matrix partitioning

**Algorithm 3** Schwarz alternating procedure [59, 60].

1: Choose initial guess of the solution $\mathbf{x}$
2: Calculate the residual

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$$

3: **repeat**
4: **for** $k = 1$ to $K$ in parallel **do**
5: Solve

$$\mathbf{A}_k \boldsymbol{\delta}_k = \mathbf{r}_k$$

6: Update solution

$$\mathbf{x}_k = \mathbf{x}_k + \boldsymbol{\delta}_k$$

7: Update residuals on boundary
8: **end for**
9: **until** all boundary conditions are converged

parallel scalability, as the bottleneck of the master is avoided. Since Schwarz methods belong to the category of iterative methods, they suffer from the convergence problem. A general conclusion is that the convergence speed can be significantly improved by increasing the overlapping areas. However, increasing overlaps leads to higher computational cost.

### *2.2.4 Parallel Relaxation Methods*

Relaxation techniques have been developed to solve linear or nonlinear equations from a variety of areas. In the circuit simulation area, there are a large amount of researches about parallel simulation using relaxation methods. Relaxation can be applied to three types of equations: linear equations, nonlinear equations, and

differential equations. Remember that we have presented several fundamental equations of SPICE-like circuit simulation in Sect. 1.1.1. Relaxation for linear equations is applied to Eq. (1.5). Typical algorithms include the Gauss-Jacobi method and the Gauss-Seidel method. They are iterative methods so they both suffer from the convergence problem. The linear relaxation methods can also be extended to nonlinear equations, e.g., Eq. (1.3). For both linear and nonlinear relaxation methods, the convergence speed is linear. In circuit simulation, a large number of the efforts are focused on relaxation for differential equations. This leads to a type of methods called waveform relaxation [61–69], which solves the circuit differential equation (i.e., Eq. (1.1)) in a given time interval by relaxation techniques.

We briefly explain waveform relaxation in a mathematical form. Equation (1.1) can be rewritten into a different form

$$\frac{\mathrm{d}\boldsymbol{q}(\mathbf{x})}{\mathrm{d}\mathbf{x}^T} \cdot \frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} + \boldsymbol{f}(\mathbf{x}(t)) - \mathbf{u}(t) = 0. \tag{2.9}$$

Let $\mathbf{C}(\mathbf{x}(t))$ be $\frac{\mathrm{d}\boldsymbol{q}(\mathbf{x})}{\mathrm{d}\mathbf{x}^T}$ and $\mathbf{b}(\mathbf{u}(t), \mathbf{x}(t))$ be $\boldsymbol{f}(\mathbf{x}(t)) - \mathbf{u}(t)$, then Eq. (2.9) is further rewritten into the following form

$$\mathbf{C}(\mathbf{x}(t)) \cdot \frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} + \mathbf{b}(\mathbf{u}(t), \mathbf{x}(t)) = 0. \tag{2.10}$$

If we use the Gauss-Seidel method to solve Eq. (2.10), it results in the following linear system

$$\begin{aligned}
&\sum_{j=1}^{i} C_{ij}(x_1^{(k+1)}, \ldots, x_i^{(k+1)}, x_{i+1}^{(k)}, \ldots, x_N^{(k)}) \frac{\mathrm{d}x_j^{(k+1)}}{\mathrm{d}t} \\
&+ \sum_{j=i+1}^{N} C_{ij}(x_1^{(k+1)}, \ldots, x_i^{(k+1)}, x_{i+1}^{(k)}, \ldots, x_N^{(k)}) \frac{\mathrm{d}x_j^{(k)}}{\mathrm{d}t} \\
&+ b_i(x_1^{(k+1)}, \ldots, x_i^{(k+1)}, x_{i+1}^{(k)}, \ldots, x_N^{(k)}) = 0, i = 1, 2, \ldots, N
\end{aligned} \tag{2.11}$$

where the superscript is the iteration count. Waveform relaxation solves the circuit DAE Eq. (1.1) in a given time interval by iterating Eq. (2.11) until the solution is converged.

To enable parallel waveform relaxation, one also needs to partition the circuit into subcircuits, while the interactions between subcircuits can be approximated by proper devices, e.g., artificial sources. An DAE is built for each subcircuit and then solved by waveform relaxation based on Eq. (2.11). When solving a subcircuit, interactions from other subcircuits are considered and the latest solutions of interacted subcircuits are always used. As can be seen, parallel waveform relaxation combines both domain decomposition and time-domain parallelism.

Although waveform relaxation has been widely studied since the 1980s, they are actually not widely used in practical circuit simulators today. The reasons mainly include the convergence conditions and limitations of waveform relaxation. As waveform relaxation is an iterative method, convergence is always a problem. A necessary

condition for Eq. (2.10) to have a unique solution, requires that the matrix $\mathbf{C}(\mathbf{x}(t))^{-1}$ exists. This also implies that there must be a grounded capacitor at each node. Such a requirement cannot be always satisfied for actual circuits, especially for pre-layout circuits. In addition, waveform relaxation also requires that one node of each independent voltage source or inductor must be the ground, which also restricts the applicability of waveform relaxation.

## 2.3 Parallel Time-Domain Simulation

Except the relaxation methods, most of the above-mentioned methods have a common point that the parallelism is explored at each time node. If we put our focus to the whole time axis in transient simulation, parallelism can also be explored in the time domain by many other techniques. Namely, different time nodes in the time domain may be computed concurrently by either parallel integration algorithms or multiple algorithms calculating different time nodes. As mentioned in Sect. 1.1.3, the DAE associated with transient simulation is usually solved by numerical integration algorithms. Numerical integration algorithms are typically completely sequential at the time node level, as a node can be computed only when one or more previous nodes are finished. To explore parallelism in the time domain, one needs to carefully resolve this problem.

### *2.3.1 Parallel Numerical Integration Algorithms*

To explore parallelism along the time axis in SPICE-like transient simulation, WavePipe has been proposed [70]. WavePipe enables the simultaneous computation of multiple time nodes by two novel techniques: backward pipelining and forward pipelining.

#### 2.3.1.1 Backward Pipelining

An illustration of the backward pipelining is shown in Fig. 2.7. Consider a two-step numerical integration method. Using the solutions at time nodes $t_1$ and $t_2$ as the initial conditions, a thread can calculate the solution at $t_3$. To enable backward pipelining, at the same time, a second thread can calculate the solution at $t_3'$ which is smaller than $t_3$, using the solutions at $t_1$ and $t_2$ as the initial conditions as well. One may argue that the solution at $t_3'$ is useless because $t_3$ is always beyond $t_3'$ due to the use of the latest solutions, which means that $t_3'$ does not contribute to a faster calculation. However, the calculation of $t_3'$ is actually useful for parallel simulation. Recall that the time step of numerical integration methods is determined by the LTE of the previous integration step. Due to the existence of $t_3'$, the first thread, which will calculate the solution at

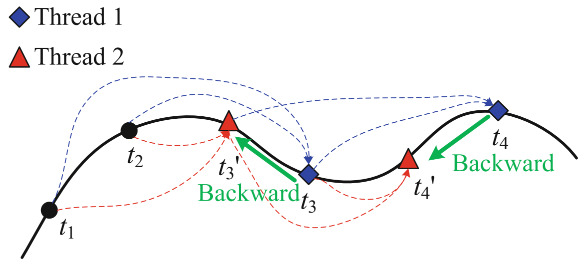



**Fig. 2.7** Backward pipelining of WavePipe [70]

a new time node using the solutions at $t_3$ and $t_3'$ as the initial conditions, can move forward by a larger time step to $t_4$, compared with a sequential integration method that uses the solutions at $t_3$ and $t_2$ as the initial conditions, due to the reduced LTE. At the same time, the second thread calculates the solution at $t_4'$ which is smaller than $t_4$. The calculations of $t_3'$ and $t_4'$ are called backward steps. As can be seen, backward pipelining results in larger time steps so accelerates transient simulation along the time axis. The basic principle behind backward pipelining is that it provides better initial conditions so that the integration time step can be larger.

#### 2.3.1.2 Forward Pipelining

Forward pipelining operates in a different way than backward pipelining. As illustrated in Fig. 2.8, a thread is calculating the solution at $t_3$ using the solutions at $t_1$ and $t_2$ as the initial conditions, and a second thread is attempting to calculate the solution at $t_4$ which is beyond $t_3$. The problem is that, if the second thread also uses the solutions at $t_2$ and $t_1$ as the initial conditions, the calculated solution at $t_4$ is unstable if the maximum step size is already exhausted at $t_3$. In the forward pipelining approach, the second thread uses the solutions at $t_3$ and $t_2$ as the initial conditions to calculate the solutions at $t_4$. Obviously, $t_3$ is still under calculation so its final solution is not available at this time. Recall that the Newton–Raphson method converges quadratically, and in SPICE-like transient simulation, it only requires a few Newton–Raphson iterations to achieve convergence at each time node. When $t_3$ is under calculation and its intermediate solution does not satisfy the LTE tolerance, the solution should be close to the final solution. Hence, the second thread can use the intermediate solution of $t_3$ as the initial condition to calculate the solution at

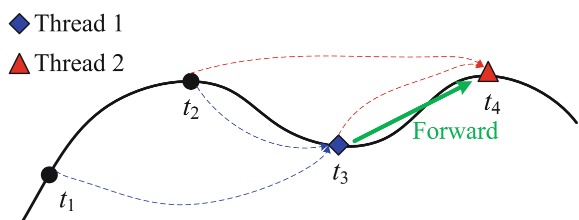



**Fig. 2.8** Forward pipelining of WavePipe [70]

$t_4$. The penalty is the increased number of Newton–Raphson iterations at $t_4$, due to the inaccurate initial conditions. The calculation of $t_4$ is called a forward step. The authors of WavePipe have proposed how to predict the time step and maintain the accuracy and stability. In addition, backward pipelining and forward pipelining can be combined together by a carefully designed thread scheduling policy.

There is no doubt that WavePipe has provided new insights to development of parallel time-domain simulation techniques, and the method can also be applied to other problems which need to solve differential equations. However, it can be expected that WavePipe requires find-grained inter-thread communications so the scalability will become poor when the number of threads increases.

### 2.3.2 *Parallel Multi-Algorithm Simulation*

A completely different parallel time-domain simulation technique named multi-algorithm simulation has been proposed in recent years [71–74]. Different from all the other parallel simulation techniques which explore intra-algorithm parallelism, i.e., parallel computing is only applied in a single algorithm, multi-algorithm simulation explores parallelism between different algorithms. The starting point of this method is the applicability of different integration algorithms to different circuit behaviors, e.g., an algorithm that is suitable for smooth waveforms may not be suitable for oscillating waveforms. Consequently, using a single algorithm may not be always the best solution in circuit simulation. Instead, running a pool of algorithms of different characteristics can be a better way. The challenge is how to efficiently schedule multiple algorithms and integrate the solutions of multiple algorithms together on the fly.

Figure 2.9 shows the general framework of parallel multi-algorithm simulation. To explore parallelism between algorithms, $n$ different algorithms are running

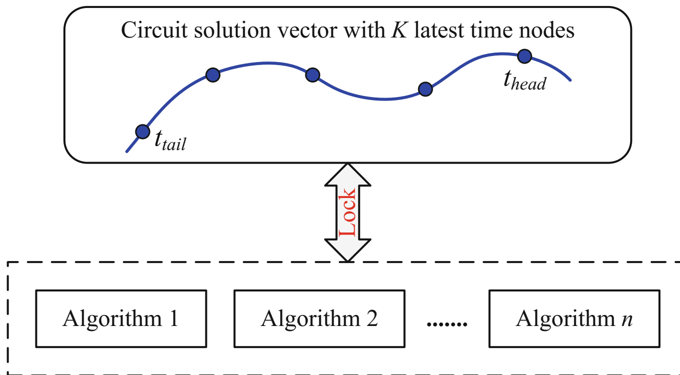



**Fig. 2.9** Framework of parallel multi-algorithm simulation [71–74]

independently in parallel to process the same simulation task. Each algorithm maintains a complete SPICE context including the sparse direct solver, device model evaluation, numerical integration method, Newton–Raphson iterations, etc. Due to the different characteristics of these algorithms, their speeds at the time axis are also different. The high performance is achieved by an algorithm selection strategy. To synchronize the solutions of these algorithms, a solution vector containing $K$ latest time nodes is maintained. Let $t_{head}$ and $t_{tail}$ be the first and last time nodes of the solution vector. As the vector is global and can be accessed by all the algorithms, a lock is required when an algorithm attempts to access it. The update strategy to the solution vector can be described as follows. Once an algorithm finishes solving one time node, it will access the solution vector by acquiring the lock. If the current time node, say $t_{alg}$, is beyond $t_{head}$, then $t_{head}$ is updated by $t_{alg}$ and $t_{tail}$ is moved forward by one node. If $t_{alg}$ is between $t_{tail}$ and $t_{head}$, then $t_{alg}$ is inserted and $t_{tail}$ is still moved forward by one node. However, if $t_{alg}$ is behind $t_{tail}$, it indicates that this algorithm is too slow so the current solution is discarded, and then the current algorithm picks up the latest time node in the solution vector, i.e., $t_{head}$, to calculate the next time node. Additionally, before each algorithm starts to calculate the next new time node, it also checks the solution vector to load the latest time node. Such a scheduling and update policy implies an algorithm selection strategy that always selects the fastest algorithms at any time, so that the speedups over a single algorithm-based simulation can be far beyond the number of used cores.

### *2.3.3 Time-Domain Partitioning*

Different from the above two approaches, a cruder method to implement parallel time-domain simulation is to directly partition the time domain, such that each segment of the time domain can be computed in parallel [75]. The major problem is that the initial solution of each segment, which is necessary in any numerical integration method, is unknown. However, considering a fact that many actual circuits have stationary operation status, with different initial solutions, the circuit will eventually go to a stationary status so the response will finally converge. This fact enables us to simulate the time-domain response in parallel by partitioning the time domain into multiple segments. The initial solution of each segment is selected as the DC operating point. Of course, the waveform obtained by this method has errors. However, if we only need to calculate some high-level or frequency-domain factors of analog circuits, such as the signal to noise-plus-distortion ratio, this method can be applied, because a small error in the waveform does not affect the frequency-domain response. Experimental results show that this method can accelerate analog circuit simulations by more than $50\times$ using 100 cores. However, anyway, such a method is not a unified approach and it can only be applied to special simulations of special circuits.

### 2.3.4 Matrix Exponential Methods

The matrix exponential method [76] is another approach to solve the circuit DAE expressed as Eq. (1.1). Unlike conventional numerical integration methods such as the backward Euler method or the trapezoid method [77] which are implicit, the matrix exponential method is explicit but also A-stable [78].

For the circuit DAE expressed as Eq. (1.1), the matrix exponential method says that its solution within the time interval $[t_n, t_{n+1}]$ can be written as the following form [79]:

$$\mathbf{x}(t_{n+1}) = e^{(t_{n+1}-t_n)\mathbf{J}(\mathbf{x}(t_n))}\mathbf{x}(t_n) + \int_0^{t_{n+1}-t_n} e^{(t_{n+1}-t_n-\tau)\mathbf{J}(\mathbf{x}(t_n))}\mathbf{C}^{-1}(\mathbf{x}(t_n))\left[\boldsymbol{f}(\mathbf{x}(t_n+\tau)) + \mathbf{u}(t_n+\tau)\right]\mathrm{d}\tau, \tag{2.12}$$

where $\mathbf{C}(\mathbf{x}(t_n))$ is a matrix of capacitances and inductances linearized at $t_n$. If we assume that the charges in nonlinear elements behave linearly within the time interval $[t_n, t_{n+1}]$, then the integration can be approximately calculated and the second-order implicit approximation is of the following form:

$$\begin{aligned}\mathbf{x}(t_{n+1}) = &\frac{t_{n+1}-t_n}{2}\mathbf{C}^{-1}(\mathbf{x}(t_{n+1}))\boldsymbol{f}(\mathbf{x}(t_{n+1}))\\ &+ e^{(t_{n+1}-t_n)\mathbf{J}(\mathbf{x}(t_n))}\left[\mathbf{x}(t_n) + \frac{t_{n+1}-t_n}{2}\mathbf{C}^{-1}(\mathbf{x}(t_n))\boldsymbol{f}(\mathbf{x}(t_n))\right]\\ &+ \left(e^{(t_{n+1}-t_n)\mathbf{J}(\mathbf{x}(t_n))} - \mathbf{I}\right)\mathbf{J}^{-1}(\mathbf{x}(t_n))\mathbf{C}^{-1}(\mathbf{x}(t_n))\mathbf{u}(t_n)\\ &+ \left\{\begin{array}{c}\left[e^{(t_{n+1}-t_n)\mathbf{J}(\mathbf{x}(t_n))} - (t_{n+1}-t_n)\mathbf{J}(\mathbf{x}(t_n)) - \mathbf{I}\right]\mathbf{J}^{-2}(\mathbf{x}(t_n))\\ \cdot\dfrac{\mathbf{C}^{-1}(\mathbf{x}(t_{n+1}))\mathbf{u}(t_{n+1}) - \mathbf{C}^{-1}(\mathbf{x}(t_n))\mathbf{u}(t_n)}{t_{n+1}-t_n}\end{array}\right\}.\end{aligned} \tag{2.13}$$

The computation of the matrix exponential $e^{(t_{n+1}-t_n)\mathbf{J}(\mathbf{x}(t_n))}$ can be reduced using Krylov subspace methods [80, 81]. Parallelism can be trivially explored in Krylov subspace methods, as their major operation is just SpMV.

Generally speaking, compared with conventional numerical integration methods, the matrix exponential method has advantages in the performance, accuracy, and scalability. It has been studied in both nonlinear [82–84] and linear [85, 86] circuits simulation. However, as a new technique in SPICE-like circuit simulation, the applicability for general nonlinear circuits, especially for highly stiff systems, still requires further investigations.

## 2.4 Hardware Acceleration Techniques

In recent years, with the rapid development of various accelerators such as GPUs and FPGAs, hardware acceleration techniques are widely used in many areas to accelerate scientific computing. Underlying state-of-the-art accelerators provide much more computing and memory resources than general-purpose CPUs, offering much higher computing capability and memory bandwidth. However, regardless of the claimed generality in computing, there are some architectural limitations that must be dealt with when developing general-purpose applications such as circuit simulation. GPUs and FPGAs have been investigated to accelerate SPICE-like circuit simulation recently. Existing researches are mainly focused on accelerating device model evaluation and the sparse direct solver.

### *2.4.1 GPU Acceleration*

GPUs, known as graphic processors, have been extended to general-purpose computing since about 10 yeas ago. Programming languages including the famous compute unified device architecture (CUDA) [87] and open computing language (OpenCL) [88] have also been developed to help users easily program GPUs. GPUs offer massive thread-level parallelism by integrating thousands of cores in a single processor. Modern GPUs execute programs in a single-instruction-multiple-data (SIMD) manner. This means that, threads are grouped into batches and each batch executes the same instruction on different data. Parallelism is explored both in one batch and between multiple batches. By executing thousands of concurrent threads, the peak performance of high-end GPUs can be one order of magnitude higher than that of state-of-the-art CPUs.

Accelerating device model evaluation by GPUs is straightforward, as the computing processes of all the devices with the same model are almost identical, and there are no inter-model communications. Such a computational pattern can be perfectly mapped to a GPU's SIMD engine. Dozens of speedups can be achieved by GPUs, compared with CPU-based device model evaluation [89–91].

Different from device model evaluation, porting sparse direct solvers onto GPUs faces many challenges. As the SIMD-based GPU architecture is designed for highly regular applications, irregular computational and memory access patterns involved by sparse direct solvers can significantly affect the performance of GPUs, and, hence, they must be well dealt with when implementing sparse direct linear solvers on GPUs. Like most of the CPU-based sparse direct solvers, general-purpose GPU-based sparse direct solvers also gather nonzero elements into dense submatrices and then adopt the CUDA BLAS [92] to solve them [93–101]. Since only small dense submatrices can be formed in sparse matrix factorization, the overhead associated with kernel launching and data transfer between the CPU and GPU can be larger than the computational cost. One can invoke batched BLAS executions on GPUs to avoid

this problem. However, another problem of load imbalance raises, as it is impossible to form a batch of dense submatrices with the identical size. Two GPU-based sparse direct solvers have been proposed for circuit matrices by employing hybrid task-level and data-level parallelism, which invoke a single kernel to process all computations without invoking the CUDA BLAS [102–104].

Basically, sparse direct solvers are memory-intensive applications so the high computing capability of modern GPUs cannot be fully explored. Reported results indicate that most of the existing GPU-based sparse direct solvers can achieve only a few times of speedups compared with single-threaded CPU-based solvers. Such performance can be easily exceeded by a parallel CPU-based solver. The performance of sparse direct solvers running on GPUs is mainly restricted by the off-chip memory bandwidth of GPUs. On the contrary, modern CPUs have a large cache which can significantly reduce the requirement for the off-chip memory bandwidth. From this point of view, it is not a good idea to use present GPUs to accelerate sparse direct solvers, especially for those solvers designed for extremely sparse circuit matrices.

### *2.4.2 FPGA Acceleration*

FPGA is known for the reconfigurability so it has advantages of both general-purpose CPUs and application-specific integrated circuits (ASIC). On one hand, FPGAs can be programmed to different functionalities so they can also be treated as general-purpose processors. On the other hand, the performance of FPGAs can be higher than CPUs and close to that of customized ASICs due to that an FPGA can be reconfigured specifically for a parallel algorithm. In recent years, FPGAs have been widely investigated to accelerate SPICE-like circuit simulation in three respects: device model evaluation, the sparse direct solver, and the whole flow control.

FPGA-based device model evaluation has been proposed in several studies [105–108]. FPGA-based sparse direct solvers have been widely studied in recent a few years [109–115]. Some of them are specially targeted at circuit matrices. Due to the complete reconfigurability, FPGAs can realize very find-grained parallel LU factorization at the basic operation level. Figure 2.10 illustrates an example of the dataflow graph used in [110]. In addition, SPICE iteration control has also been ported onto FPGAs to achieve more speedups [116–118]. Generally speaking, the speedups obtained by FPGA-based acceleration techniques are similar to that of GPUs.

One major shortage of FPGA-based sparse direct solvers is the universality. As an FPGA can be configured to fit a specific matrix, i.e, the FPGA is programmed to fit a specific symbolic pattern and computational flow, once the symbolic pattern of the LU factors changes due to different pivot choices, which also leads to a change of the computational flow, the FPGA needs to be re-programmed. This issue greatly restricts the practicability of FPGA-based sparse direct solvers.

Almost all of the existing hardware acceleration techniques are experimental. It is difficult to apply them in practical applications due to the inflexibility and poor

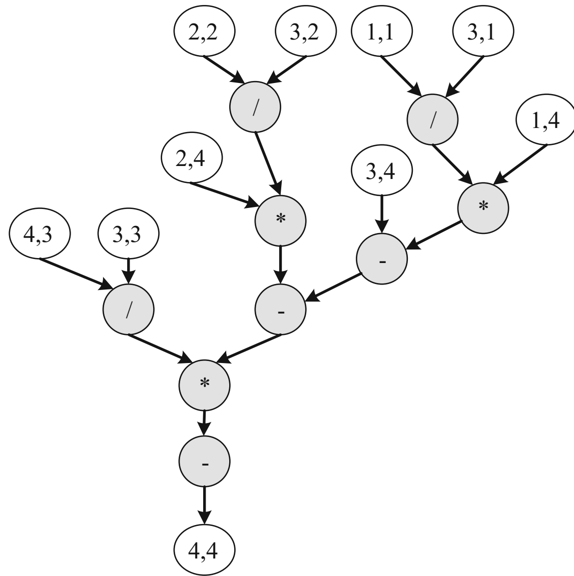



**Fig. 2.10** Dataflow graph for sparse LU factorization on FPGAs [110]

universality of the hardware platforms. For example, memory reallocation and dynamic memory management, which is required by partial pivoting of sparse direct solvers, is difficult to implement on both GPUs and FPGAs. Another important problem is that the performance of hardware platforms may strongly depend on the runtime configurations. For example, performance of many CUDA programs strongly depends on the number of launched threads. The optimal number of threads, in turn, depends on the underlying hardware. Such problems require users to have rich knowledge about the GPU architecture and the code to tune the runtime configurations.

## References

1. Li, P.: Parallel circuit simulation: a historical perspective and recent developments. Found. Trends Electron. Des. Autom. **5**(4), 211–318 (2012)
2. Saleh, R.A., Gallivan, K.A., Chang, M.C., Hajj, I.N., Smart, D., Trick, T.N.: Parallel circuit simulation on supercomputers. Proc. IEEE **77**(12), 1915–1931 (1989)
3. Li, X.S.: Sparse gaussian elimination on high performance computers. Ph.D. thesis, Computer Science Division, UC Berkeley, California, US (1996)
4. Li, X.S., Demmel, J.W.: SuperLU_DIST: a scalable Distributed-Memory sparse direct solver for unsymmetric linear systems. ACM Trans. Math. Softw. **29**(2), 110–140 (2003)
5. Li, X.S.: An overview of SuperLU: algorithms, implementation, and user interface. ACM Trans. Math. Softw. **31**(3), 302–325 (2005)
6. Demmel, J.W., Eisenstat, S.C., Gilbert, J.R., Li, X.S., Liu, J.W.H.: A supernodal approach to sparse partial pivoting. SIAM J. Matrix Anal. Appl. **20**(3), 720–755 (1999)
7. Demmel, J.W., Gilbert, J.R., Li, X.S.: An asynchronous parallel supernodal algorithm for sparse gaussian elimination. SIAM J. Matrix Anal. Appl. **20**(4), 915–952 (1999)

8. Davis, T.A.: Algorithm 832: UMFPACK V4.3-An Unsymmetric-Pattern multifrontal method. ACM Trans. Math. Softw. **30**(2), 196–199 (2004)
9. Davis, T.A., Palamadai Natarajan, E.: Algorithm 907: KLU, A direct sparse solver for circuit simulation problems. ACM Trans. Math. Softw. **37**(3), 36:1–36:17 (2010)
10. Schenk, O., Gärtner, K.: Solving unsymmetric sparse systems of linear equations with PARDISO. Future Gener. Comput. Syst. **20**(3), 475–487 (2004)
11. Schenk, O., Gärtner, K., Fichtner, W.: Efficient sparse LU factorization with Left-Right looking strategy on shared memory multiprocessors. BIT Numer. Math. **40**(1), 158–176 (2000)
12. Schenk, O., Gärtner, K.: Two-Level dynamic scheduling in PARDISO: improved scalability on shared memory multiprocessing systems. Parallel Comput. **28**(2), 187–197 (2002)
13. Amestoy, P.R., Duff, I.S., L'Excellent, J.Y., Koster, J.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM J. Matrix Anal. Appl. **23**(1), 15–41 (2001)
14. Amestoy, P.R., Guermouche, A., L'Excellent, J.Y., Pralet, S.: Hybrid scheduling for the parallel solution of linear systems. Parallel Comput. **32**(2), 136–156 (2006)
15. Amestoy, P., Duff, I., L'Excellent, J.Y.: Multifrontal parallel distributed symmetric and unsymmetric solvers. Comput. Methods Appl. Mech. Eng. **184**(2–4), 501–520 (2000)
16. Gupta, A., Joshi, M., Kumar, V.: WSMP: A High-Performance Shared- and Distributed-Memory parallel sparse linear solver. Technical report, IBM T. J. Watson Research Center (2001)
17. Dongarra, J.J., Cruz, J.D., Hammerling, S., Duff, I.S.: Algorithm 679: a set of level 3 basic linear algebra subprograms: model implementation and test programs. ACM Trans. Math. Softw. **16**(1), 18–28 (1990)
18. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia, PA (1999)
19. Liu, J.W.H.: The role of elimination trees in sparse factorization. SIAM J. Matrix Anal. Appl. **11**(1), 134–172 (1990)
20. Gilbert, J.R., Peierls, T.: Sparse partial pivoting in time proportional to arithmetic operations. SIAM J. Sci. Statist. Comput. **9**(5), 862–874 (1988)
21. Gould, N.I.M., Scott, J.A., Hu, Y.: A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. ACM Trans. Math. Softw. **33**(2), 1–32 (2007)
22. Liu, J.W.H.: The multifrontal method for sparse matrix solution: theory and practice. SIAM Rev. **34**(1), 82–109 (1992)
23. Zitney, S., Mallya, J., Davis, T., therr, M.S.: Multifrontal vs Frontal techniques for chemical process simulation on supercomputers. Comput. Chem. Eng. **20**(6-7), 641–646 (1996)
24. Fischer, M., Dirks, H.: Multigranular parallel algorithms for solving linear equations in VLSI circuit simulation. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **23**(5), 728–736 (2004)
25. Davis, T.A.: Direct Methods for Sparse Linear Systems, 1st edn. Society for Industrial and Applied Mathematics, US (2006)
26. Rajamanickam, S., Boman, E., Heroux, M.: ShyLU: A Hybrid-Hybrid solver for multicore platforms. In: 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS), pp. 631–643 (2012)
27. Zhang, F.: The Schur Complement and Its Applications. Numerical Methods and Algorithms. Springer, Berlin, Germany (2005)
28. Thornquist, H.K., Rajamanickam, S.: A hybrid approach for parallel Transistor-Level Full-Chip circuit simulation. In: International Meeting on High-Performance Computing for Computational Science, pp. 102–111 (2015)
29. MehriDehnavi, M., El-Kurdi, Y., Demmel, J., Giannacopoulos, D.: Communication-Avoiding Krylov techniques on graphic processing units. IEEE Trans. Magn. **49**(5), 1749–1752 (2013)
30. Fowers, J., Ovtcharov, K., Strauss, K., Chung, E.S., Stitt, G.: A High memory bandwidth FPGA accelerator for sparse Matrix-Vector multiplication. In: 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 36–43 (2014)

31. Tang, W.T., Tan, W.J., Ray, R., Wong, Y.W., Chen, W., Kuo, S.H., Goh, R.S.M., Turner, S.J., Wong, W.F.: Accelerating sparse matrix-vector multiplication on GPUs using Bit-Representation-Optimized schemes. In: 2013 SC—International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–12 (2013)
32. Greathouse, J.L., Daga, M.: Efficient sparse Matrix-Vector multiplication on GPUs using the CSR storage format. In: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 769–780 (2014)
33. Ashari, A., Sedaghati, N., Eisenlohr, J., Parthasarath, S., Sadayappan, P.: Fast sparse Matrix-Vector multiplication on GPUs for graph applications. In: SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 781–792 (2014)
34. Grigoras, P., Burovskiy, P., Hung, E., Luk, W.: Accelerating SpMV on FPGAs by compressing nonzero values. In: 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 64–67 (2015)
35. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. Society for Industrial and Applied Mathematics, Boston, US (2004)
36. Basermann, A., Jaekel, U., Hachiya, K.: Preconditioning parallel sparse iterative solvers for circuit simulation. In: Proceedings of the 8th SIAM Proceedings on Applied Linear Algebra, Williamsburg VA (2003)
37. Suda, R.: New iterative linear solvers for parallel circuit simulation. Ph.D. thesis, University of Tokio (1996)
38. Basermann, A., Jaekel, U., Nordhausen, M., Hachiya, K.: Parallel iterative solvers for sparse linear systems in circuit simulation. Future Gener. Comput. Syst. **21**(8), 1275–1284 (2005)
39. Li, Z., Shi, C.J.R.: An efficiently preconditioned GMRES method for fast Parasitic-Sensitive Deep-Submicron VLSI circuit simulation. In: Design, Automation and Test in Europe, Vol. 2, pp. 752–757 (2005)
40. Li, Z., Shi, C.J.R.: A Quasi-Newton preconditioned Newton-Krylov method for robust and efficient Time-Domain simulation of integrated circuits with strong parasitic couplings. Asia S. Pac. Conf. Des. Autom. **2006**, 402–407 (2006)
41. Li, Z., Shi, C.J.R.: A Quasi-Newton preconditioned newton—Krylov method for robust and efficient Time-Domain simulation of integrated circuits with strong parasitic couplings. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **25**(12), 2868–2881 (2006)
42. Zhao, X., Han, L., Feng, Z.: A Performance-Guided graph sparsification approach to scalable and robust SPICE-Accurate integrated circuit simulations. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **34**(10), 1639–1651 (2015)
43. Zhao, X., Feng, Z.: GPSCP: A General-Purpose Support-Circuit preconditioning approach to Large-Scale SPICE-Accurate nonlinear circuit simulations. In: 2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 429–435 (2012)
44. Zhao, X., Feng, Z.: Towards efficient SPICE-Accurate nonlinear circuit simulation with On-the-Fly Support-Circuit preconditioners. In: Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE, pp. 1119–1124 (2012)
45. Bern, M., Gilbert, J.R., Hendrickson, B., Nguyen, N., Toledo, S.: Support-Graph preconditioners. SIAM J. Matrix Anal. Appl. **27**(4), 930–951 (2006)
46. Thornquist, H.K., Keiter, E.R., Hoekstra, R.J., Day, D.M., Boman, E.G.: A parallel preconditioning strategy for efficient Transistor-Level circuit simulation. In: 2009 IEEE/ACM International Conference on Computer-Aided Design—Digest of Technical Papers, pp. 410–417 (2009)
47. Chan, K.W.: Parallel algorithms for direct solution of large sparse power system matrix equations. IEE Proc.—Gener. Transm. Distrib. **148**(6), 615–622 (2001)
48. Zecevic, A.I., Siljak, D.D.: Balanced decompositions of sparse systems for multilevel parallel processing. IEEE Trans. Circuits Syst. I: Fundam. Theory Appl. **41**(3), 220–233 (1994)
49. Koester, D.P., Ranka, S., Fox, G.C.: Parallel Block-Diagonal-Bordered sparse linear solvers for electrical power system applications. In: Proceedings of the Scalable Parallel Libraries Conference, 1993, pp. 195–203 (1993)

50. Paul, D., Nakhla, M.S., Achar, R., Nakhla, N.M.: Parallel circuit simulation via binary link formulations (PvB). IEEE Trans. Compon. Packag. Manuf. Technol. **3**(5), 768–782 (2013)
51. Hu, Y.F., Maguire, K.C.F., Blake, R.J.: Ordering unsymmetric matrices into bordered block diagonal form for parallel processing. In: Euro-Par'99 Parallel Processing: 5th International Euro-Par Conference Toulouse, pp. 295–302 (1999)
52. Aykanat, C., Pinar, A., Çatalyürek, U.V.: Permuting sparse rectangular matrices into Block-Diagonal form. SIAM J. Sci. Comput. **25**(6), 1860–1879 (2004)
53. Duff, I.S., Scott, J.A.: Stabilized bordered block diagonal forms for parallel sparse solvers. Parallel Comput. **31**(3–4), 275–289 (2005)
54. Frohlich, N., Riess, B.M., Wever, U.A., Zheng, Q.: A new approach for parallel simulation of VLSI circuits on a transistor level. IEEE Trans. Circuits Syst. I: Fundam. Theory Appl. **45**(6), 601–613 (1998)
55. Honkala, M., Roos, J., Valtonen, M.: New multilevel Newton-Raphson method for parallel circuit simulation. Proc. Eur. Conf. Circuit Theory Des. **1**, 113–116 (2001)
56. Zhu, Z., Peng, H., Cheng, C.K., Rouz, K., Borah, M., Kuh, E.S.: Two-Stage Newton-Raphson method for Transistor-Level simulation. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **26**(5), 881–895 (2007)
57. Rabbat, N., Sangiovanni-Vincentelli, A., Hsieh, H.: A multilevel newton algorithm with macromodeling and latency for the analysis of Large-Scale nonlinear circuits in the time domain. IEEE Trans. Circuits Syst. **26**(9), 733–741 (1979)
58. Smith, B., Bjorstad, P., Gropp, W.: Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations, 1st edn. Cambridge University Press (2004)
59. Peng, H., Cheng, C.K.: Parallel transistor level circuit simulation using domain decomposition methods. In: 2009 Asia and South Pacific Design Automation Conference, pp. 397–402 (2009)
60. Peng, H., Cheng, C.K.: Parallel transistor level full-Chip circuit simulation. In: 2009 Design, Automation Test in Europe Conference Exhibition, pp. 304–307 (2009)
61. Lelarasmee, E., Ruehli, A.E., Sangiovanni-Vincentelli, A.L.: The waveform relaxation method for Time-Domain analysis of large scale integrated circuits. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **1**(3), 131–145 (1982)
62. Achar, R., Nakhla, M.S., Dhindsa, H.S., Sridhar, A.R., Paul, D., Nakhla, N.M.: Parallel and scalable transient simulator for power grids via waveform relaxation (PTS-PWR). IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **19**(2), 319–332 (2011)
63. Odent, P., Claesen, L., Man, H.D.: A combined waveform Relaxation-Waveform relaxation newton algorithm for efficient parallel circuit simulation. In: Proceedings of the European Design Automation Conference, 1990, EDAC, pp. 244–248 (1990)
64. Rissiek, W., John, W.: A dynamic scheduling algorithm for the simulation of MOS and Bipolar circuits using waveform relaxation. In: Design Automation Conference, 1992, EURO-VHDL '92, EURO-DAC '92. European, pp. 421–426 (1992)
65. Saviz, P., Wing, O.: PYRAMID-A hierarchical waveform Relaxation-Based circuit simulation program. In: IEEE International Conference on Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers, pp. 442–445 (1988)
66. Erdman, D.J., Rose, D.J.: A newton waveform relaxation algorithm for circuit simulation. In: 1989 IEEE International Conference on Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers, pp. 404–407 (1989)
67. Saviz, P., Wing, O.: Circuit simulation by hierarchical waveform relaxation. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **12**(6), 845–860 (1993)
68. Fang, W., Mokari, M.E., Smart, D.: Robust VLSI circuit simulation techniques based on overlapped waveform relaxation. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **14**(4), 510–518 (1995)
69. Gristede, G.D., Ruehli, A.E., Zukowski, C.A.: Convergence properties of waveform relaxation circuit simulation methods. IEEE Trans. Circuits Syst. I: Fundam. Theory Appl. **45**(7), 726–738 (1998)
70. Dong, W., Li, P., Ye, X.: WavePipe: parallel transient simulation of analog and digital circuits on Multi-Core Shared-Memory machines. In: Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE, pp. 238–243 (2008)

71. Ye, X., Dong, W., Li, P., Nassif, S.: MAPS: Multi-Algorithm parallel circuit simulation. In: 2008 IEEE/ACM International Conference on Computer-Aided Design, pp. 73–78 (2008)
72. Ye, X., Li, P.: Parallel program performance modeling for runtime optimization of Multi-Algorithm circuit simulation. In: 2010 47th ACM/IEEE Design Automation Conference (DAC), pp. 561–566 (2010)
73. Ye, X., Li, P.: On-the-fly runtime adaptation for efficient execution of parallel Multi-Algorithm circuit simulation. In: 2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 298–304 (2010)
74. Ye, X., Dong, W., Li, P., Nassif, S.: Hierarchical multialgorithm parallel circuit simulation. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **30**(1), 45–58 (2011)
75. Ye, Z., Wu, B., Han, S., Li, Y.: Time-Domain segmentation based massively parallel simulation for ADCs. In: Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE, pp. 1–6 (2013)
76. Chua, L.O., Lin, P.Y.: Computer-Aided analysis of electronic circuits: algorithms and computational techniques, 1st edn. Prentice Hall Professional Technical Reference (1975)
77. Süli, E., Mayers, D.F.: An Introduction to Numerical Analysis, 2nd edn. Cambridge University Press, England (2003)
78. Dahlquist, G.G.: A special stability problem for linear multistep methods. BIT Numer. Math. **3**(1), 27–43 (1963)
79. Nie, Q., Zhang, Y.T., Zhao, R.: Efficient Semi-Implicit schemes for stiff systems. J. Comput. Phys. **214**(2), 521–537 (2006)
80. Hochbruck, M., Lubich, C.: On Krylov subspace approximations to the matrix exponential operator. SIAM J. Numer. Anal. **34**(5), 1911–1925 (1997)
81. Saad, Y.: Analysis of some Krylov subspace approximations to the matrix exponential operator. SIAM J. Numer. Anal. **29**(1), 209–228 (1992)
82. Zhuang, H., Wang, X., Chen, Q., Chen, P., Cheng, C.K.: From circuit theory, simulation to SPICE_Diego: a matrix exponential approach for Time-Domain analysis of Large-Scale circuits. IEEE Circuits Syst. Mag. **16**(2), 16–34 (2016)
83. Zhuang, H., Yu, W., Kang, I., Wang, X., Cheng, C.K.: An algorithmic framework for efficient Large-Scale circuit simulation using exponential integrators. In: 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6 (2015)
84. Weng, S.H., Chen, Q., Wong, N., Cheng, C.K.: Circuit simulation via matrix exponential method for stiffness handling and parallel processing. In: 2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 407–414 (2012)
85. Chen, Q., Zhao, W., Wong, N.: Efficient matrix exponential method based on extended Krylov subspace for transient simulation of Large-Scale linear circuits. In: 2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 262–266 (2014)
86. Zhuang, H., Weng, S.H., Lin, J.H., Cheng, C.K.: MATEX: A distributed framework for transient simulation of power distribution networks. In: 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6 (2014)
87. NVIDIA Corporation: NVIDIA CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
88. Khronos OpenCL Working Group: The OpenCL Specification v1.1 (2010)
89. Gulati, K., Croix, J.F., Khatri, S.P., Shastry, R.: Fast circuit simulation on graphics processing units. In: 2009 Asia and South Pacific Design Automation Conference, pp. 403–408 (2009)
90. Poore, R.E.: GPU-Accelerated Time-Domain circuit simulation. In: 2009 IEEE Custom Integrated Circuits Conference, pp. 629–632 (2009)
91. Bayoumi, A.M., Hanafy, Y.Y.: Massive parallelization of SPICE device model evaluation on GPU-based SIMD architectures. In: Proceedings of the 1st International Forum on Next-generation Multicore/Manycore Technologies, pp. 12:1–12:5 (2008)
92. NVIDIA Corporation: CUDA BLAS. http://docs.nvidia.com/cuda/cublas/
93. Christen, M., Schenk, O., Burkhart, H.: General-Purpose sparse matrix building blocks Using the NVIDIA CUDA technology platform. In: First Workshop on General Purpose Processing on Graphics Processing Units. Citeseer (2007)

94. Krawezik, G.P., Poole, G.: Accelerating the ANSYS direct sparse solver with GPUs. In: 2009 Symposium on Application Accelerators in High Performance Computing (SAAHPC'09) (2009)
95. Yu, C.D., Wang, W., Pierce, D.: A CPU-GPU hybrid approach for the unsymmetric multifrontal method. Parallel Comput. **37**(12), 759–770 (2011)
96. George, T., Saxena, V., Gupta, A., Singh, A., Choudhury, A.: Multifrontal factorization of sparse SPD matrices on GPUs. In: 2011 IEEE International Parallel Distributed Processing Symposium (IPDPS), pp. 372–383 (2011)
97. Lucas, R.F., Wagenbreth, G., Tran, J.J., Davis, D.M.: Multifrontal Sparse Matrix Factorization on Graphics Processing Units. Technical report. Information Sciences Institute, University of Southern California (2012)
98. Lucas, R.F., Wagenbreth, G., Davis, D.M., Grimes, R.: Multifrontal computations on GPUs and their Multi-Core Hosts. In: Proceedings of the 9th International Conference on High Performance Computing for Computational Science, pp. 71–82 (2011)
99. Kim, K., Eijkhout, V.: Scheduling a parallel sparse direct solver to multiple GPUs. In: 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops Ph.D. Forum (IPDPSW), pp. 1401–1408 (2013)
100. Hogg, J.D., Ovtchinnikov, E., Scott, J.A.: A sparse symmetric indefinite direct solver for GPU architectures. ACM Trans. Math. Softw. **42**(1), 1:1–1:25 (2016)
101. Sao, P., Vuduc, R., Li, X.S.: A distributed CPU-GPU sparse direct solver. In: Euro-Par 2014 Parallel Processing: 20th International Conference, pp. 487–498 (2014)
102. Ren, L., Chen, X., Wang, Y., Zhang, C., Yang, H.: Sparse LU factorization for parallel circuit simulation on GPU. In: Proceedings of the 49th Annual Design Automation Conference. DAC '12, pp. 1125–1130. ACM, New York, NY, USA (2012)
103. Chen, X., Ren, L., Wang, Y., Yang, H.: GPU-Accelerated sparse LU factorization for circuit simulation with performance modeling. IEEE Trans. Parallel Distrib. Syst. **26**(3), 786–795 (2015)
104. He, K., Tan, S.X.D., Wang, H., Shi, G.: GPU-Accelerated parallel sparse LU factorization method for fast circuit analysis. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **24**(3), 1140–1150 (2016)
105. Kapre, N., DeHon, A.: Accelerating SPICE Model-Evaluation using FPGAs. In: 17th IEEE Symposium on Field Programmable Custom Computing Machines, 2009. FCCM '09, pp. 37–44 (2009)
106. Kapre, N.: Exploiting input parameter uncertainty for reducing datapath precision of SPICE device models. In: 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 189–197 (2013)
107. Martorell, H., Kapre, N.: FX-SCORE: a framework for fixed-point compilation of SPICE device models using Gappa++. In: Field-Programmable Custom Computing Machines (FCCM), pp. 77–84 (2012)
108. Kapre, N., DeHon, A.: Performance comparison of Single-Precision SPICE Model-Evaluation on FPGA, GPU, Cell, and Multi-Core processors. In: 2009 International Conference on Field Programmable Logic and Applications, pp. 65–72 (2009)
109. Wu, W., Shan, Y., Chen, X., Wang, Y., Yang, H.: FPGA accelerated parallel sparse matrix factorization for circuit simulations. In: Reconfigurable Computing: Architectures, Tools and Applications: 7th International Symposium, ARC 2011, pp. 302–315 (2011)
110. Kapre, N., DeHon, A.: Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs. In: International Conference on Field-Programmable Technology, 2009. FPT 2009, pp. 190–198 (2009)
111. Wang, X., Jones, P.H., Zambreno, J.: A configurable architecture for sparse LU decomposition on matrices with arbitrary patterns. SIGARCH Comput. Archit. News **43**(4), 76–81 (2016)
112. Wu, G., Xie, X., Dou, Y., Sun, J., Wu, D., Li, Y.: Parallelizing sparse LU decomposition on FPGAs. In: 2012 International Conference on Field-Programmable Technology (FPT), pp. 352–359 (2012)

113. Johnson, J., Chagnon, T., Vachranukunkiet, P., Nagvajara, P., Nwankpa, C.: Sparse LU decomposition using FPGA. In: International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA) (2008)
114. Siddhartha, Kapre, N.: Heterogeneous dataflow architectures for FPGA-based sparse LU factorization. In: 2014 24th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–4 (2014)
115. Siddhartha, Kapre, N.: Breaking sequential dependencies in FPGA-Based sparse LU factorization. In: 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 60–63 (2014)
116. Kapre, N., DeHon, A.: VLIW-SCORE: beyond C for sequential control of SPICE FPGA acceleration. In: 2011 International Conference on Field-Programmable Technology (FPT), pp. 1–9 (2011)
117. Kapre, N., DeHon, A.: SPICE2: spatial processors interconnected for concurrent execution for accelerating the SPICE circuit simulator using an FPGA. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **31**(1), 9–22 (2012)
118. Kapre, N.: SPICE2—A spatial parallel architecture for accelerating the SPICE circuit simulator. Ph.D. thesis, California Institute of Technology (2010)

http://www.springer.com/978-3-319-53428-2

Parallel Sparse Direct Solver for Integrated Circuit Simulation
Chen, X.; Wang, Y.; Yang, H.
2017, IX, 129 p. 51 illus., 43 illus. in color., Hardcover
ISBN: 978-3-319-53428-2