An *EScheduler*-Based Data Dependence Analysis and Task Scheduling for Parallel Circuit Simulation

Xiaoming Chen, Wei Wu, Yu Wang, *Member, IEEE*, Hao Yu, *Member, IEEE*, and Huazhong Yang, *Senior Member, IEEE*

***Abstract*—The sparse matrix solver has become the bottleneck in a Simulation Program with Integrated Circuit Emphasis circuit simulator. It is difficult to parallelize the sparse matrix solver because of the high data dependence during the numerical LU factorization. In this brief, a parallel LU factorization algorithm is developed on shared-memory computers with multicore central processing units, based on KLU algorithms. An *Elimination Scheduler (EScheduler)* is proposed to represent the data dependence during the LU factorization. Based on the *EScheduler*, the parallel tasks are scheduled in two modes to achieve a high level of concurrence, i.e., *cluster mode* and *pipeline mode*. The experimental results on 26 circuit matrices reveal that the developed algorithm can achieve speedup of 1.18–4.55× (on geometric average), as compared with KLU, with 1–8 threads. The result analysis indicates that for different data dependence, different parallel strategies should be dynamically selected to obtain optimal performance.**

***Index Terms*—Circuit simulation, Elimination Scheduler (*EScheduler*), parallel LU factorization.**

I. Introduction

THE Simulation Program with Integrated Circuit Emphasis (SPICE) [1] circuit simulator developed by the University of California, Berkeley is widely deployed for verifications of integrated circuits (ICs). With the growing complexity of the very large scale integration at nanoscale, the sparse matrix solver ($A\vec{x} = \vec{b}$) has become the bottleneck [2], since it is repeated in every Newton–Raphson iteration during the transient analysis, and the circuit matrices of postlayout simulation can even reach a dimension of millions. The traditional SPICE circuit simulator has become inefficient in providing accurate verifications for IC designs. At the same time, the recent advance in the underlying hardware by multicore CPUs has unleashed the power and possibility for rethinking of the circuit simulation algorithms. Consequently, there is an emerging need to develop efficient parallel algorithms inside the SPICE engine in order to reduce the design cost and design cycle, particularly during postlayout verifications. Thereby, the acceleration for solving $A\vec{x} = \vec{b}$ has become an interest for developing fast SPICE-like simulators.

There are two categories of methods to solve $A\vec{x} = \vec{b}$, i.e., iterative methods [3] and direct methods [4]. As iterative approach requires precondition in each iteration during the Newton–Raphson iterations, its advantage is unclear for circuit simulation. The direct method by sparse LU factorization is still widely used in modern circuit simulators. One typical LU factorization has three steps, i.e., 1) preordering/symbolic factorization; 2) numerical factorization; and 3) right-hand solving. Among these, the first step performs column/row permutations to increase the stability and reduce fill-ins; the second step performs numerical factor operations for lower triangular matrix L and upper triangular matrix U (i.e., $A = LU$); and the last step solves the triangular equations $L\vec{y} = \vec{b}$ and $U\vec{x} = \vec{y}$. In the SPICE simulation flow, although the entry values of the matrix vary during the iterations, the nonzero pattern remains unchanged. Consequently, the preordering/symbolic factorization can be performed only once; the bottleneck becomes the numeric factorization step.

In this brief, in order to accelerate circuit simulation, a **column-level** parallel sparse LU factorization algorithm is developed on multicore CPUs. The more fine-grained parallelism is suitable for a field-programmable gate array (FPGA) but not a CPU. The contributions of this brief can be summarized as follows.

- The concept of *EScheduler* is introduced to represent data dependence. Compared with *Elimination Tree (ETree)* used in SuperLU_MT [5], our approach can represent data dependence in a more convenient fashion.
- Two parallel task scheduling methods are introduced, i.e., *cluster* and *pipeline modes*. The two modes are dynamically selected during the numeric LU factorization, according to the different structures of the *EScheduler*. Therefore, the parallel tasks in numerical factorization can be effectively scheduled.

The rest of this brief is organized as follow. We review the related work in Section II. The proposed parallel LU factorization algorithm is illustrated in Section III. The experimental results and analysis are presented in Section IV. Finally, Section V concludes this brief.

II. Related Work

Research on LU factorization has been active for decades, and there are some popular software implementations. SuperLU package includes three versions, i.e., the sequential version [6], the multithread version SuperLU_MT [5], and SuperLU_DIST

Manuscript received January 19, 2011; revised April 5, 2011 and June 19, 2011; accepted July 15, 2011. Date of publication September 15, 2011; date of current version October 19, 2011. This work was supported in part by the National Key Technological Program of China under Grant 2008ZX01035-001 and Grant 2010ZX01030-001 and in part by the National Natural Science Foundation of China under Grant 60870001. This paper was recommended by Associate Editor K. Chakrabarty.

X. Chen, W. Wu, Y. Wang, and H. Yang are with 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).

H. Yu is with the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore 639798.

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/TCSII.2011.2164148

1549-7747/$26.00 © 2011 IEEE

```
Algorithm: Left-looking algorithm
1   L = I;
2   for k = 1:N do
3       Solve Lx = b, where b=A(:, k), the kth column of A;
4       U(1:k, k) = x(1:k);
5       L(k:N, k) = x(k:N) / U(k, k);
6   end for
```

(a)

```
Algorithm: Solving Lx=b, where b=A(:, k), the kth column of A
1   x = b;
2   for j = 1:k-1 where U(j, k)!=0 do
3       x(j+1:n) = x(j+1:n) - L(j+1:n, j)*x(j);
4   end for
```

(b)

Fig. 1. Left-looking G–P algorithm-based numeric factorization without partial pivoting.

[7] for distributed memory. SuperLU incorporates *Supernode* in Gilbert–Peierls (G–P) left-looking algorithm [8] to enhance its capability when computing dense blocks. However, because of the high sparsity of circuit matrices, it is hard to form supernodes in circuit simulation. Therefore, in KLU [9], *block triangular form (BTF)* is adopted directly based on G–P left-looking algorithm without *Supernode*. UMFPACK [10], which is integrated in MATLAB, and MUMPS [11] are based on a multifrontal algorithm [12]. In PARDISO [13], the left–right-looking algorithm [14] is developed.

Recently, several approaches have been developed on reconfigurable devices [2], [15], [16]. The scalability to large-scale circuits is still limited by FPGA on-chip resources.

Among all the software implementations of LU factorization, the parallel versions do not dominate; the primary challenge is lack of efficient data dependence analysis and parallel strategies. In this brief, we develop an *EScheduler*-based data dependence analysis; furthermore, different parallel patterns are studied according to the *EScheduler* to reduce the parallel overheads. To evaluate the performance on circuit matrices, our algorithm is compared with KLU (optimized for circuit simulation problems) [9] and SuperLU_MT (a general-purpose parallel sparse matrix solver) [5].

III. Parallel LU Factorization Algorithm

A. Sequential LU Factorization Algorithm

The sequential LU factorization for circuit simulation is introduced first. Our preprocessing step consists of three operations, i.e., 1) HSL_MC64 algorithm [17] to ensure numeric stability, 2) approximate minimum degree algorithm [18] to reduce fill-ins, and 3) G–P algorithm-based prefactorization (a complete numeric factorization with partial pivoting) [8] to calculate the symbolic structure of L and U. The preprocessing step is performed only once in circuit simulation; after this, a fixed symbolic structure of the LU factors is obtained. In the numeric factorization (which is repeated for many times in the iterations in circuit simulation), the left-looking G–P numeric factorization (without partial pivoting) is performed based on the symbolic structure. As shown in Fig. 1(a), the left-looking G–P numeric algorithm (without partial pivoting) factorizes matrix A by sequentially processing each column k in two steps, i.e., 1) solving a lower triangular system $Lx = b$, where $b = A(:, k)$ and 2) storing x in to U and L (with normalization). As shown in Fig. 1(b), the numeric factorization of column k refers to the data in some previous columns $\{j|U(j, k) \neq 0, j < k\}$. In other words, **column k depends on column j, if $U(j, k) \neq 0 (j < k)$**. Given that the symbolic factorization is already performed in the preprocessing step, the column-level data dependence can be extracted from the structure of U.

B. EScheduler Definition

Based on the above discussion, the primary data dependence during sparse LU factorization is the column-level data dependence. In this brief, a **column-level** parallelism is exploited. First, we define an *EScheduler* to represent the column-level data dependence, and then, the parallel tasks are scheduled by the *EScheduler*.

In SuperLU_MT, an *ETree* [5], [19] is constructed from the symmetric matrix $A^T A$ or $A^T + A$ to illustrate the data dependence. However, it introduces much redundant data dependence. In order to describe the data dependence more exactly, an *EScheduler* is proposed in this brief based on the *Elimination Graph (EGraph)*.

EGraph Definition: *Given the structure of U after the symbolic factorization, EGraph is built from U. It is a directed acyclic graph $G_E(V, E)$, where $V = \{1, 2, \ldots, n\}$ (node set) corresponds to all the columns of A and $E = \{(j, k)|U(j, k) \neq 0, k = 1, 2, \ldots, n, j < k\}$ (edge set)*, and $e = (j, k) \in E$ *represents an edge from j to k*. In the following text, "node" and "column" are equivalent. The edges in the *EGraph* represent the column-level data dependence during the numeric LU factorization. To explore some commonness from the *EGraph*, we define the *level* of each node in the *EGraph*.

Level Definition: *Given the EGraph, the level of each node is the length of the longest path from any "source node" to the node itself, if there is a path connecting the source node and itself. "Source node" means the node that has no incoming edges.*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 $S(V, LV)$, in which $V = \{1, 2, \ldots, n\}$ corresponds to the nodes in the EGraph, and $LV = \{\text{level}(k)|1 \leq k \leq n\}$, where* $\text{level}(k)$ *represents the level of a node k.*

The *EScheduler* is directly calculated from the symbolic factorization result of U, i.e.,

$$\text{level}(k) = \max(-1, \text{level}(j_1), \text{level}(j_2), \ldots) + 1 \quad (1)$$

where $j_1, j_2, \ldots$ are row indexes of all off-diagonal nonzeros in column k of U (i.e., $U(j_1, k) \neq 0$ and $U(j_2, k) \neq 0, \ldots$).

The above definitions indicate that the nodes in the same level are independent. From the definitions, although we do not know the exact data dependence from the *EScheduler*, it is sufficient to implement the column-level parallelism on multicore CPUs.

C. EScheduler-Based Parallel Task Scheduling

Take Fig. 2(c) as an example, the *EScheduler* has two primary structures, i.e., in levels 0–1, there are many nodes in each level, whereas in levels 2–4, each level has very few nodes. In the former structure, since the nodes in the same level are

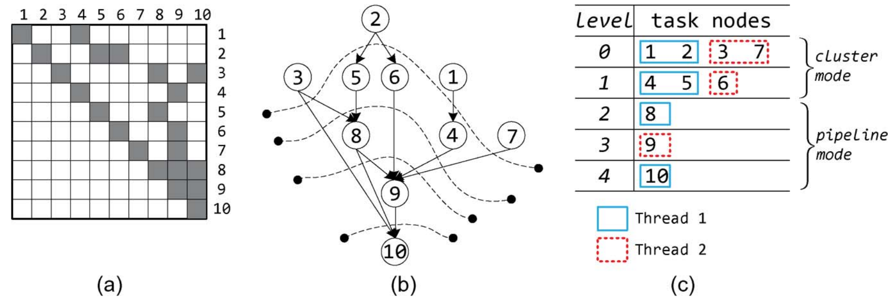

Fig. 2. Example of upper triangular matrix U with its corresponding *EGraph* and *EScheduler*.

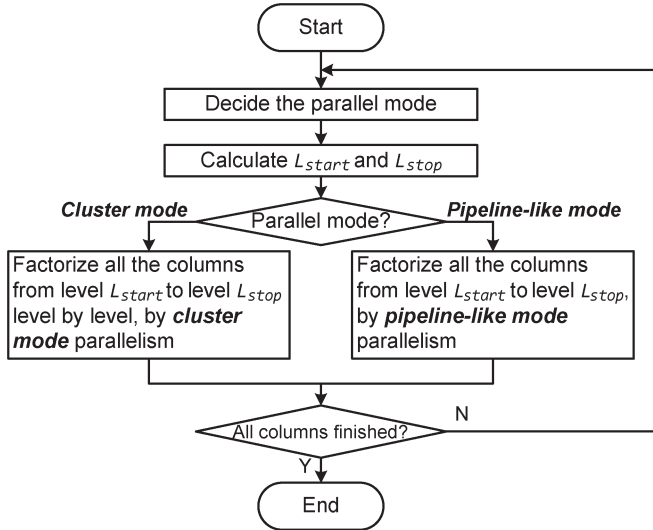

Fig. 3. Overall flow of the *EScheduler*-based parallel task scheduling.

completely independent, they can be factorized in parallel; we call this parallel strategy *cluster mode*. In the latter structure, we use another parallel strategy that **exploits the parallelism between dependent levels** called *pipeline mode*. We set a threshold N_{th}, which is defined as the node number in one level, to differentiate *cluster mode* levels from *pipeline mode* levels. $N_{th} = 1$ is adopted in this example.

Fig. 3 shows the overall flow of the *EScheduler*-based parallel task scheduling, where $L_{\rm start}$ and $L_{\rm stop}$ denote the start and stop levels, respectively, corresponding to the current parallel mode in the *EScheduler*. For the above example, first, the parallel mode is set to the *cluster mode* since there are four nodes in level 0, and then, we get $L_{\rm start} = 0$ and $L_{\rm stop} = 1$. All the columns in levels 0–1 are factorized level by level through the *cluster mode*. In the next iteration, the parallel mode is switched to the *pipeline mode,* and then, $L_{\rm start} = 2$ and $L_{\rm stop} = 4$. All the columns in levels 2–4 are factorized in parallel through the *pipeline mode*. After two iterations, all the columns are factorized, and the task scheduling flow ends.

1) Parallelism in the Cluster Mode: As aforementioned, the nodes in the same level are independent. In the *cluster mode*, all the levels in $[L_{\rm start}, L_{\rm stop}]$ will be processed level by level. In each level L, 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 node number in each *cluster*. Node-level synchronization is not needed since the nodes in one level are independent, which reduces bulk of the synchronization time. We wait for all the threads to finish

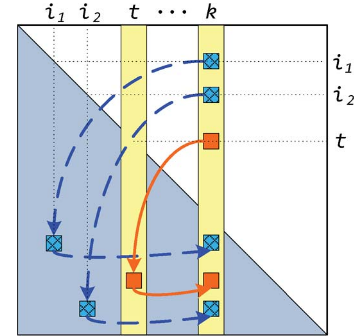

Fig. 4. *Pipeline mode* parallelism.

factorizing the nodes in level L, and then, nodes in the next level will be processed by the same approach. Fig. 2(c) shows an example of node allocation to two threads in the *cluster mode*.

2) Parallelism in the Pipeline Mode: In the *pipeline mode*, each level has very few nodes; therefore, the dependence among nodes is much stronger since these nodes are usually located in different levels. However, even for dependent columns, we can still factorize them in parallel.

Take the case in Fig. 4 for example, currently two threads are factorizing columns t and k $(t < k)$ **simultaneously**. Column k depends on column t and some other columns $i_1, i_2, \ldots$. Assume that columns $i_1, i_2, \ldots$ have been already factorized in the *cluster mode* or the previous *pipeline mode*. While processing column k, column k can be first updated by the finished columns $i_1, i_2, \ldots$ (corresponding to the dotted line in Fig. 4). When it needs to touch column t, it will wait until column t is finished (if currently column t is already finished, then no wait happens); after this, column k is updated by column t, corresponding to the solid line (at this moment, the thread that just factorized column t is now factorizing another unfinished column, which is similar to a *pipeline*). To make a general definition of the *pipeline mode*, **during the factorization of the kth column, we use the off-diagonal nonzeros in $U(:, k)$ to synchronize the factorization. In other words, the *EGraph* is used to synchronize the *pipeline mode* parallelism**. Therefore, the *pipeline mode* parallelism is a node-level synchronization algorithm.

IV. Experimental Results and Analysis

A. Experiment Setup and Results

The experiments are implemented by C on a Linux server with two Xeon5670 CPUs (12 cores in total) and 24-GB random access memory. Twenty-six circuit matrices from the

TABLE I
RESULTS OF OUR ALGORITHM, AS COMPARED WITH KLU AND SUPERLU_MT

Matrix benchmark	[1] N $\times 10^3$	[2] NNZ $\times 10^3$	[3] NNZ_F KLU $\times 10^6$	[4] NNZ_F Ours $\times 10^6$	[5] NNZ_F SuperLU_MT $\times 10^6$	[6] KLU time(s)	[7] $P=1$ time(s)	speedup	[8] $P=4$ time(s)	speedup	[9] $P=8$ time(s)	speedup	[10] SuperLU_MT $P=1$ time(s)	$P=4$ time(s)	$P=8$ time(s)
add20	2.4	17.3	0.02	0.02	0.04	0.000	0.000	**1.13**	0.001	**0.67**	0.001	**0.63**	0.008	0.004	0.006
circuit_1	2.6	35.8	0.04	0.04	0.08	0.001	0.002	**0.89**	0.001	**1.65**	0.001	**2.23**	0.012	0.012	0.018
circuit_2	4.5	21.2	0.04	0.03	0.43	0.001	0.001	**0.99**	0.001	**1.67**	0.001	**1.18**	0.352	0.150	0.067
add32	5.0	23.9	0.02	0.02	0.03	0.001	0.000	**1.33**	0.000	**1.83**	0.000	**1.17**	0.007	0.004	0.010
rajat03	7.6	32.7	0.16	0.16	0.22	0.009	0.010	**0.87**	0.004	**2.06**	0.002	**5.15**	0.038	0.015	0.011
coupled	11.3	98.5	0.36	0.37	6.46	0.033	0.035	**0.94**	0.016	**2.12**	0.006	**5.52**	8.804	3.208	1.808
circuit_3	12.1	48.1	0.07	0.07	1.21	0.001	0.001	**1.11**	0.001	**1.17**	0.001	**1.54**	0.652	0.274	0.232
onetone1	36.1	341.1	11.20	2.88	4.54	14.847	1.920	**7.73**	0.513	**28.93**	0.241	**61.70**	3.655	0.925	0.664
onetone2	36.1	227.6	2.13	1.24	1.16	0.530	0.232	**2.29**	0.071	**7.51**	0.039	**13.47**	0.183	0.084	0.052
ckt11752_dc_1	49.7	333.0	1.06	1.52	15.45	0.052	0.147	**0.35**	0.049	**1.07**	0.028	**1.87**	12.710	3.930	2.970
circuit_4	80.2	307.6	0.44	0.43	17.89	0.016	0.015	**1.06**	0.007	**2.24**	0.006	**2.75**	46.239	46.463	28.900
ASIC_100ks	99.2	578.9	4.27	3.64	55.67	2.775	1.990	**1.39**	0.499	**5.56**	0.272	**10.22**	124.198	38.302	23.430
ASIC_100k	99.3	954.2	4.37	4.01	−−	2.098	1.770	**1.19**	0.443	**4.74**	0.246	**8.54**	−−	−−	−−
dc1	116.8	766.4	1.14	1.12	−−	0.066	0.064	**1.03**	0.026	**2.53**	0.017	**3.77**	−−	−−	−−
trans4	116.8	766.4	1.14	1.12	−−	0.069	0.066	**1.04**	0.026	**2.66**	0.020	**3.46**	−−	−−	−−
G2_circuit	150.1	726.7	19.97	19.97	37.14	13.050	14.289	**0.91**	3.735	**3.49**	2.215	**5.89**	20.440	5.251	3.145
transient	178.9	961.8	2.09	2.01	−−	0.329	0.347	**0.95**	0.098	**3.35**	0.062	**5.30**	−−	−−	−−
ASIC_320ks	321.7	1827.8	4.84	4.84	49.75	30.107	29.950	**1.01**	7.509	**4.01**	3.913	**7.69**	854.059	290.664	166.800
ASIC_320k	321.8	2635.4	5.54	5.63	−−	26.619	25.912	**1.03**	6.488	**4.10**	3.384	**7.87**	−−	−−	−−
rajat30	644.0	6175.4	31.70	19.12	−−	27.284	5.817	**4.69**	1.775	**15.37**	1.032	**26.44**	−−	−−	−−
ASIC_680ks	682.7	2329.2	4.96	4.96	−−	1.509	1.580	**0.96**	0.523	**2.88**	0.300	**5.04**	−−	−−	−−
ASIC_680k	682.9	3871.8	6.65	6.59	−−	2.487	1.986	**1.25**	0.615	**4.04**	0.381	**6.53**	−−	−−	−−
G3_circuit	1585.5	7660.8	377.00	376.62	−−	646.432	663.072	**0.97**	251.904	**2.57**	175.380	**3.69**	−−	−−	−−
Freescale1	3428.8	18920.3	61.28	61.28	120.03	13.190	16.268	**0.81**	6.450	**2.04**	4.049	**3.26**	44.448	12.958	8.140
rajat31	4690.0	20316.3	365.00	350.70	−−	538.253	461.564	**1.17**	170.129	**3.16**	101.920	**5.28**	−−	−−	−−
circuit5M	5558.3	59524.3	61.96	62.07	−−	2.401	2.023	**1.19**	1.089	**2.20**	0.820	**2.93**	−−	−−	−−
arithmetic-average								**1.47**		**4.37**		**7.81**			
geometric-average								**1.18**		**2.97**		**4.55**			

[1] matrix dimension
[2] the number of nonzeros in A
[3] the number of nonzeros in $L+U$, after factorization, by KLU
[4] the number of nonzeros in $L+U$, after factorization, by our algorithm
[5] the number of nonzeros in $L+U$, after factorization, by SuperLU_MT
[6] the numeric factorization time (in seconds) of KLU re-factorization
[7,8,9] the numeric factorization time (in seconds) and speedup values of our algorithm compared with *klu_refactor*, where P is the number of threads
[10] the numeric factorization time (in seconds) of SuperLU_MT, where P is the number of threads

University of Florida Sparse Matrix Collection [20] are used to evaluate our algorithm. We also test KLU (with the use of ***klu_refactor*** function but not *klu_factor*) and SuperLU_MT, which are considered as the baseline for comparison. KLU and SuperLU_MT are implemented with their default configurations. We define two types of speedups, i.e., the pure **speedup** is defined as the speedup compared with KLU, and the **relative speedup** is defined as the speedup compared with the sequential version of our proposed algorithm.

Table I shows the **speedup** of the proposed parallel algorithm compared with KLU, and Table II shows the **relative speedup** of large matrices $(N > 10K)$. It indicates that, for most circuit matrices, our parallel LU factorization algorithm can achieve stable acceleration with the multithread parallelism. In addition, we get better performance than SuperLU_MT since SuperLU_MT fails on many large matrices (see "−−" in Table I), and the number of fill-ins is much larger than our algorithm and KLU.

TABLE II
RELATIVE SPEEDUP, COMPARISON AMONG DIFFERENT PARALLEL MODES ($P = 4$), AND *EScheduler* INFORMATION

Matrix benchmark	relative speedup $P=8$	relative speedup $P=4$	[1] Mode_P $P=4$	[2] Mode_C $P=4$	[3] cluster levels	[4] cluster nodes
coupled	5.87	2.26	2.35	1.19	5.1	90.3
circuit_3	1.39	1.06	0.92	1.25	23.0	98.4
onetone1	7.98	3.74	2.43	1.05	2.7	92.2
onetone2	5.89	3.28	2.41	1.03	2.6	94.1
ckt11752_dc_1	5.28	3.01	2.55	1.47	6.4	91.8
circuit_4	2.60	2.12	0.92	1.09	7.1	99.6
ASIC_100ks	7.33	3.99	2.78	1.03	3.7	97.4
ASIC_100k	7.20	4.00	2.73	1.08	4.8	97.3
dc1	3.67	2.46	1.55	1.25	6.5	99.2
trans4	3.31	2.55	1.55	1.25	6.5	99.2
G2_circuit	6.45	3.83	2.64	1.27	5.8	94.2
transient	5.59	3.53	2.20	1.04	5.3	99.2
ASIC_320ks	7.65	3.99	3.67	1.01	3.7	99.2
ASIC_320k	7.66	3.99	3.74	1.04	4.1	99.1
rajat30	5.64	3.28	2.69	1.10	6.2	99.0
ASIC_680ks	5.27	3.02	2.78	1.05	4.2	99.6
ASIC_680k	5.22	3.23	2.59	1.11	4.7	99.6
G3_circuit	3.78	2.63	2.51	1.26	8.5	97.9
Freescale1	4.02	2.52	1.99	1.43	15.1	99.7
rajat31	4.53	2.71	2.58	1.23	6.7	99.3
circuit5M	2.47	1.86	1.42	1.14	3.2	99.9
arithmetic-average	**5.18**	**3.00**	**2.33**	**1.16**	**6.47**	**97.44**
geometric-average	**4.78**	**2.88**	**2.20**	**1.15**	**5.56**	**97.40**

[1] **Mode_P**: factorizing all columns using **pure *pipekine-like mode*** parallelism
[2] **Mode_C**: factorizing all columns using **pure *cluster mode*** parallelism
[3] the percentage of the levels computed in *cluster mode*
[4] the percentage of the nodes (columns) computed in *cluster mode*

B. Result Analysis

1) Synchronization Overhead: For small matrices $(N < 5K)$, the speedup values keep low and do not obviously increase or even decrease with more threads. Because the factorization operations of small matrices consume little time, thread operations such as the thread synchronization consume a large part of the total runtime.

2) Impact of the Fill-Ins: For some matrices, we get very high speedup values. This is because of the preprocessing step, i.e., KLU uses BTF, by default. Our algorithm, however, does not use BTF. Table I shows the number of nonzeros in $L+U$ after factorization. For some matrices, the fill-ins have big differences between our implementation and KLU,

such as onetone1, onetone2, and rajat30. However, the **relative speedup** still keeps stable, as shown in Table II.

3) Comparison Among Different Parallel Modes: Table II shows the comparison among the combined *cluster mode* and *pipeline mode* (i.e., the proposed algorithm, combined mode in short), the pure *cluster mode* (Mode_C), and the pure *pipeline mode* (Mode_P), with four threads. All the values reported in Table II are **relative speedups**. The combined mode is much better than both pure *cluster mode* and pure *pipeline mode*. Another phenomenon is that the pure *pipeline mode* has the similar trend with the combined mode, whereas the pure *cluster mode* is extremely terrible. To explain this, we show the *EScheduler* information in the last two columns of Table II. The nodes (columns) computed by *cluster mode* account for more than 97% of the total nodes, but the levels computed by *cluster mode* only account for about 5% of the total levels. It means that the levels computed by *pipeline mode* occupy a large part and form a long chain in the *EGraph* and the *EScheduler*. Since the *pipeline mode* has much more synchronization overhead than the *cluster mode*, the bottleneck of the parallel task scheduling flow is the *pipeline mode*. Therefore, the overall performance is mainly determined by the *pipeline mode*. The conclusion is that the two structures of the *EScheduler* dramatically differ; therefore, different parallel strategies should be utilized and dynamically selected to fit the different data dependence.

4) Consideration of Scalability and Matrix Characteristic: Tables I and II show that our parallel LU factorization algorithm works well for most of the circuit matrices. Since we do not have a platform with more cores, we cannot get the actual results with more threads. The performance with more cores mainly depends on the shape of the *EGraph* and the *EScheduler*, the synchronization overhead, and the memory bandwidth: 1) the shape of the *EGraph* and the *EScheduler* represents the data dependence and the matrix characteristics. If the data dependence is weaker, the *EGraph* will be wider and shorter and have less edges. When the nodes computed by *cluster mode* become more, the performance will be better since the *cluster mode* has less synchronization overhead; 2) since the *pipeline mode* is a node-level synchronization algorithm, the synchronization overhead will increase with more threads and then affect the performance. If the problem size also becomes larger, the computational time will be much more than the synchronization time, and vice versa. This is why we get poor performance for small matrices; and 3) if more threads are utilized, the memory bandwidth will become a big bottleneck since all the threads fetch lots of data from the memory simultaneously. Combining with the distributed approach may be a potential solution for this challenge.

The size of the circuit matrices we used is up to five million, which is larger than the available test results of other software [5], [6], [9]–[11], [13] and FPGA implementations [2], [15], [16]. Our algorithm can be scaled to bigger problem sizes.

V. Conclusion

The sparse matrix solver becomes the bottleneck in fast SPICE-like simulators. In this brief, we have proposed an *EScheduler* to analyze data dependence and task scheduling, and then, a parallel LU factorization algorithm is developed on multicore CPUs. To improve parallel scalability, we have proposed two scheduling methods, i.e., *cluster mode* and *pipeline mode*, which are dynamically configured during the numeric factorization based on the *EScheduler*. The experimental results on 26 circuit matrices show that the proposed algorithm can achieve speedup of 1.18–4.55$\times$ on geometric average compared with KLU, with 1–8 threads. Our performance is also better than SuperLU_MT. We analyze the structures of the *EScheduler* and conclude that, for different data dependence, different parallel strategies should be dynamically selected.

In the future, we will consider multiple levels of parallel granularities in LU factorization. Furthermore, we will consider building a performance model to evaluate our algorithm and study the load balancing of multiple threads.

References

[1] L. W. Nagel, “SPICE 2: A computer program to stimulate semiconductor circuits,” Ph.D. dissertation, Univ. California, Berkeley, 1975.

[2] 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.

[3] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Boston, MA: PWS, 2004.

[4] T. A. Davis, *Direct Methods for Sparse Linear Systems*. Philadelphia, PA: SIAM, 2006.

[5] J. W. Demmel, J. R. Gilbert, and X. S. Li, “An asynchronous parallel supernodal algorithm for sparse Gaussian elimination,” *SIAM J. Matrix Anal. Appl.*, vol. 20, no. 4, pp. 915–952, Oct. 1999.

[6] 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. Appl.*, vol. 20, no. 3, pp. 720–755, 1999.

[7] 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.

[8] J. R. Gilbert and T. Peierls, “Sparse partial pivoting in time proportional to arithmetic operations,” *SIAM J. Sci. Statist. Comput.*, vol. 9, pp. 862–874, 1988.

[9] T. A. Davis and E. Palamadai Natarajan, “Algorithm 907: KLU, a direct sparse solver for circuit simulation problems,” *ACM Trans. Math. Softw.*, vol. 37, no. 3, pp. 36:1–36:17, Sep. 2010.

[10] T. A. Davis, “Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method,” *ACM Trans. Math. Softw.*, vol. 30, no. 2, pp. 196–199, Jun. 2004.

[11] 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.

[12] J. W. H. Liu, “The multifrontal method for sparse matrix solution: Theory and practice,” *SIAM Rev.*, vol. 34, no. 1, pp. 82–109, Mar. 1992.

[13] O. Schenk and K. Gartner, “Solving unsymmetric sparse systems of linear equations with PARDISO,” in *Proc. ICCS*, 2002, vol. 2330, pp. 355–363.

[14] O. Schenk, K. Gartner, and W. Fichtner, “Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors,” *BIT Numer. Math.*, vol. 40, no. 1, pp. 158–176, 2000.

[15] N. Kapre, “SPICE2—A spatial parallel architecture for accelerating the spice circuit simulator,” Ph.D. dissertation, California Inst. Technol., Pasadena, 2010.

[16] T. Nechma, M. Zwolinski, and J. Reeve, “Parallel sparse matrix solver for direct circuit simulations on FPGAs,” in *Proc. IEEE ISCAS*, 2010, pp. 2358–2361.

[17] 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, Oct. 1999.

[18] P. R. Amestoy, Enseeiht-Irit, 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.

[19] X. S. Li, “Sparse Gaussian elimination on high performance computers,” Ph.D. dissertation, Comput. Sci. Div., Univ. California, Berkeley, Sep. 1996.

[20] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM Trans. Math. Softw.* [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices