# NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation

Xiaoming Chen, *Student Member, IEEE,* Yu Wang, *Member, IEEE,* and Huazhong Yang, *Senior Member, IEEE*

***Abstract*—The sparse matrix solver has become a bottleneck in simulation program with integrated circuit emphasis (SPICE)-like circuit simulators. It is difficult to parallelize the solver because of the high data dependency during the numeric LU factorization and the irregular structure of circuit matrices. This paper proposes an adaptive sparse matrix solver called NICSLU, which uses a multithreaded parallel LU factorization algorithm on shared-memory computers with multicore/multisocket central processing units to accelerate circuit simulation. The solver can be used in all the SPICE-like circuit simulators. A simple method is proposed to predict whether a matrix is suitable for parallel factorization, such that each matrix can achieve optimal performance. The experimental results on 35 matrices reveal that NICSLU achieves speedups of 2.08× ∼ 8.57× (on the geometric mean), compared with KLU, with 1 ∼ 12 threads, for the matrices which are suitable for the parallel algorithm. NICSLU can be downloaded from http://nicslu.weebly.com.**

***Index Terms*—Parallel circuit simulation, parallel LU factorization.**

## I. Introduction

A SIMULATION program with integrated circuit emphasis (SPICE) [1] circuit simulator developed by the University of California, Berkeley, is widely used for transistor-level simulation in integrated circuit (IC) design and verification. However, with the growing complexity of the very large-scale IC at nanoscale, the traditional SPICE-like simulators require more and more time to perform accurate simulations. It can often take days or even weeks to perform post-layout simulations on modern processors. So, in recent decades, parallel circuit simulation has attracted researchers' interest, and many attempts have been made to accelerate SPICE simulation by parallel algorithms.

Fig. 1 shows a typical SPICE flow of transient simulation. SPICE runtime is dominated by two steps: model evaluation and the sparse matrix solver. Parallelization of model evaluation is straightforward, but the sparse matrix solver by LU factorization [2] is difficult to be parallelized because of the high data dependency during LU factorization and the irregular structure of circuit matrices. In addition, the solver is repeated in every Newton–Raphson iteration during the transient analysis. So it has become the bottleneck in the SPICE flow [3].

One typical LU factorization has three steps: 1) preprocessing; 2) numeric factorization; and 3) right-hand solving (substitution) [2]. The first step performs column/row permutations to reduce fill-ins, the second step factorizes matrix $A$ into the product of a lower triangular matrix $L$ and an upper triangular matrix $U$, which is the most time-consuming step among the three steps, and the last step solves the triangular equations $Ly = b$ and $Ux = y$. An important feature of the circuit simulation flow is that, although the values of matrix elements change during the Newton–Raphson iterations and transient iterations, the nonzero pattern of matrix $A$ remains unchanged, so the preprocessing step can be performed only once, while the last two steps are performed many times for the iterations. Consequently, the bottleneck becomes the numeric factorization step.

In this paper, we propose a parallel matrix solver NICSLU by sparse LU factorization algorithm, based on the Gilbert–Peierls (G–P) algorithm [4] and KLU algorithm [5], to accelerate circuit simulation on shared-memory computers with multicore/multisocket central processing units (CPUs). The features of NICSLU are as follows.

1) It is performed in column-level parallelism, which is suitable for the sparsity of circuit matrices [5].
2) It automatically decides if a matrix is suitable for the sequential or parallel algorithm.
3) Two parallel modes (cluster mode and pipeline mode) are statically scheduled to fit the different data dependency and reduce the scheduling overheads.
4) It is based on shared-memory machines and can be conveniently used on multicore/multisocket computers.
5) It can be used in all the SPICE-like circuit simulators.

Fig. 2 shows the overall flow of NICSLU. We have two parallel algorithms: without partial pivoting [6] and with partial pivoting [7]. The latter will be introduced in this paper, which is extended from our conference paper [7]. The former is proposed in [6], which is performed without partial pivoting. Since the matrix elements may change drastically during the Newton–Raphson iterations and the transient iterations, it is

Manuscript received April 19, 2012; revised July 12, 2012; accepted August 27, 2012. Date of current version January 18, 2013. This work was supported in part by the National Science and Technology Major Project under Grants 2011ZX01035-001-001-002, 2010ZX01030-001-001-04, and 2011ZX03003-003-01, in part by the National Basic Research Program of China (973 Program), in part by the National Natural Science Foundation of China under Grants 61028006 and 61076035, and in part by the Tsinghua University Initiative Scientific Research Program. This paper was recommended by Associate Editor P. Li.

The authors are with the Department of Electronic Engineering, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: chenxm05@mails.tsinghua.edu.cn; yu-wang@tsinghua.edu.cn; yanghz@tsinghua.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2012.2217964

0278-0070/$31.00 © 2013 IEEE

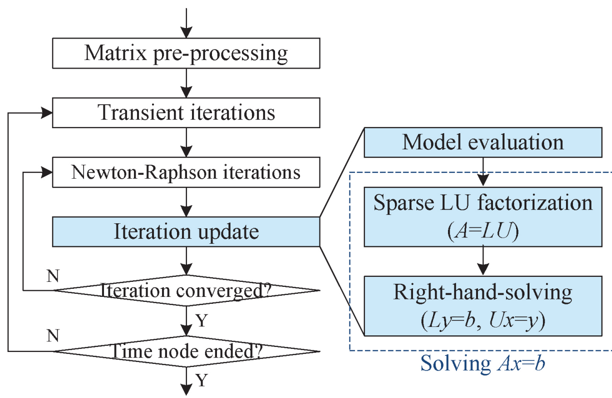



Fig. 1. Typical SPICE flow of transient simulation.

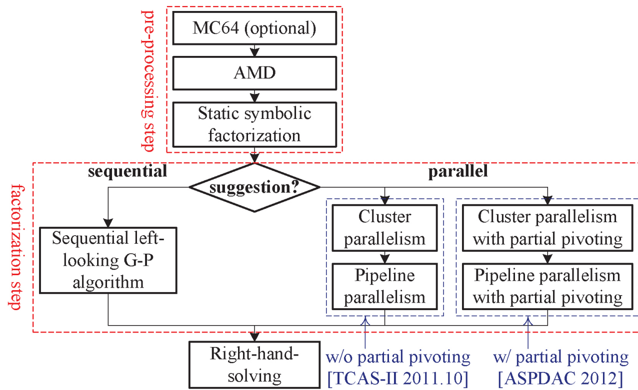



Fig. 2. Overall flow of NICSLU.

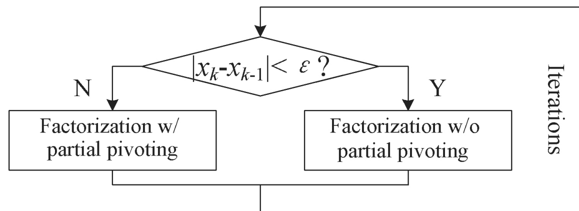



Fig. 3. Usage of NICSLU in circuit simulation.

possible to generate small elements or even zeros on the diagonal during numeric factorization (leading to unstable algorithm), especially in digital circuit simulation. So partial pivoting is needed. The usage of the two algorithms is illustrated in Fig. 3. Before each Newton–Raphson iteration, the residual of the solutions in the last two iterations is calculated. If the residual is large enough, the algorithm with partial pivoting is performed. Otherwise, it means that the matrix values in the last two iterations do not change much, which also means that this round of Newton–Raphson iterations will converge, so the pivoting order generated by the last factorization can be used in subsequent factorizations to ensure numeric stability. This paper is not a simple improvement of [6] but a new parallel algorithm, and the two algorithms are both useful in circuit simulation. The contributions of this paper are summarized as follows.

1) Not every matrix is suitable for parallel factorization, so we present a predictive method in Section III to decide if a matrix should use the sequential or parallel solver. Consequently, each matrix can achieve the optimal performance.
2) This paper is implemented with partial pivoting. When adopting partial pivoting, the exact data dependency cannot be determined prior to factorization; the primary challenge is to predict the data dependency before factorization and dynamically determine the exact dependency during factorization. This problem is solved by the elimination tree (ETree) [8] and our proposed scheduling method in Section V. Our solver is $2.08\times \sim 8.57\times$ (on geometric mean) faster than KLU, with $1 \sim 12$ threads, for the matrices that are suitable for the parallel algorithm.
3) The effect of the preprocessing algorithms is evaluated. When NICSLU and KLU use the same preprocessing algorithm, NICSLU still performs better than KLU, which is given in Section VI-D.
4) A task flow graph-based performance model is proposed in Section VII to predict the speedups, taking into account the number of available threads.

The remainder of this paper is organized as follows. Section II shows some related work about parallel circuit simulation. The preprocessing algorithms of NICSLU are shown in Section III. The basics of LU factorization algorithm are introduced in Section IV. The proposed parallel LU factorization algorithm is illustrated in Section V. We show the experimental results and analysis in Section VI. Section VII presents a performance model. Finally, conclusions are drawn in Section VIII.

## II. Related Work

Research on parallel circuit simulation has been active for decades, and there are several primary categories of parallel approaches for SPICE-accurate simulation.

### A. Fast Direct Solver

The traditional and straightforward parallel strategy is to parallelize the sparse matrix solver. There are some popular software implementations of LU factorization, such as SuperLU [9], SuperLU_MT [8], SuperLU_DIST [10], KLU [5], UMFPACK [11], MUMPS [12], PARDISO [13], and so on. Among them, SuperLU and PARDISO incorporates *supernodes* to enhance the computational capability when computing dense blocks, and UMFPACK and MUMPS are based on multifrontal algorithm [14], which is also based on the idea of using dense submatrices. However, because of the high sparsity of circuit matrices, it is hard to form supernodes or dense blocks in circuit simulation. Among all the software implementations, only KLU is specially optimized for circuit simulation, however, KLU has no parallel version. A multigranularity parallel LU factorization algorithm has been proposed in [15]; however, it is performed for only symmetric matrices. Actually, in nonlinear circuit simulation, the matrix is usually asymmetric, so symmetric LU factorization is not useful; in addition, for symmetric matrices, Cholesky factorization [2] is about twice as efficient than LU factorization.

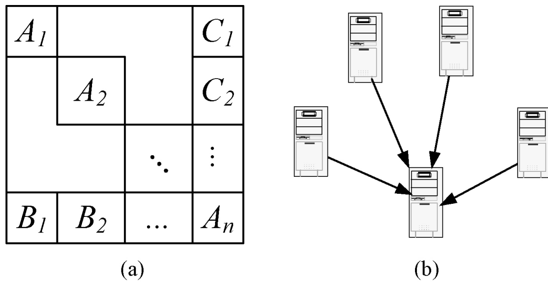



Fig. 4. (a) BBD structure and (b) master–slave parallel mode.

### B. Iterative Methods

In [16], a preconditioned generalized minimal residual (GMRES) method [17] was used to solve the linear equations for circuits with strong parasitic couplings; this paper is limited to the piecewise weakly nonlinear transistor model but does not have universality. Thornquist *et al.* also used preconditioned GMRES for transistor-level simulation [18]. Since iterative method [19] requires precondition in each Newton–Raphson iteration, its advantages for circuit simulation need to be proved. A recent solver called ShyLU [20], which combines direct method and iterative method, is not specially targeted at circuit matrices. Other methods, such as conjugate gradient [21] and multigrid [22], are usually used for linear circuit simulation [23], [24].

### C. Hardware Acceleration

Several parallel approaches have recently been developed on field-programmable gate array (FPGA) [3], [25]–[28] and graphics processing unit (GPU) [29]–[31] to accelerate the matrix solver and/or the model evaluation step for circuit simulation. Some of them have attractive performance; however, the scalability to large-scale circuits is limited by FPGA and GPU on-chip resources.

### D. Circuit Partitioning

Some circuit partitioning-based parallel simulation methods have been proposed in [32]–[34]. Most of these methods partition a circuit into several small subcircuits to build the bordered-block-diagonal (BBD) matrix [35], as shown in Fig. 4(a). However, although the solution of each diagonal block can be independently parallelized, the right-bottom block is a serious bottleneck, as BBD-based approach is in a master–slave mode, all the slave machines send data to the master machine [Fig. 4(b)].

The above partitioning-based methods are all flat partitioning-based simulation approaches. Unlike these, a hierarchical method is proposed in [36] to simulate the power distribution networks, using macromodels to abstract the subblocks. However, the abstracted port admittance matrices for the subblocks are dense, which may lead to lots of runtime and memory occupation, so they are sparsified by dropping some small values, leading to inaccurate results. Similar to the hierarchical method in [36], some multilevel Newton–Raphson methods are proposed in [37]–[39]. They abstract the subblocks by calculating the Jacobi matrices.

Another circuit partitioning-based parallel approach is waveform relaxation [40]. However, it usually requires a grounded capacitor at each node to guarantee convergence and often performs poorly for feedback circuits.

### E. Transient-Domain Parallelization

The parallelism of all the aformentioned methods is performed at one time node during the transient iteration. The parallelism can be also explored among different time nodes. This method is called Wavepipe [41]; however, the reported scalability is not very good. Another approach that explores parallelism among different algorithms in transient simulation, called multialgorithm parallelism [42], [43], needs many redundant computation resources.

Although the scalability of the circuit partitioning-based approaches may be better than the fast solver-based methods, the parallel matrix solver for circuit simulation is still a problem that has not been well solved. So, in this paper, we develop a parallel matrix solver for circuit simulation, which can be used in traditional SPICE-like parallel simulators or in circuit partitioning-based methods, to solve each submatrix.

## III. Preprocessing Algorithms of NICSLU

The preprocessing step of NICSLU consists of three algorithms (see Fig. 2): the MC64 algorithm [44], [45] from the HSL Mathematical Software Library [46], the approximate minimum degree (AMD) algorithm [47], and a static symbolic factorization [5], [48], [49]. The preprocessing step is performed only once in circuit simulation.

### A. MC64 Algorithm

The MC64 algorithm [44], [45] first finds a column permutation matrix $P_c$ to maximize the sum or product of the diagonal elements of $AP_c$. It then finds two diagonal scaling matrices $D_r$ and $D_c$, to make each diagonal element of $A_1 = D_r A P_c D_c$ is $\pm 1$, and each off-diagonal element is bounded by 1 in absolute value. The MC64 algorithm can effectively enhance the numeric stability. This step is optional in our solver.

### B. AMD Algorithm

The AMD algorithm [47] is a fill-reducing ordering algorithm to minimize fill-ins for LU factorization, which finds a symmetric row/column permutation matrix $P$ and then performs the symmetric ordering on the symmetric matrix $A_1 + A_1^T$, i.e., $A_2 = P(A_1 + A_1^T)P^T$, such that factorizing $A_2$ has much fewer fill-ins than factorizing $A_1$. Since the runtime of LU factorization is correlative to the fill-ins, the AMD algorithm can reduce LU factorization time effectively.

### C. Static Symbolic Factorization

This step is used to give a suggestion as to whether a matrix should use the parallel or sequential algorithm. It is based on the symbolic factorization algorithm in [5], [48], and [49], without any partial pivoting or numeric computation. After this step, we obtain the nonzero structure of $L$ and $U$ (the nonzero

**Algorithm 1** Left-looking G–P algorithm

$L = I$;
**for** $k = 1 : n$ **do**
  **Symbolic factorization**: determine the structure of column $k$, and which columns will update column $k$;
  **Numeric factorization**: solve $Lx = b$, where $b = A(:, k)$;
  **Partial pivoting** on $x$;
  $U(1 : k, k) = x(1 : k)$;
  $L(k : n, k) = x(k : n)/U(k, k)$;
**end for**

structure obtained here is not the actual structure, since the actual structure depends on the pivot choices). In addition, the number of floating-point operations (FLOPs) to factorize the matrix is also predicted. Then we propose a simple method to give the suggestion based on the following two values.

1) $R_1 = \frac{\text{NNZ}(L+U)}{\text{NNZ}(A)}$, where NNZ$(L + U)$ is the number of nonzeros in $L + U$, which is predicted by the static symbolic factorization, and NNZ$(A)$ is the number of nonzeros in $A$. $R_1$ can be regarded as the relative fill-in.
2) $R_2 = \frac{\text{FLOPs}}{\text{NNZ}(L+U)}$, which can be regarded as the average number of FLOPs of each nonzero element in $L + U$.

We suggest that if ($R_1 \geq 2.0$ || $R_2 \geq 50$), the parallel algorithm should be used; otherwise (i.e., ($R_1 < 2.0$ && $R_2 < 50$)) the sequential algorithm should be used rather than the parallel algorithm. A detailed discussion will be presented in Section VI-C.

The preprocessing step of NICSLU is different from KLU. In KLU, the matrix is permuted into a block triangular form (BTF) [50] according to the characteristics of circuit matrices, but without the MC64 algorithm. Since BTF can affect the matrix structure, and MC64 can affect the numeric results and pivot choices, a detailed discussion will be presented in Section VI-D.

In the following content, we will still regard the matrix after preprocessing step as $A$.

## IV. LU Factorization Basics

The basic LU factorization algorithm we utilized is the left-looking G–P algorithm [4], which is shown in Algorithms 1 and 2. It factorizes matrix $A$ by sequentially processing each column ($k$) in four steps: 1) symbolic factorization; 2) numeric factorization; 3) partial pivoting; and 4) storing $x$ in to $U$ and $L$ (with normalization). The first three steps are introduced as follows.

### A. Symbolic Factorization

The purpose of symbolic factorization is to determine the nonzero structure of the $k$th column, and which columns will update column $k$ (i.e., the dependency). The symbolic factorization of column $k$ depends on the symbolic structure of columns $1 \sim k - 1$. Gilbert [49] has presented the theory of symbolic factorization, a detailed depth first search-based implementation can be found in [5] and [48]. After this step, the numeric factorization is performed only on the nonzeros but not all the matrix elements, since this algorithm is targeted at sparse matrices.

**Algorithm 2** Solving $Lx = b$, where $b = A(:, k)$

$x = b$;
**for** $j < k$ where $U(j, k) \neq 0$ **do**
  $x(j + 1 : n) = x(j + 1 : n) - L(j + 1 : n, j) \cdot x(j)$;
**end for**

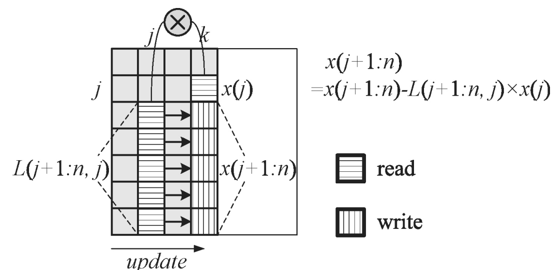



Fig. 5. Primary operation (vector multiply-and-add) in the left-looking algorithm.

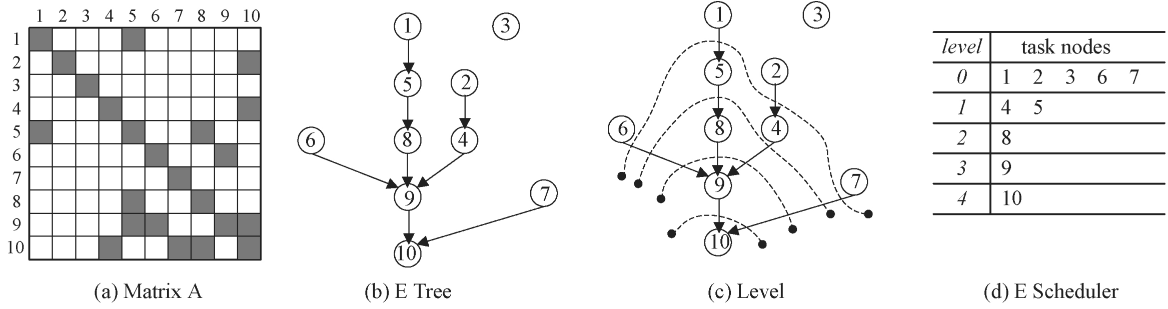
### B. Numeric Factorization

This step is the most time-consuming step, and is shown in Algorithm 2; the primary operation is the vector multiply-and-add operation (Fig. 5). It uses all the dependent columns to perform the numeric update on column $k$. The pseudocode in Algorithm 2 indicates that the numeric factorization of column $k$ refers to the factorization results of some previous (left side) columns $\{j|U(j, k) \neq 0, j < k\}$. In other words, column $k$ depends on column $j$, iff $U(j, k) \neq 0 (j < k)$, which means the column-level dependency is determined by the structure of $U$. If matrix $A$ is dense, $U$ is also dense; so column $k$ depends on all of its left columns (i.e., $1, 2, \ldots, k - 1$) and it is a completely sequential algorithm in column level. However, when matrix $A$ and $U$ are sparse, column $k$ only depends on part of its left columns, the parallelism can be explored from the sparsity. This is the fundamental column-level dependency in sparse left-looking LU factorization. In the next section, we will introduce the column-level dependency based parallel LU factorization algorithm.

### C. Partial Pivoting

Although we can use MC64 algorithm in the preprocessing step to enhance numeric stability, it is still possible to generate small pivots or even zeros on the diagonal during the Newton–Raphson iterations and the transient iterations (leading to unstable algorithm), especially in digital circuit simulation (the matrix entries may vary drastically according to the digital circuit operation). So partial pivoting is still needed. It is done by finding the largest element in each column of $L$, swapping it to the diagonal when the original diagonal entry is small enough [4], [48]. However, partial pivoting violates the matrix permutation obtained by AMD algorithm in the preprocessing step, so off-diagonal pivoting can increase fill-ins dramatically.

Even worse, when adopting partial pivoting, the actual nonzero structure of $L$ and $U$ depends on the pivot choices (partial pivoting can interchange the rows). Since the column-level dependency is determined by the structure of $U$, the actual dependency cannot be determined before LU factorization. The challenge is to dynamically determine the actual column-



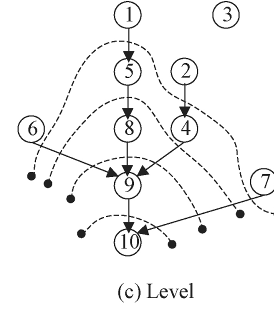


Fig. 6. Example to illustrate the ETree, level, and the EScheduler.

level dependency during LU factorization. This difference leads to the differences between this paper and [6].

## V. Parallel LU Factorization Strategies

In this section, we introduce the column-level parallel LU factorization algorithm, which is developed based on the sequential left-looking G-P algorithm [4] and KLU algorithm [5]; the parallelism is explored from the analysis of the data dependency.

### *A. Dependency Determination*

As mentioned above, the column-level dependency is determined by the structure of $U$. In [6], since partial pivoting is not adopted, the nonzero structure of $L$ and $U$ calculated by the symbolic factorization in the preprocessing step is the actual nonzero structure; the exact column-level dependency is directly obtained from the structure of $U$. An elimination scheduler (EScheduler) is directly built from the structure of $U$ to represent the column-level dependency, and then the parallel tasks are scheduled by two methods: cluster mode and pipeline mode. However, when adopting partial pivoting, the symbolic factorization cannot be separated from the numeric factorization because the nonzero structure depends on the numeric pivot choices, which means the nonzero structure obtained by the symbolic factorization in the preprocessing step is not actual. So the exact column-level dependency cannot be determined before numeric factorization.

Only when the symbolic factorization of column $k$ is finished (i.e., the symbolic structure of column $k$ is known), do we know which columns will update column $k$. However, before factorizing column $k$, we must know the dependency of column $k$ because we need it to schedule the parallel algorithm. This problem can be solved by the concept of ETree [8].

### *B. Elimination Tree*

In SuperLU_MT, ETree is adopted to represent the column-level dependency [8], [48]. It is not an exact representation of the dependency, but contains all potential dependency regardless of the actual pivoting choices (which means it is a toplimit of the dependency), so it overestimates the actual dependency. The definition of ETree and how to calculate the ETree can be found in [8] and [48]; here, we only present an example, as shown in Fig. 6(b), corresponding to the matrix shown in Fig. 6(a). Node $k$ in the ETree corresponds to column $k$ in the matrix, so in the following contents, "node" and "column" are equivalent. A node or a column is also called a task. The ETree can be regarded as a directed acyclic graph (DAG). The edges in the ETree represent the potential dependency. If there is an edge $i \rightarrow j$ in the ETree, then $i$ is a *child* of $j$, and $j$ is the *parent* of $i$, which indicates that column $j$ (potentially) depends on column $i$. The node who has no children is a *leaf node*. For example, in Fig. 6(b), nodes 1, 2, 3, 6, and 7 are leaf nodes, node 8 is a child of node 9, and node 9 is the parent of node 8.

### *C. Dynamic Scheduling Versus Static Scheduling*

Actually, the parallel tasks can be directly scheduled by the ETree, and SuperLU_MT just does it. It sets a flag to each task, the flag is one of "unready," "ready," and "busy," "done" [8], [48]. Each thread finds a ready task (a ready task is the task whose children are all finished), sets its flag to "busy," and executes it. When the task is finished, the thread sets its flag to "done," searches the unready tasks that are now ready, and changes their flags. This is a dynamic scheduling method, which has a critical section (a critical section is a piece of code that accesses a shared resource that must not be concurrently accessed by more than one thread of execution [51]). When one thread is searching a ready task to execute, it must use a mutex to lock other threads to prevent other threads from fetching the same task.

In SuperLU and SuperLU_MT, supernodes are utilized to enhance the capability when computing dense blocks [8], [9], [48]; therefore, the scheduling overheads can be ignored compared with the computational time of a dense block. However, since the circuit matrices are extremely sparse, and so are their LU factors, the computational time of one task is so little that the dynamic scheduling overheads of one node may be larger than its computational time. In our solver, we attempt to reduce the scheduling time as much as possible. So we use static scheduling instead of the dynamic scheduling method.

### *D. Elimination Scheduler*

To perform static scheduling, each node is assigned to a specified thread before factorization. However, the ETree is insufficient to perform static scheduling. The nodes that can be assigned to one thread must have something in common. To explore the commonality from the ETree to assign the nodes, we define level of each node in the ETree.

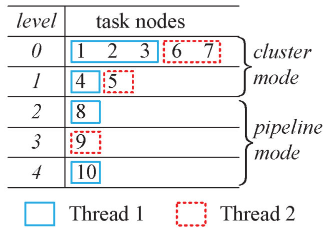



Fig. 7. Scheduling results using two threads.

*Level definition:* Given the ETree, the level of each node is the maximum length among all the paths that connect the leaf nodes and the node itself.

It is simple to calculate level of each node by visiting all the nodes in a topological order

$$\text{level}(k) = \max(-1, \text{level}(c_1), \text{level}(c_2), \cdots) + 1 \quad (1)$$

where $c_1, c_2, \ldots$ are the children of node $k$.

Based on the definition of level, we categorize all the $n$ nodes into different levels, and then we obtain the EScheduler.

*EScheduler definition:* EScheduler is a table $T(V, LV)$ in which $V = \{1, 2, \ldots, n\}$ corresponds to the nodes in the ETree, and $LV = \{\text{level}(k)|1 \leq k \leq n\}$.

Fig. 6(d) shows an example of the EScheduler, corresponding to the ETree in Fig. 6(b). The above definitions indicate that the nodes in the same level are completely independent; this is the commonality of these nodes. Although the EScheduler is coarser grained than the ETree, it is sufficient to implement the column-level parallelism and reduce the scheduling overheads.

### *E. EScheduler-Guided Parallel LU Factorization*

In this paper, we still use cluster mode and pipeline mode to perform the parallel LU factorization. However, the algorithm is different from [6].

We use Fig. 6(d) as an example to illustrate how the tasks are scheduled by the EScheduler. Since the nodes in the same level are completely independent, these nodes can be calculated in parallel effectively, with little scheduling time. We call it cluster mode. However, when the nodes in one level become fewer, cluster mode is ineffective to achieve good parallelism. In this case, the pipeline mode, which exploits parallelism between dependent levels, will be used.

We set a threshold $N_{\text{th}}$ to differentiate cluster mode levels from pipeline mode levels, which is defined as the number of nodes in one level, then the EScheduler can be divided into two parts, the former belongs to the cluster mode and the latter belongs to the pipeline mode. $N_{\text{th}} = 1$ is adopted in this example, so levels $0 \sim 1$ are factorized by cluster mode, and then the rest of the nodes are factorized in parallel by pipeline mode. In practice, we find that $N_{\text{th}} = 5 \sim 20$ is the optimal value for the cases with less than 12 threads. The threshold is a experiential value and tunable in our solver. Generally, $N_{\text{th}}$ should be greater or equal to the number of threads, otherwise some threads may have no tasks in cluster mode.

1) *Cluster Mode:* All the levels belonging to cluster mode are processed level by level. For each level, nodes are allocated to different threads (nodes assigned to one thread are regarded as a cluster), and the load balance is achieved by equalizing the number of nodes in each cluster. Each thread performs the same code (i.e., the left-looking G–P algorithm) to factorize the nodes that are assigned to it. Node-level synchronization is not needed since the nodes in the same level are independent, which reduces bulk of the synchronization time. We wait for all the threads finish factorizing the current level, and then the nodes in the next level will be processed by the same strategy. So cluster mode is a level-synchronization algorithm. Fig. 7 shows an example of node allocation to two threads in the cluster mode.

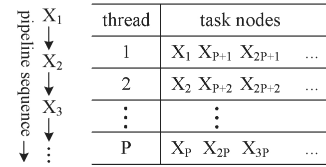



Fig. 8. Static scheduling method in pipeline mode parallelism.

**Algorithm 3** Pipeline mode parallel algorithm for each thread

```
while the pipeline tail is not achieved do
  Get a new unfactorized column k;
  //pre-factorization:
  while the previous column in the pipeline sequence is not finished do
    Clear S;
    Symbolic factorization;
      •determine which columns will update column k
      •skip all the unfinished columns
      •put the newly detected columns that column k depends on into S
    Numeric factorization;
      •use the columns in S to update column k
  end while
  //post-factorization:
  Clear S;
  Symbolic factorization;
    •determine the exact nonzero structure of column k
    •determine which columns will update column k
    •without skipping any columns
    •put the newly detected columns that column k depends on into S
  Numeric factorization;
    •use the columns in S to update column k
  Partial pivoting;
  Store factorization results into column k of L and U;
end while
```

2) *Pipeline Mode:* In pipeline mode, the dependency is much stronger since the nodes are usually located in different levels. Although we also call it pipeline mode, it is different from the pipeline mode in [6].

We first sort all the nodes belonging to the pipeline mode into a topological sequence (in the above example, the topological sequence is 8, 9, 10), and then perform a static scheduling, which is illustrated in Fig. 8. The task nodes of each thread are completely assigned before the pipeline mode starts, which also reduces the scheduling time.

In the pipeline mode, each thread also executes the same code, as shown in Algorithm 3. The pseudocode can be partitioned into two parts: prefactorization and postfactorization. In both parts, a set $S$ is maintained to keep all the newly detected columns that are found in the latest symbolic factorization.

In prefactorization, the thread (factorizing node $k$) performs a while loop, which will not end until the previous column in the pipeline sequence is finished. During the while loop, the symbolic factorization and numeric factorization are repeatedly executed. Symbolic factorization is to determine that columns will update column $k$. However, currently some dependent columns are not finished, they will be skipped. (So it is an incomplete symbolic factorization. If some columns are still unfinished in the later symbolic factorization, they will be skipped again.) The newly detected columns that column $k$ depends on are put into $S$. It then uses the columns in $S$ to perform numeric factorization on column $k$. These columns are marked as used and they will not be put into $S$ in the later symbolic factorizations when factorizing column $k$. The code of numeric accumulation is in Algorithm 2; the difference is that currently it uses only part of the dependent columns (i.e., the columns in $S$) to update column $k$. The prefactorization will be repeated until the previous column in the pipeline sequence is finished, and then it performs the postfactorization.

In postfactorization, the thread performs a complete symbolic factorization without skipping any columns (all the dependent columns are finished currently), to determine the exact nonzero structure of column $k$ and to detect the columns that column $k$ depends on and are not used in prefactorization. Finally, it uses these newly detected columns to do the numeric update on column $k$.

The pipeline mode exploits parallelism by prefactorization. In the sequential algorithm, one column ($k$), starts strictly after the previous column is finished. However, in the pipeline mode parallelism, before the previous column finishes, column $k$ has already performed some numeric update by some dependent, finished columns.

### *F. Discussion on Parallel Right-Hand Solving*

Although the computational cost of the right-hand-solving step is relatively small, it will affect scalability of the solver when the runtime of numeric factorization is significantly reduced. The parallel strategy of right-hand solving is very similar to the parallel factorization algorithm in [6]. The structure of $L(U)$ determines the dependency of forward(back) substitution, forward(back) substitution can be also parallelized by cluster mode and pipeline mode. However, circuit matrices are extremely sparse and the right-hand-solving step has many fewer FLOPs than the LU factorization step, leading to a high communication/synchronization to computation ratio; so parallelization of right-hand solving may not be efficient.

## VI. Experimental Results and Discussion

### *A. Experimental Setup*

NICSLU is written in C. The experiments are implemented on a 64-b Linux server with 2 Xeon5670 CPUs (two sockets, six cores per socket) and 24 GB memory. Thirty-five matrices from the University of Florida Sparse Matrix Collection [52] are used to evaluate NICSLU. We also test KLU [5] as a baseline for comparison, which is implemented with its default configurations. Both NICSLU and KLU are compiled using gcc with the optimization flag -O2. Two types of speedups are defined as follows (runtime is the factorization time, in seconds):

$$\text{speedup} = \frac{\text{runtime of KLU}}{\text{runtime of NICSLU}} \tag{2}$$

$$\text{relative speedup} = \frac{\text{runtime of NICSLU(sequential)}}{\text{runtime of NICSLU(parallel)}}. \tag{3}$$

The basic information of the benchmarks is shown in Table I. Most of the matrices are from differential algebraic equations (DAEs), except for matrices onetone1, onetone2, twotone, and pre2 that are from frequency domain simulation; mac_econ_fwd500 and mc2depi are noncircuit matrices but have similar sparsity with circuit matrices. The number of benchmarks we used is up to 5.55 million, which is larger than the available test results of other software implementations [5], [8]–[13]. NICSLU can be scaled to bigger problem sizes. An important feature of circuit matrices is that they are extremely sparse. The row density ($\frac{\text{NNZ}(A)}{n}$) is less than 10 for most of the benchmarks, regardless of the matrix dimension.

TABLE I
Test Benchmarks

| ID | Name | Dimension($n$) | NNZ($A$) | Row Density |
|---|---|---|---|---|
| 1 | rajat03 | 7602 | 32 653 | 4.30 |
| 2 | coupled | 11 341 | 98 523 | 8.69 |
| 3 | ckt11752_dc_1 | 49 702 | 333 029 | 6.70 |
| 4 | ASIC_100ks | 99 190 | 578 890 | 5.84 |
| 5 | ASIC_100k | 99 340 | 954 163 | 9.61 |
| 6 | G2_circuit | 150 102 | 726 674 | 4.84 |
| 7 | transient | 178 866 | 961 790 | 5.38 |
| 8 | Raj1 | 263 743 | 1 302 464 | 4.94 |
| 9 | ASIC_320ks | 321 671 | 1 827 807 | 5.68 |
| 10 | ASIC_320k | 321 821 | 2 635 364 | 8.19 |
| 11 | rajat30 | 643 994 | 6 175 377 | 9.59 |
| 12 | ASIC_680ks | 682 712 | 2 329 176 | 3.41 |
| 13 | ASIC_680k | 682 862 | 3 871 773 | 5.67 |
| 14 | Hamrle3 | 1 447 360 | 5 514 242 | 3.81 |
| 15 | G3_circuit | 1 585 478 | 7 660 826 | 4.83 |
| 16 | memchip | 2 707 524 | 14 810 202 | 5.47 |
| 17 | Freescale1 | 3 428 755 | 18 920 347 | 5.52 |
| 18 | circuit5M_dc | 3 523 317 | 19 194 193 | 5.45 |
| 19 | rajat31 | 4 690 002 | 20 316 253 | 4.33 |
| 20 | onetone1 | 36 057 | 341 088 | 9.46 |
| 21 | onetone2 | 36 057 | 227 628 | 6.31 |
| 22 | twotone | 120 750 | 1 224 224 | 10.14 |
| 23 | pre2 | 659 033 | 5 959 282 | 9.04 |
| 24 | mac_econ_fwd500 | 206 500 | 1 273 389 | 6.17 |
| 25 | mc2depi | 525 825 | 2 100 225 | 3.99 |
| 26 | add20 | 2395 | 17 319 | 7.23 |
| 27 | circuit_1 | 2624 | 35 823 | 13.65 |
| 28 | circuit_2 | 4510 | 21 199 | 4.70 |
| 29 | add32 | 4960 | 23 884 | 4.82 |
| 30 | circuit_3 | 12 127 | 48 137 | 3.97 |
| 31 | circuit_4 | 80 209 | 307 604 | 3.84 |
| 32 | hcircuit | 105 676 | 513 072 | 4.86 |
| 33 | dc1 | 116 835 | 766 396 | 6.56 |
| 34 | trans4 | 116 835 | 766 396 | 6.56 |
| 35 | circuit5M | 5 558 326 | 59 524 291 | 10.71 |

Row density = $\frac{\text{NNZ}(A)}{n}$.

TABLE II
FACTORIZATION TIME AND THE SPEEDUPS OVER KLU (WITH BTF)

| Matrix Benchmark | KLU (With BTF) | NICSLU | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 Core | | 4 Cores | | 8 Cores | | 12 Cores | |
| | Runtime | Runtime | Speedup | Runtime | Speedup | Runtime | Speedup | Runtime | Speedup |
| Set 1A Benchmarks | | | | | | | | | |
| rajat03 | 0.024 | 0.023 | 1.02 | 0.012 | 1.99 | 0.013 | 1.89 | 0.015 | 1.62 |
| coupled | 0.074 | 0.074 | 1.00 | 0.027 | 2.67 | 0.026 | 2.80 | 0.026 | 2.79 |
| ckt11752_dc_1 | 0.086 | 0.417 | 0.21 | 0.252 | 0.34 | 0.154 | 0.56 | 0.340 | 0.25 |
| ASIC_100ks | 2.924 | 1.793 | 1.63 | 0.636 | 4.60 | 0.398 | 7.35 | 0.325 | 9.00 |
| ASIC_100k | 2.342 | 1.501 | 1.56 | 0.629 | 3.72 | 0.433 | 5.41 | 0.374 | 6.26 |
| G2_circuit | 13.153 | 12.989 | 1.01 | 3.970 | 3.31 | 2.377 | 5.53 | 2.075 | 6.34 |
| transient | 0.417 | 0.358 | 1.16 | 0.260 | 1.60 | 0.228 | 1.83 | 0.231 | 1.80 |
| Raj1 | 110.815 | 1.108 | 100.01 | 0.641 | 172.93 | 0.561 | 197.68 | 0.595 | 186.13 |
| ASIC_320ks | 31.342 | 31.061 | 1.01 | 8.946 | 3.50 | 4.717 | 6.64 | 4.026 | 7.79 |
| ASIC_320k | 27.632 | 27.433 | 1.01 | 7.585 | 3.64 | 4.202 | 6.58 | 3.283 | 8.42 |
| rajat30 | 27.498 | 6.661 | 4.13 | 2.879 | 9.55 | 2.074 | 13.26 | 2.123 | 12.95 |
| ASIC_680ks | 2.915 | 1.752 | 1.66 | 0.764 | 3.82 | 0.522 | 5.59 | 0.500 | 5.83 |
| ASIC_680k | 2.989 | 2.047 | 1.46 | 1.361 | 2.20 | 1.141 | 2.62 | 1.065 | 2.81 |
| Hamrle3[a] | **Fail** | 1073.550 | – | 392.018 | – | 249.118 | – | 209.354 | – |
| G3_circuit | 760.998 | 747.161 | 1.02 | 229.002 | 3.32 | 146.063 | 5.21 | 127.491 | 5.97 |
| memchip | 462.144 | 456.546 | 1.01 | 140.346 | 3.29 | 87.854 | 5.26 | 76.205 | 6.06 |
| Freescale1 | 16.219 | 15.646 | 1.04 | 6.948 | 2.33 | 4.857 | 3.34 | 4.608 | 3.52 |
| circuit5M_dc | 90.940 | 89.704 | 1.01 | 27.081 | 3.36 | 15.473 | 5.88 | 12.258 | 7.42 |
| rajat31 | 581.478 | 535.335 | 1.09 | 159.805 | 3.64 | 100.917 | 5.76 | 92.599 | 6.28 |
| Set 1B Benchmarks | | | | | | | | | |
| onetone1 | 16.373 | 2.373 | 6.90 | 0.601 | 27.24 | 0.319 | 51.38 | 0.274 | 59.78 |
| onetone2 | 0.620 | 0.265 | 2.34 | 0.116 | 5.35 | 0.081 | 7.71 | 0.071 | 8.78 |
| twotone | 80.717 | 14.069 | 5.74 | 4.832 | 16.70 | 2.962 | 27.25 | 2.368 | 34.08 |
| pre2[a] | **Fail** | 505.565 | – | 159.423 | – | 102.698 | – | 91.953 | – |
| Set 1C Benchmarks | | | | | | | | | |
| mac_econ_fwd500 | 13414.872 | 46.125 | 290.84 | 18.883 | 710.41 | 11.129 | 1205.41 | 8.965 | 1496.39 |
| mc2depi | 85.328 | 84.513 | 1.01 | 23.643 | 3.61 | 12.810 | 6.66 | 9.204 | 9.27 |
| **Geometric mean** | | | **2.08** | | **5.36** | | **7.98** | | **8.57** |
| Set 2 Benchmarks | | | | | | | | | |
| add20 | 0.002 | 0.002 | 1.05 | 0.011 | 0.15 | 0.004 | 0.38 | 0.030 | 0.06 |
| circuit_1 | 0.004 | 0.004 | 1.00 | 0.004 | 1.20 | 0.026 | 0.16 | 0.033 | 0.13 |
| circuit_2 | 0.003 | 0.003 | 1.06 | 0.003 | 1.09 | 0.020 | 0.17 | 0.038 | 0.09 |
| add32 | 0.002 | 0.002 | 0.71 | 0.005 | 0.36 | 0.018 | 0.10 | 0.025 | 0.07 |
| circuit_3 | 0.007 | 0.006 | 1.22 | 0.010 | 0.69 | 0.013 | 0.55 | 0.075 | 0.09 |
| circuit_4 | 0.032 | 0.025 | 1.27 | 0.068 | 0.47 | 0.063 | 0.50 | 0.062 | 0.51 |
| hcircuit | 0.046 | 0.036 | 1.28 | 0.076 | 0.60 | 0.077 | 0.59 | 0.081 | 0.56 |
| dc1 | 0.114 | 0.102 | 1.11 | 0.146 | 0.78 | 0.147 | 0.77 | 0.156 | 0.73 |
| trans4 | 0.118 | 0.108 | 1.09 | 0.148 | 0.79 | 0.144 | 0.82 | 0.154 | 0.77 |
| circuit5M | 4.346 | 3.299 | 1.32 | 6.129 | 0.71 | 6.390 | 0.68 | 6.888 | 0.63 |
| **Geometric mean** | | | **1.10** | | **0.60** | | **0.39** | | **0.30** |

The runtime is shown in seconds.
[a]For Hamrle3 and pre2, if KLU is compiled into 32-b version, it fails because of integer overflow; if it is compiled into 64-b version, it still fails because of insufficient memory.

*Performance profile:* In the following results, some figures are plotted by performance profile [53], which is defined as follows, taking computing time as an example. Assume that we have a solver set $\mathcal{S}$ and a problem set $\mathcal{P}$, $t_{p,s}$ is defined as the runtime to solve the problem $p \in \mathcal{P}$ by solver $s \in \mathcal{S}$. We want to compare the performance on problem $p$ by solver $s$ with the best performance by any solver on this problem, so performance ratio is defined as follows:

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} : s \in \mathcal{S}\}}. \quad (4)$$

If solver $s$ fails to solve problem $p$, $t_{p,s}$ is set to infinite. The performance profile of solver $s$ is defined as follows:

$$P_s(\alpha) = \frac{|\{p \in \mathcal{P} : r_{p,s} \leq \alpha\}|}{|\mathcal{P}|}, \alpha \geq 1 \quad (5)$$

where $|\cdot|$ is the size of the set. $P_s(\alpha)$ is the probability for solver $s$ that $r_{p,s}$ is within a factor $\alpha$ of the best possible ratio.

## B. Experimental Results

Table II shows the results of NICSLU (with MC64) and the comparison with KLU (with BTF). The runtime listed in the table is the factorization time, excluding the runtime of preprocessing and right-hand solving. These two steps are not time consuming; they cost only 4.6% and 0.16%

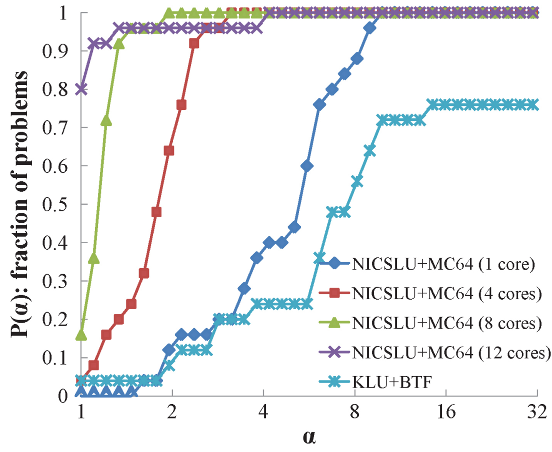



Fig. 9. Performance profile $P(\alpha)$: the factorization time for Set 1 benchmarks.

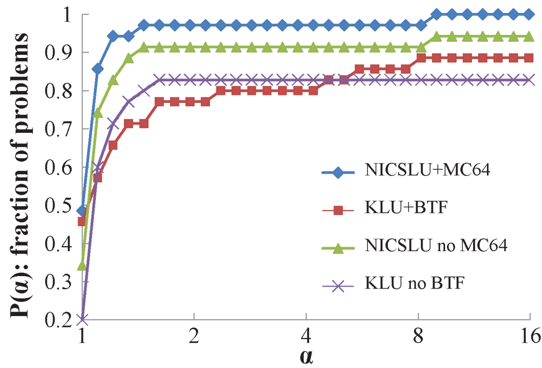



Fig. 10. Performance profile $P(\alpha)$: the number of FLOPs.

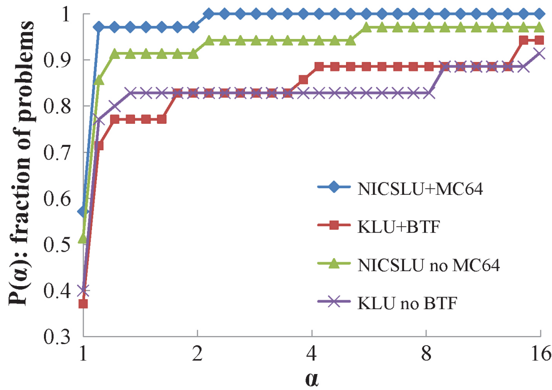



Fig. 11. Performance profile $P(\alpha)$: relative fill-ins ($\frac{\text{NNZ}_F(L+U)}{\text{NNZ}(A)}$).

(on arithmetic mean) of the sequential factorization time, for the 35 test matrices.

We categorize all the benchmarks into two sets, named Set 1 and Set 2, by the proposed predictive method. We suggest using parallel algorithm on Set 1 and sequential algorithm on Set 2, and the results support our suggestion. We will discuss this point in Section VI-C. Set 1 is partitioned into three groups, Set 1A matrices are from DAEs, Set 1B matrices are from frequency simulation, and Set 1C matrices are noncircuit matrices that have similar sparsity with circuit matrices. Our parallel algorithm is generally faster than KLU for Set 1

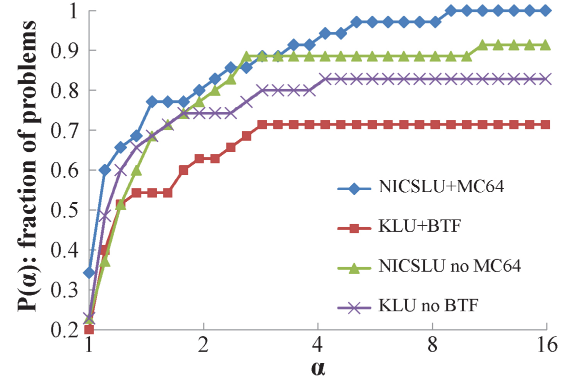



Fig. 12. Performance profile $P(\alpha)$: the residual ($|Ax - b|_1$).

TABLE III
RELATIVE SPEEDUPS AND BENCHMARK CATEGORIZATION

| Matrix Benchmark | Relative Speedup | | | $R_1$ | $R_2$ |
|---|---|---|---|---|---|
| | 4 Cores | 8 Cores | 12 Cores | | |
| Set 1A Benchmarks | | | | | |
| rajat03 | 1.95 | 1.84 | 0.67 | 4.89 | 44 |
| coupled | 2.68 | 2.80 | 2.80 | 3.79 | 66 |
| ckt11752_dc_1 | 1.65 | 2.70 | 1.23 | 3.33 | 37 |
| ASIC_100ks | 2.82 | 4.51 | 5.52 | 6.29 | 383 |
| ASIC_100k | 2.39 | 3.47 | 4.01 | 4.20 | 278 |
| G2_circuit | 3.27 | 5.46 | 6.26 | 27.48 | 503 |
| transient | 1.38 | 1.57 | 1.55 | 2.09 | 113 |
| Raj1 | 1.73 | 1.98 | 1.86 | 5.60 | 99 |
| ASIC_320ks | 3.47 | 6.59 | 7.72 | 2.65 | 283 |
| ASIC_320k | 3.62 | 6.53 | 8.35 | 2.14 | 218 |
| rajat30 | 2.31 | 3.21 | 3.14 | 3.10 | 247 |
| ASIC_680ks | 2.29 | 3.36 | 3.51 | 2.13 | 185 |
| ASIC_680k | 1.50 | 1.79 | 1.92 | 1.70 | 152 |
| Hamrle3 | 2.74 | 4.31 | 5.13 | 43.71 | 1683 |
| G3_circuit | 3.26 | 5.12 | 5.86 | 49.21 | 1591 |
| memchip | 3.25 | 5.20 | 5.99 | 14.79 | 1663 |
| Freescale1 | 2.25 | 3.22 | 3.40 | 3.24 | 192 |
| circuit5M_dc | 3.31 | 5.80 | 7.32 | 4.11 | 343 |
| rajat31 | 3.35 | 5.30 | 5.78 | 17.28 | 1218 |
| Set 1B Benchmarks | | | | | |
| onetone1 | 3.94 | 7.45 | 8.67 | 8.44 | 415 |
| onetone2 | 2.28 | 3.29 | 1.39 | 5.46 | 159 |
| twotone | 2.91 | 4.75 | 5.94 | 9.44 | 949 |
| pre2 | 3.17 | 4.92 | 5.50 | 17.12 | 3812 |
| Set 1C Benchmarks | | | | | |
| mac_econ_fwd500 | 2.44 | 4.14 | 5.15 | 47.54 | 510 |
| mc2depi | 3.57 | 6.60 | 9.18 | 25.88 | 376 |
| **Arithmetic mean** | **2.72** | **4.24** | **4.71** | **12.62** | **621** |
| Set 2 Benchmarks | | | | | |
| add20 | 0.14 | 0.36 | 0.05 | 1.00 | 8 |
| circuit_1 | 1.20 | 0.16 | 0.13 | 1.21 | 21 |
| circuit_2 | 1.03 | 0.16 | 0.09 | 1.54 | 13 |
| add32 | 0.51 | 0.14 | 0.10 | 1.00 | 2 |
| circuit_3 | 0.56 | 0.45 | 0.08 | 1.42 | 4 |
| circuit_4 | 0.37 | 0.40 | 0.40 | 1.40 | 12 |
| hcircuit | 0.47 | 0.46 | 0.44 | 1.23 | 4 |
| dc1 | 0.70 | 0.69 | 0.66 | 1.47 | 32 |
| trans4 | 0.73 | 0.75 | 0.70 | 1.47 | 32 |
| circuit5M | 0.54 | 0.52 | 0.48 | 1.04 | 18 |
| **Arithmetic mean** | **0.63** | **0.41** | **0.31** | **1.28** | **15** |

$R_1 = \frac{\text{NNZ}(L+U)}{\text{NNZ}(A)}$, where $\text{NNZ}(L+U)$ is the number of nonzeros in $L+U$, predicted by static symbolic factorization. $R_2 = \frac{\text{FLOPs}}{\text{NNZ}(L+U)}$, where FLOPs is the number of floating-point operations predicted by static symbolic factorization.

and slower for Set 2. Fig. 9 shows the performance profile of the factorization time, for Set 1 benchmarks. The parallel algorithm achieves speedups of $2.08\times \sim 8.57\times$ (on geometric mean) over KLU, when using $1 \sim 12$ threads, for Set 1 benchmarks. Our sequential algorithm is faster than KLU for both Set 1 and Set 2 benchmarks, except for very few matrices (ckt11752_dc_1 and add32), since the MC64 algorithm leads to fewer off-diagonal pivots. Note that KLU fails on two matrices, the reason is explained in the footnote of Table II.

We also compare the number of FLOPs, relative fill-ins ($\frac{\mathrm{NNZ}_F(L+U)}{\mathrm{NNZ}(A)}$, where $\mathrm{NNZ}_F(L+U)$ is the number of nonzeros in $L+U$ after LU factorization), and residual ($||Ax-b||_1$) between KLU and our solver, as shown in the performance profiles in Figs. 10–12 (please refer to the curves "NICSLU+MC64" and "KLU+BTF" in all the three figures). From the comparison of FLOPs, fill-ins, and residual, NICSLU generally performs better than KLU.

*Impact of the fill-ins:* We get very high speedup over KLU for matrices onetone1, onetone2, twotone, Raj1, mac_econ_fwd500, and rajat30, and low speedup for ckt11752_dc_1. This is because of the preprocessing step: KLU uses BTF, by default; NICSLU does not use BTF but uses MC64. This difference lead to the differences of the matrix structure and pivoting choices, which also leads to the big differences of the fill-ins. Specifically, BTF may dramatically affect matrix structure and the MC64 algorithm can dramatically affect pivot choices. There will be a detailed discussion in Section VI-D.

*Necessity of pivoting:* We have tested the algorithm without partial pivoting for the 35 benchmarks. The results show that the solver fails on the following four matrices: ckt11752_dc_1, rajat30, pre2, and Hamrle3. The reason is that there are some zeros or small elements on the diagonal, leading to interruption or floating-point overflow.

### C. Benchmark Categorization and the Relative Speedups

In this section, we show the relative speedups and discuss the benchmark categorization. For Set 1 benchmarks, our parallel algorithm can always get effective relative speedups but not so for Set 2. We claim that not every matrix is suitable for parallel algorithm, and this is because of the characteristics of the matrix itself.

If the number of FLOPs is small, the parallel algorithm cannot achieve effective relative speedup. That is what we find from the results. As mentioned in Section III, we calculate $R_1 = \frac{\mathrm{NNZ}(L+U)}{\mathrm{NNZ}(A)}$ and $R_2 = \frac{\mathrm{FLOPs}}{\mathrm{NNZ}(L+U)}$, which can be regarded as the relative fill-in and the average number of FLOPs of each nonzero element. We suggest that if ($R_1 < 2.0$ && $R_2 < 50$), the sequential algorithm should be utilized, otherwise the parallel algorithm should be used. Table III shows the $R_1$ and $R_2$ values. For all the Set 2 benchmarks, $R_1$ values are near 1.0 and $R_2$ values are less than 50, which means these matrices only increase a few fill-ins after LU factorization, so the number of FLOPs of these matrices is small. On the contrary, the parallel overheads, such as scheduling time, synchronization time, workload imbalance, memory, and cache conflicts, will dominate in the total runtime.

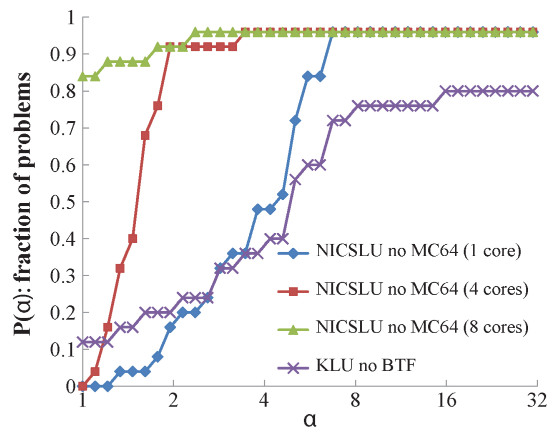



Fig. 13. Performance profile $P(\alpha)$: the factorization time for Set 1 benchmarks, for NICSLU no MC64 and KLU no BTF.

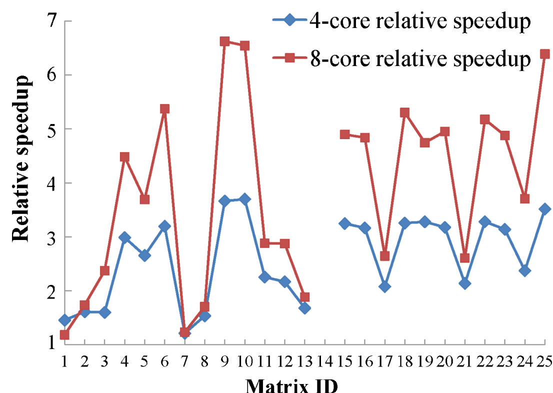



Fig. 14. Relative speedups for Set 1 benchmarks, when disabling MC64 in NICSLU.

We also test SuperLU_MT [8], which is a general-purpose parallel solver by multithreaded parallelism. It is also implemented with its default configurations. The results are shown in Table IV. SuperLU_MT can also achieve speedups for some matrices in Set 1, over KLU, and for these matrices, it also achieves relative speedups over its sequential algorithm. However, the speedups are much smaller then our solver, and for Set 2 benchmarks, neither speedups nor relative speedups can be effectively achieved by SuperLU_MT. The fill-ins of SuperLU_MT is much more then KLU for most of the benchmarks. In addition, SuperLU_MT fails on many benchmarks, caused by memory related problems or software bugs, which indicates that its stability may not be good.

The conclusion is that not every matrix is suitable for the parallel algorithm. For the matrices that have small number of FLOPs, the sequential algorithm should be used rather than the parallel algorithm, then each matrix can achieve the optimal performance. Our proposed predictive method, which performs the static symbolic factorization and calculates $R_1$ and $R_2$, is performed very fast. For the 35 matrices, the runtime of the static symbolic factorization is only 1.2% (on arithmetic mean) of the sequential factorization time, and can be ignored.

### D. Impact of the Preprocessing Algorithms

In Section III, we have mentioned that the preprocessing step of NICSLU is different from KLU, KLU uses BTF without MC64. The above results show that this difference

TABLE IV
RESULTS OF SUPERLU_MT AND THE COMPARISON WITH KLU (WITH BTF)

| Matrix Benchmark | KLU (With BTF) | | SuperLU_MT | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 1 Core | | 4 Cores | | 8 Cores | |
| | Fill-In | Runtime | Fill-In | Runtime | Speedup | Runtime | Speedup | Runtime | Speedup |
| Set 1A Benchmarks | | | | | | | | | |
| rajat03 | **4.79** | 0.024 | 6.63 | 0.038 | 0.63 | 0.015 | 1.60 | **0.011** | 2.18 |
| coupled | **3.68** | **0.074** | 65.54 | 8.804 | 0.01 | 3.208 | 0.02 | 1.808 | 0.04 |
| ckt11752_dc_1 | **3.17** | **0.086** | 46.39 | 12.710 | 0.01 | 3.930 | 0.02 | 2.970 | 0.03 |
| ASIC_100ks | **7.38** | **2.924** | 96.17 | 124.198 | 0.02 | 38.302 | 0.08 | 23.430 | 0.12 |
| ASIC_100k | **4.58** | **2.342** | **Fail** | | | | | | |
| G2_circuit | **27.48** | 13.153 | 51.12 | 20.440 | 0.64 | 5.251 | 2.50 | **3.145** | 4.18 |
| transient | **2.17** | **0.417** | **Fail** | | | | | | |
| Raj1 | **77.55** | **110.815** | | | | **Fail** | | | |
| ASIC_320ks | **2.65** | **31.342** | 38.20 | 854.059 | 0.04 | 290.664 | 0.11 | 166.800 | 0.19 |
| ASIC_320k | **2.10** | **27.632** | **Fail** | | | | | | |
| rajat30 | **5.13** | **27.498** | **Fail** | | | | | | |
| ASIC_680ks | **2.13** | **2.915** | **Fail** | | | | | | |
| ASIC_680k | **1.72** | **2.989** | **Fail** | | | | | | |
| Hamrle3 | **Fail** | | **277.84** | 1047.269 | – | 300.113 | – | **169.448** | – |
| G3_circuit | **49.21** | **760.998** | **Fail** | | | | | | |
| memchip | **14.79** | 462.144 | 23.43 | 94.652 | 4.88 | 27.090 | 17.06 | **17.169** | 26.92 |
| Freescale1 | **3.24** | 16.219 | 8.10 | 44.448 | 0.36 | 12.958 | 1.25 | **8.140** | 1.99 |
| circuit5M_dc | **4.11** | 90.940 | 5.86 | 260.370 | 0.35 | 74.187 | 1.23 | **42.807** | 2.12 |
| rajat31 | **17.97** | **581.478** | **Fail** | | | | | | |
| Set 1B Benchmarks | | | | | | | | | |
| onetone1 | 32.83 | 16.373 | **13.31** | 3.655 | 4.48 | 0.925 | 17.70 | **0.664** | 24.66 |
| onetone2 | 9.36 | 0.620 | **5.09** | 0.183 | 3.39 | 0.084 | 7.38 | **0.052** | 11.92 |
| twotone | 37.08 | 80.717 | **15.83** | 15.976 | 5.05 | 3.879 | 20.81 | **2.306** | 35.00 |
| pre2 | **Fail** | | **56.12** | 720.134 | – | 226.172 | – | **136.702** | – |
| Set 1C Benchmarks | | | | | | | | | |
| mac_econ_fwd500 | 726.41 | 13414.872 | **134.64** | 165.801 | 80.91 | 47.968 | 279.66 | **29.038** | 461.98 |
| mc2depi | **25.88** | 85.328 | 36.61 | 82.607 | 1.03 | 22.999 | 3.71 | **12.975** | 6.58 |
| **Geometric mean** | | | | | **0.53** | | **1.64** | | **2.66** |
| Set 2 Benchmarks | | | | | | | | | |
| add20 | **1.00** | **0.002** | 2.21 | 0.008 | 0.25 | 0.004 | 0.50 | 0.006 | 0.33 |
| circuit_1 | **1.18** | **0.004** | 2.10 | 0.012 | 0.33 | 0.012 | 0.33 | 0.018 | 0.22 |
| circuit_2 | **1.68** | **0.003** | 20.45 | 0.352 | 0.01 | 0.150 | 0.02 | 0.067 | 0.04 |
| add32 | **1.00** | **0.002** | 1.17 | 0.007 | 0.29 | 0.004 | 0.50 | 0.010 | 0.20 |
| circuit_3 | **1.38** | **0.007** | 25.23 | 0.652 | 0.01 | 0.274 | 0.03 | 0.232 | 0.03 |
| circuit_4 | **1.44** | **0.032** | 58.16 | 46.239 | 0.00 | 46.463 | 0.00 | 28.900 | 0.00 |
| hcircuit | **1.22** | **0.046** | 9.94 | 1.985 | 0.02 | 0.732 | 0.06 | 0.498 | 0.09 |
| dc1 | **1.49** | **0.114** | **Fail** | | | | | | |
| trans4 | **1.48** | **0.118** | **Fail** | | | | | | |
| circuit5M | **1.04** | **4.346** | **Fail** | | | | | | |
| **Geometric mean** | | | | | **0.03** | | **0.06** | | **0.06** |

The runtime is shown in seconds.
The best values of fill-ins and runtime are shown in boldface.
The fill-in in this table is relative fill-in $\frac{\mathrm{NNZ}_F(L+U)}{\mathrm{NNZ}(A)}$, where $\mathrm{NNZ}_F(L+U)$ is the number of nonzeros in $L+U$.

may lead to large differences on the fill-ins and FLOPs, because BTF affects the matrix structure, MC64 affects the pivot choices and further affects the fill-ins. In this section, we analyze the impact of the BTF algorithm and the MC64 algorithm. To analyze this point, KLU is implemented without BTF and NICSLU is implemented without MC64, and then NICSLU and KLU have the same preprocessing algorithm (both only have the AMD algorithm).

As can be seen from Table V, the factorization time and the FLOPs/fill-ins are correlative. For ontone1, onetone2, and twotone, NICSLU runs much faster then KLU, and gets much fewer FLOPs/fill-ins than KLU; but for ckt_11752_dc_1, NICSLU is much slower than KLU. These differences are caused by the scaling algorithm in KLU. KLU uses max-scaling or sum-scaling (the default is max-scaling) to scale each column when factorizing [5]. Scaling does not always have positive effect; for ckt_11752_dc_1, scaling can reduce the fill-ins and FLOPs effectively. This result is also reported in [5]. However, for ontone1, onetone2, and twotone, scaling has negative effect. For other matrices, NICSLU and KLU have the similar performance and do not have obvious differences on fill-ins or FLOPs. The performance profiles of FLOPs, fill-ins, and residual are shown in Figs. 10–12 (please refer to the curves " NICSLU no MC64" and " KLU no BTF" in all the three figures). NICSLU generally performs better then KLU when they use the same preprocessing algorithm.

Fig. 13 shows the performance profile of factorization time, and Fig. 14 shows the relative speedups when the MC64

TABLE V
COMPARISON BETWEEN NICSLU (NO MC64) AND KLU (NO BTF)

| Matrix Benchmark | Runtime | | Fill-In | | FLOPs | |
|---|---|---|---|---|---|---|
| | NICSLU | KLU | NICSLU | KLU | NICSLU | KLU |
| Set 1A Benchmarks | | | | | | |
| rajat03 | 0.023 | **0.012** | **4.89** | 5.06 | 6.97E+06 | **6.55E+06** |
| coupled | 0.074 | **0.038** | 3.79 | **3.74** | 2.47E+07 | **2.43E+07** |
| ckt11752_dc_1 | 0.418 | **0.079** | 6.47 | **3.22** | 3.04E+08 | **3.58E+07** |
| ASIC_100ks | **1.909** | 3.186 | **6.29** | 7.38 | **1.39E+09** | 2.19E+09 |
| ASIC_100k | **1.682** | 1.960 | **4.20** | 4.34 | **1.11E+09** | 1.39E+09 |
| G2_circuit | **12.971** | 13.042 | 27.48 | 27.48 | 1.00E+10 | 1.00E+10 |
| transient | **0.347** | 0.371 | **2.09** | 2.17 | **2.27E+08** | 2.60E+08 |
| Raj1 | 1.092 | **0.970** | 5.60 | **5.53** | 7.18E+08 | **6.75E+08** |
| ASIC_320ks | 31.015 | **28.789** | 2.65 | 2.65 | 1.37E+09 | 1.37E+09 |
| ASIC_320k | 27.265 | **20.870** | 2.14 | **2.05** | 1.23E+09 | **1.02E+09** |
| rajat30 | **6.526** | 7.768 | **3.10** | 3.19 | **4.74E+09** | 5.52E+09 |
| ASIC_680ks | 1.883 | **1.747** | 2.13 | 2.13 | 9.18E+08 | 9.18E+08 |
| ASIC_680k | **2.342** | 2.536 | 1.70 | 1.70 | **9.99E+08** | 1.00E+09 |
| Hamrle3 | **Fail** | **Fail** | **Fail** | **Fail** | **Fail** | **Fail** |
| G3_circuit | 754.595 | **751.823** | 49.16 | 49.16 | 5.99E+11 | 5.99E+11 |
| memchip | 458.403 | **458.105** | 14.79 | 14.79 | 3.64E+11 | 3.64E+11 |
| Freescale1 | **15.702** | 15.965 | 3.24 | 3.24 | 1.17E+10 | 1.17E+10 |
| circuit5M_dc | 90.651 | **86.543** | 4.11 | 4.11 | 2.71E+10 | 2.71E+10 |
| rajat31 | **535.318** | 746.797 | **17.26** | 22.41 | **4.27E+11** | 5.93E+11 |
| Set 1B Benchmarks | | | | | | |
| onetone1 | **2.204** | 44.439 | **10.28** | 74.49 | **1.68E+09** | 2.89E+10 |
| onetone2 | **0.269** | 23.858 | **5.46** | 79.11 | **1.98E+08** | 1.55E+10 |
| twotone | **14.526** | 4682.740 | **9.51** | 395.50 | **1.10E+10** | 2.35E+12 |
| pre2 | 675.200 | **Fail** | 21.13 | **Fail** | 5.30E+11 | **Fail** |
| Set 1C Benchmarks | | | | | | |
| mac_econ_fwd500 | **1023.949** | 4370.240 | **279.09** | 446.20 | **7.94E+11** | 2.30E+12 |
| mc2depi | 83.267 | **81.721** | 25.88 | 25.88 | 2.04E+10 | 2.04E+10 |
| Set 2 Benchmarks | | | | | | |
| add20 | 0.002 | **0.001** | 1.00 | 1.00 | 1.31E+05 | 1.31E+05 |
| circuit_1 | 0.004 | **0.003** | 1.21 | **1.18** | 9.09E+05 | **7.57E+05** |
| circuit_2 | 0.003 | 0.003 | 1.54 | **1.50** | 4.16E+05 | **3.63E+05** |
| add32 | 0.002 | **0.001** | 1.00 | 1.00 | 4.87E+04 | 4.87E+04 |
| circuit_3 | 0.006 | **0.005** | 1.42 | **1.41** | 2.46E+05 | **2.35E+05** |
| circuit_4 | **0.024** | 0.035 | **1.40** | 1.53 | **5.36E+06** | 6.76E+06 |
| hcircuit | **0.032** | 0.043 | 1.23 | 1.23 | 2.42E+06 | **2.40E+06** |
| dc1 | **0.098** | 0.107 | 1.47 | 1.47 | 3.59E+07 | 3.59E+07 |
| trans4 | **0.097** | 0.106 | 1.47 | 1.47 | 3.59E+07 | 3.59E+07 |
| circuit5M | **3.253** | 4.077 | 1.04 | 1.04 | 1.10E+09 | **1.06E+09** |

The runtime is shown in seconds.
The best values of runtime, fill-ins, and FLOPs are shown in boldface.
The fill-in in this table is relative fill-in $\frac{\mathrm{NNZ}_F(L+U)}{\mathrm{NNZ}(A)}$, where $\mathrm{NNZ}_F(L+U)$ is the number of nonzeros in $L+U$.

algorithm is disabled, both for Set 1 benchmarks. We get 2.60× and 3.86× (on arithmetic mean) relative speedups for 4 core and 8 core respectively, for Set 1 benchmarks.

As a conclusion of this section, when NICSLU and KLU use the same preprocessing algorithm, NICSLU performs better then KLU for some matrices and worse for only 1 matrix, and for other matrices, the two solvers have the similar performance. The MC64 algorithm has positive effect for some matrices, but not for all, and the same conclusion is for the BTF algorithm in KLU.

## VII. PERFORMANCE MODEL

In SuperLU_MT, a performance model is built to evaluate the upper bound of the speedups [8], [48], which does not consider the number of threads. Inspired by this, we build

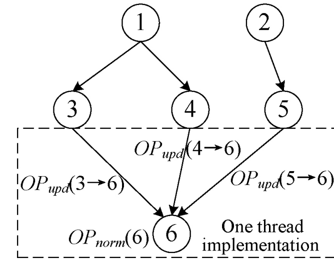



Fig. 15. Example to illustrate the task flow graph and the constraints.

**Algorithm 4** Performance model algorithm

Sort all the nodes in a topological order;
**for** $k = 1 : n$ **do**
  Find $p$ such that thread $p$ has minimum $END(p)$;
  $\mathrm{FT}(k) = \mathrm{END}(p)$;
  **for** each column $i$ that column $k$ depends on **do**
    $\mathrm{FT}(k) = \mathrm{MAX}\{\mathrm{FT}(k), \mathrm{FT}(i)\} + T_{\mathrm{upd}}(i \to k)$;
    $\mathrm{FT}(k) += T_{\mathrm{sync}}$;
  **end for**
  $\mathrm{FT}(k) += T_{\mathrm{norm}}(k)$;
  $\mathrm{END}(p) = \mathrm{FT}(k)$;
**end for**
$\mathrm{FT} = \mathrm{MAX}_{1\le k\le n}\{\mathrm{FT}(k)\}$;

another performance model to predict the speedups, taking into account the number of available threads. Our target is to estimate an upper bound of the relative speedups by the available threads. In our model, only floating-point operations and the essential synchronization overheads are considered. We assume that each floating-point operation takes one unit of runtime, and each synchronization takes $T_{\mathrm{sync}}$ units of time.

All the floating-point operations for factorizing column $k$ can be partitioned into two parts. One part is related to the numeric update on column $k$ from the dependent columns (Algorithm 2). When using column $i$ to update column $k$, we call it $\mathrm{OP}_{\mathrm{upd}}(i \to k)$, which takes $2\mathrm{NNZ}(L(:,i))$ of runtime. The other is related to the normalization of column $k$ in $L$ (the second line from bottom in Algorithm 1), we call it $\mathrm{OP}_{\mathrm{norm}}(k)$, which takes $\mathrm{NNZ}(L(:,k))$ of runtime. Finishing $\mathrm{OP}_{\mathrm{norm}}(k)$ is equivalent to finishing column $k$.

As mentioned above, the structure of $U$ forms a DAG that represents the exact column-level dependency. In the DAG, node $k$ can be treated as $\mathrm{OP}_{\mathrm{norm}}(k)$, and edge $i \to k$ can be treated as $\mathrm{OP}_{\mathrm{upd}}(i \to k)$, then the DAG becomes a task flow graph that describes all the floating-point operations. We use Fig. 15 to illustrate the task flow graph. Take node 6 as an example that has three constraints:

1) $\mathrm{OP}_{\mathrm{upd}}(3 \to 6)$ must be started after $\mathrm{OP}_{\mathrm{norm}}(3)$ is finished, and the same for $\mathrm{OP}_{\mathrm{upd}}(4 \to 6)$ and $\mathrm{OP}_{\mathrm{upd}}(5 \to 6)$;
2) according to the column-level parallelism, the four tasks $\mathrm{OP}_{\mathrm{upd}}(3 \to 6)$, $\mathrm{OP}_{\mathrm{upd}}(4 \to 6)$, $\mathrm{OP}_{\mathrm{upd}}(5 \to 6)$, and $\mathrm{OP}_{\mathrm{norm}}(6)$ are implemented by one thread, so they are implemented sequentially;
3) $\mathrm{OP}_{\mathrm{norm}}(6)$ must be started after $\mathrm{OP}_{\mathrm{upd}}(3 \to 6)$, $\mathrm{OP}_{\mathrm{upd}}(4 \to 6)$, and $\mathrm{OP}_{\mathrm{upd}}(5 \to 6)$ are all finished.

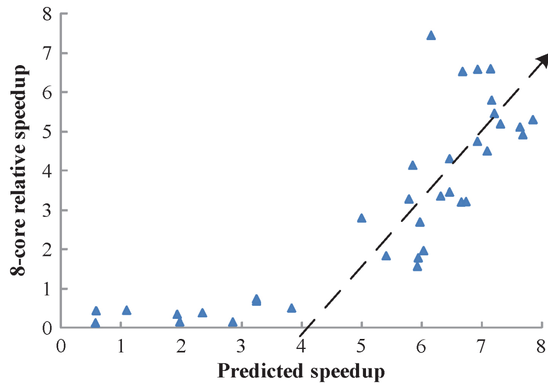



Fig. 16. Relation between the predicted speedups and the actual 8-core relative speedups.

We define the following symbols.

1) $T_{\mathrm{upd}}(i \rightarrow k)$: the runtime of $\mathrm{OP}_{\mathrm{upd}}(i \rightarrow k)$.
2) $T_{\mathrm{norm}}(k)$: the runtime of $\mathrm{OP}_{\mathrm{norm}}(k)$.
3) $\mathrm{FT}(k)$: the possible earliest finish time of $\mathrm{OP}_{\mathrm{norm}}(k)$.
4) FT: the possible earliest finish time of the whole graph.
5) $\mathrm{END}(p)$: the time at which thread $p$ finishes its last task.

The performance model algorithm is shown in Algorithm 4. All the nodes are visited in a topological order. For each node $k$, the thread with the minimum $\mathrm{END}(p)$ is always selected to implement node $k$. In short, the algorithm is to calculate the possible earliest finish time of the task flow graph by the available threads. Finally, the predicted relative speedup is calculated by the following:

$$\text{predicted relative speedup} = \frac{\text{FLOPs}}{\text{FT}} \tag{6}$$

where FLOPs is the total floating-point operations, which can be treated as the finish time of sequential factorization. So the ratio denotes the maximum relative speedup.

Fig. 16 shows the scatter plot of the predicted relative speedups (with eight available threads) and the actual 8-core relative speedups (with MC64), each point represents a matrix benchmark. In this experiment, $T_{\mathrm{sync}} = 30$. There are some differences between the actual relative speedups and the predicted speedups, since it is very hard to capture all the low-level details of a real machine with real scheduling and implementation. However, there is an approximate linear relationship between them, and the correlation coefficient of them is 0.833. So the proposed model can effectively predict the solver performance.

## VIII. Conclusion

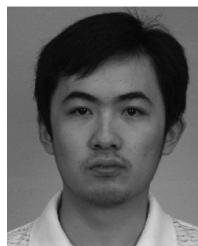
The sparse matrix solver has become the bottleneck in fast SPICE-like simulators. In this paper, we developed NICSLU, a parallel LU factorization (with partial pivoting)-based sparse matrix solver. We found that not every matrix is suitable for parallel factorization, and proposed a simple predictive method to decide whether a matrix should use the sequential or parallel algorithm, so that each matrix can achieve the optimal performance.

The sparse matrix solver was a critical component in circuit simulators; although many parallel approaches have been developed to accelerate circuit simulation, they still need a fast sparse matrix solver. This paper contributed to the development of a parallel solver for circuit matrices, which can be used in all the SPICE-like circuit simulators. Since there is no universal efficient solver/algorithm for general circuit matrices, we suggest that different solvers/algorithms can be adopted, and some predictive methods can be used to select the optimal solver/algorithm before simulating a specific circuit.

## References

[1] L. W. Nagel, "SPICE 2: A computer program to stimulate semiconductor circuits," Ph.D. dissertation, Dept. Electric. Eng. Comput. Sci., Univ. California, Berkeley, 1975.
[2] T. A. Davis, *Direct Methods for Sparse Linear Systems.* Philadelphia, PA: SIAM, 2006.
[3] N. Kapre and A. DeHon, "Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs," in *Proc. Int. Conf. FPT*, Dec. 2009, pp. 190–198.
[4] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," *SIAM J. Sci. Statist. Comput.*, vol. 9, no. 5, pp. 862–874, 1988.
[5] T. A. Davis and E. P. Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*, vol. 37, pp. 36:1–36:17, Sep. 2010.
[6] X. Chen, W. Wu, Y. Wang, H. Yu, and H. Yang, "An EScheduler-based data dependence analysis and task scheduling for parallel circuit simulation," *IEEE Trans. Circuits Syst. II, Express Briefs*, vol. 58, no. 10, pp. 702–706, Oct. 2011.
[7] X. Chen, Y. Wang, and H. Yang, "An adaptive LU factorization algorithm for parallel circuit simulation," in *Proc. 17th ASP-DAC*, Jan.–Feb. 2012, pp. 359–364.
[8] J. W. Demmel, J. R. Gilbert, and X. S. Li, "An asynchronous parallel supernodal algorithm for sparse Gaussian elimination," *SIAM J. Matrix Anal. Applicat.*, vol. 20, no. 4, pp. 915–952, 1999.
[9] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, "A supernodal approach to sparse partial pivoting," *SIAM J. Matrix Anal. Applicat.*, vol. 20, no. 3, pp. 720–755, May 1999.
[10] X. S. Li and J. W. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Trans. Math. Softw.*, vol. 29, no. 2, pp. 110–140, Jun. 2003.
[11] T. A. Davis, "Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method," *ACM Trans. Math. Softw.*, vol. 30, pp. 196–199, Jun. 2004.
[12] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, "Hybrid scheduling for the parallel solution of linear systems," *Parallel Comput.*, vol. 32, no. 2, pp. 136–156, Feb. 2006.
[13] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO," *Future Gener. Comput. Syst.*, vol. 20, no. 3, pp. 475–487, Apr. 2004.
[14] J. W. H. Liu, "The multifrontal method for sparse matrix solution: Theory and practice," *SIAM Rev.*, vol. 34, no. 1, pp. 82–109, 1992.
[15] M. Fischer and H. Dirks, "Multigranular parallel algorithms for solving linear equations in VLSI circuit simulation," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 23, no. 5, pp. 728–736, May 2004.
[16] Z. Li and C.-J. R. Shi, "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.*, vol. 25, no. 12, pp. 2868–2881, Dec. 2006.
[17] A. Meister, *Numerik Linearer Gleichungssysteme: Eine Einführung in Moderne Verfahren*. Wiesbaden, Germany: Vieweg, 2005.
[18] H. Thornquist, E. Keiter, R. Hoekstra, D. Day, and E. Boman, "A parallel preconditioning strategy for efficient transistor-level circuit simulation," in *Proc. IEEE/ACM ICCAD: Dig. Tech. Papers*, Nov. 2009, pp. 410–417.
[19] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Boston, MA: PWS, 2004.
[20] S. Rajamanickam, E. G. Boman, and M. A. Heroux, "ShyLU: A hybrid–hybrid solver for multicore platforms," in *Proc. IEEE 26th IPDPS*, May 2012, pp. 631–643.

[21] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *J. Res. Nat. Bureau Standards*, vol. 49, no. 6, pp. 409–436, 1952.
[22] U. Trottenberg, C. Oosterlee, and A. Schüller, *Multigrid*. San Diego, CA: Academic, 2001.
[23] C.-H. Chou, N.-Y. Tsai, H. Yu, C.-R. Lee, Y. Shi, and S.-C. Chang, "On the preconditioner of conjugate gradient method: A power grid simulation perspective," in *Proc. IEEE/ACM ICCAD*, Nov. 2011, pp. 494–497.
[24] Z. Zeng, T. Xu, Z. Feng, and P. Li, "Fast static analysis of power grids: Algorithms and implementations," in *Proc. IEEE/ACM ICCAD*, Nov. 2011, pp. 488–493.
[25] J. Johnson, T. Chagnon, P. Vachranukunkiet, P. Nagvajara, and C. Nwankpa, "Sparse LU decomposition using FPGA," in *Proc. Int. Workshop State-of-the-Art Scientific Parallel Comput. (PARA)*, 2008, pp. 1–12.
[26] N. Kapre, "SPICE2: A spatial parallel architecture for accelerating the spice circuit simulator," Ph.D. dissertation, Dept. Eng. Appl. Sci., California Inst. Technol., Pasadena, 2010.
[27] T. Nechma, M. Zwolinski, and J. Reeve, "Parallel sparse matrix solver for direct circuit simulations on FPGAs," in *Proc. IEEE ISCAS*, May–Jun. 2010, pp. 2358–2361.
[28] W. Wu, Y. Shan, X. Chen, Y. Wang, and H. Yang, "FPGA accelerated parallel sparse matrix factorization for circuit simulations," in *Proc. Reconfigurable Comput.: Architect., Tools Applicat.*, vol. 6578. 2011, pp. 302–315.
[29] R. Poore, "GPU-accelerated time-domain circuit simulation," in *Proc. IEEE CICC*, Sep. 2009, pp. 629–632.
[30] K. Gulati, J. Croix, S. Khatri, and R. Shastry, "Fast circuit simulation on graphics processing units," in *Proc. ASP-DAC*, Jan. 2009, pp. 403–408.
[31] Z. Feng, Z. Zeng, and P. Li, "Parallel on-chip power distribution network analysis on multi-core-multi-GPU platforms," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 19, no. 10, pp. 1823–1836, Oct. 2011.
[32] N. Frohlich, B. Riess, U. Wever, and Q. Zheng, "A new approach for parallel simulation of VLSI circuits on a transistor level," *IEEE Trans. Circuits Syst. I: Fundamental Theory Applicat.*, vol. 45, no. 6, pp. 601–613, Jun. 1998.
[33] Q. Zhou, K. Sun, K. Mohanram, and D. Sorensen, "Large power grid analysis using domain decomposition," in *Proc. DATE*, vol. 1. Mar. 2006, pp. 1–6.
[34] H. Peng and C.-K. Cheng, "Parallel transistor level circuit simulation using domain decomposition methods," in *Proc. ASP-DAC*, Jan. 2009, pp. 397–402.
[35] K. Chan, "Parallel algorithms for direct solution of large sparse power system matrix equations," *IEE Proc.: Generat., Transmission Distribut.*, vol. 148, no. 6, pp. 615–622, Nov. 2001.
[36] M. Zhao, R. Panda, S. Sapatnekar, and D. Blaauw, "Hierarchical analysis of power distribution networks," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 21, no. 2, pp. 159–168, Feb. 2002.
[37] N. Rabbat, A. Sangiovanni-Vincentelli, and H. Hsieh, "A multilevel Newton algorithm with macromodeling and latency for the analysis of large-scale nonlinear circuits in the time domain," *IEEE Trans. Circuits Syst.*, vol. 26, no. 9, pp. 733–741, Sep. 1979.
[38] M. Honkala, J. Roos, and M. Valtonen, "New multilevel Newton–Raphson method for parallel circuit simulation," in *Proc. ECCTD*, 2001, pp. 113–116.
[39] Z. Zhu, H. Peng, C.-K. Cheng, K. Rouz, M. Borah, and E. Kuh, "Two-stage Newton–Raphson method for transistor-level simulation," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 26, no. 5, pp. 881–895, May 2007.
[40] E. Lelarasmee, A. Ruehli, and A. Sangiovanni-Vincentelli, "The waveform relaxation method for time-domain analysis of large scale integrated circuits," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 1, no. 3, pp. 131–145, Jul. 1982.
[41] W. Dong, P. Li, and X. Ye, "Wavepipe: Parallel transient simulation of analog and digital circuits on multi-core shared-memory machines," in *Proc. 45th ACM/IEEE DAC*, Jun. 2008, pp. 238–243.
[42] X. Ye, W. Dong, P. Li, and S. Nassif, "MAPS: Multi-algorithm parallel circuit simulation," in *Proc. IEEE/ACM ICCAD*, Nov. 2008, pp. 73–78.
[43] X. Ye, W. Dong, P. Li, and S. Nassif, "Hierarchical multialgorithm parallel circuit simulation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 1, pp. 45–58, Jan. 2011.
[44] I. S. Duff and J. Koster, "The design and use of algorithms for permuting large entries to the diagonal of sparse matrices," *SIAM J. Matrix Anal. Appl.*, vol. 20, no. 4, pp. 889–901, Jul. 1999.
[45] I. S. Duff and J. Koster, "On algorithms for permuting large entries to the diagonal of a sparse matrix," *SIAM J. Matrix Anal. Appl.*, vol. 22, no. 4, pp. 973–996, Jul. 2000.
[46] HSL Team. (2011). *The HSL Mathematical Software Library* [Online]. Available: http://www.hsl.rl.ac.uk
[47] P. R. Amestoy, T. A. Davis, and I. S. Duff, "Algorithm 837: AMD, an approximate minimum degree ordering algorithm," *ACM Trans. Math. Softw.*, vol. 30, no. 3, pp. 381–388, Sep. 2004.
[48] X. S. Li, "Sparse Gaussian elimination on high performance computers," Ph.D. dissertation, Comput. Sci. Division, Univ. California, Berkeley, Sep. 1996.
[49] J. R. Gilbert, "Predicting structure in sparse matrix computations," *SIAM J. Matrix Anal. Appl.*, vol. 15, pp. 62–79, Jan. 1994.
[50] I. S. Duff and J. K. Reid, "Algorithm 529: Permutations to block triangular form," *ACM Trans. Math. Softw.*, vol. 4, no. 2, pp. 189–192, Jun. 1978.
[51] Wikipedia. (2012). *Critical Section* [Online]. Available: http://en.wikipedia.org/wiki/Critical_section
[52] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.
[53] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Math. Softw.*, vol. 91, no. 2, pp. 201–213, 2002.



**Xiaoming Chen** (S'12) received the B.S. degree from the Department of Electronic Engineering, Tsinghua University, Beijing, China, in 2009, where he is currently pursuing the Ph.D. degree.

His current research interests include power and reliability-aware circuit design methodology, parallel circuit simulation, and high-performance computing on graphics processing unit.

Mr. Chen was nominated for the Best Paper Award in ISLPED 2009 and ASPDAC 2012.

**Yu Wang** (S'05–M'07) received the B.S. degree in 2002 and the Ph.D. degree (with Hons.) in 2007 from Tsinghua University, Beijing, China.

He is currently an Associate Professor with the Department of Electrical Engineering, Tsinghua University. His current research interests include parallel circuit analysis, low-power and reliability-aware system design methodology, and application-specific hardware computing, especially for brain-related topics. He has authored and co-authored over 70 papers in refereed journals and conference proceedings.

Dr. Wang was a recipient of the Best Paper Award in ISVLSI 2012. He was nominated five times for the Best Paper Award (ISLPED 2009, CODES 2009, twice in ASPDAC 2010, and ASPDAC 2012). He is the Technical Program Committee Co-Chair of ICFPT 2011 and the Publicity Co-Chair of ISLPED 2011.

**Huazhong Yang** (M'97–SM'00) was born in Ziyang, Sichuan Province, China, on August 18, 1967. He received the B.S. degree in microelectronics and the M.S. and Ph.D. degrees in electronic engineering from Tsinghua University, Beijing, China, in 1989, 1993, and 1998, respectively.

In 1993, he joined the Department of Electronic Engineering, Tsinghua University, where he is currently a Specially Appointed Professor of the Cheung Kong Scholars Program. He has authored and co-authored over 200 technical papers and holds 70 granted patents. His current research interests include wireless sensor networks, data converters, parallel circuit simulation algorithms, nonvolatile processors, and energy-harvesting circuits.