An Adaptive LU Factorization Algorithm for Parallel Circuit Simulation

Xiaoming Chen, Yu Wang, Huazhong Yang
Department of Electronic Engineering
Tsinghua National Laboratory for Information Science and Technology
Tsinghua University, Beijing, China
Email: chenxm05@mails.tsinghua.edu.cn, yu-wang@tsinghua.edu.cn

***Abstract*—Sparse matrix solver has become the bottleneck in SPICE simulator. It is difficult to parallelize the solver because of the high data-dependency during the numerical LU factorization. This paper proposes a parallel LU factorization (with partial pivoting) algorithm on shared-memory computers with multi-core CPUs, to accelerate circuit simulation. Since not every matrix is suitable for parallel algorithm, a predictive method is proposed to decide whether a matrix should use parallel or sequential algorithm. The experimental results on 35 circuit matrices reveal that the developed algorithm achieves speedups of 2.11$\times\sim$8.38$\times$ (on geometric-average), compared with KLU, with 1$\sim$8 threads, on the matrices which are suitable for parallel algorithm. Our solver can be downloaded from http://nicslu.weebly.com.**

I. Introduction

SPICE (Simulation Program with Integrated Circuit Emphasis) [1] circuit simulator developed by UC-Berkeley is widely deployed for verifications of ICs. However, with the growing complexity of the VLSI at nano-scale, the traditional SPICE simulator has become inefficient to provide accurate verifications for IC designs. In a typical SPICE simulation flow, the sparse matrix solver by LU factorization is repeated in every Newton-Raphson iterations during the transient analysis, so it becomes the bottleneck [2]. Consequently, acceleration for solving $A\vec{x} = \vec{b}$ has become an interest for developing fast SPICE-like simulators.

Research on sparse LU factorization has been active for decades, and there are some popular software implementations. SuperLU includes 3 versions: the sequential version [3], the multi-thread version SuperLU_MT [4], and SuperLU_DIST [5] for distributed machines. SuperLU incorporates *Supernode* in left-looking Gilbert-Peierls (G-P) algorithm [6] 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. So in KLU [7], *Block Triangular Form (BTF)* is adopted without *Supernode*. UMFPACK [8], and MUMPS [9], are based on multifrontal algorithm [10]. In PARDISO [11], the left-right-looking algorithm [12] is developed. Among all the software above, only KLU is specially optimized for circuit simulation.

There are also some works that use iterative method to perform parallel circuit simulation, such as [13], [14]. However, iterative method requires good pre-condition in each iteration, its advantage is unclear for circuit simulation.

Recently, several approaches are developed on reconfigurable devices [15], [16]. The scalability to large-scale circuits is still limited by FPGA on-chip resources and the bandwidth between FPGA and CPU.

In this paper, in order to accelerate circuit simulation, a column-level parallel sparse LU factorization algorithm (with partial pivoting) is developed on multi-core CPUs. Our previous work [17] has presented a column-level parallel LU factorization algorithm without partial pivoting. The primary differences between [17] and this paper (*i.e.* the contributions of this work) are:

- Not every matrix is suitable for parallel factorization, so we present a predictive method to decide whether a matrix should use sequential or parallel solver (Section II). Consequently, each matrix can achieve the optimal performance.
- This work is implemented with partial pivoting to enhance numeric stability. When adopting partial pivoting, data dependency cannot be determined prior factorization, the primary challenge is to dynamically determine data dependency during factorization (Section IV). The parallel algorithm achieves speedups of 2.11$\times\sim$8.38$\times$ (on geometric-average), compared with KLU, with 1$\sim$8 threads, on the matrices which are suitable for parallel factorization (Section V).

II. The Solver Flow

Fig. 1 shows the overall flow of the proposed adaptive solver. A typical LU factorization has three steps: 1) pre-processing, 2) factorization, and 3) right-hand-solving. Among them, the first step performs column/row permutations to increase numeric 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 $U\vec{x} = \vec{y}$.

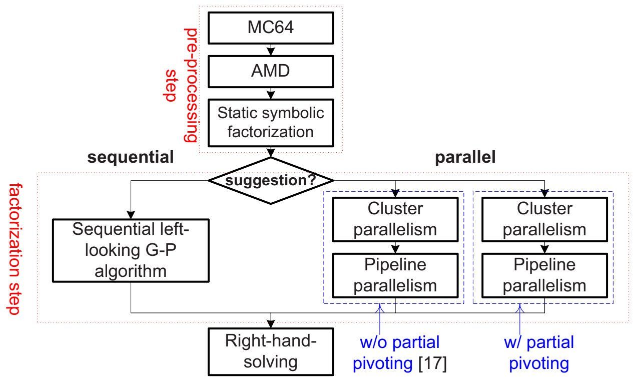

Fig. 1. The overall flow of the proposed adaptive solver

Y. Wang is the Corresponding Author.
This work was supported by National Key Technological Program of China (2008ZX01035-001, 2010ZX01030-001) and National Natural Science Foundation of China (No.60870001).

In circuit simulation, the pre-processing step is performed only once, the factorization step and right-hand-solving are repeated for many times during the iterations.

We introduce the pre-processing step in this section, the factorization step will be introduced in Section III (sequential algorithm) and IV (parallel algorithm).

Our pre-processing step consists of 3 algorithms: MC64 algorithm [18], AMD (Approximate Minimum Degree) algorithm [19], and a static symbolic factorization [6], [20].

1) MC64 algorithm: MC64 algorithm [18] is to find a column permutation matrix P_c and two diagonal scaling matrices D_r and D_c, to make each diagonal entry of $A_1 = D_r A P_c D_c$ is ± 1, and each off-diagonal entry is bounded by 1 in absolute value. MC64 algorithm can effectively enhance numeric stability.

2) AMD algorithm: AMD algorithm [19] is a fill-reducing ordering algorithm to minimize fill-ins for LU factorization, which finds a symmetric row/column permutation matrix P and performs the symmetric ordering $A_2 = P(A_1 + A_1^T)P^T$, such that factorizing A_2 has much fewer fill-ins than factorizing A.

3) Static symbolic factorization: This step is used to give a suggestion that whether a matrix should use parallel or sequential algorithm. It is a direct implementation of the G-P symbolic factorization algorithm [6], [20], without any partial pivoting or numeric computation. After this step, we obtain the nonzero structure of L and U, and the number of nonzeros in symbolic matrix $L + U$ (NNZ_S). We further calculate $\delta = \frac{NNZ_S}{NNZ_A}$ (NNZ_A: the number of nonzeros in A), and suggest that if $\delta < 2.0$, the sequential algorithm should be utilized, otherwise the parallel algorithm should be used. A detailed discussion will be presented in Section V.

Our pre-processing is different from KLU. In KLU, *BTF* is adopted to permute the matrix into an upper block triangular form, but without MC64 algorithm.

In the following content, we still regard the matrix after pre-processing step as A.

III. The Sequential LU factorization algorithm

The factorization step is based on the left-looking G-P algorithm [6], which is shown in Fig. 2. It factorizes matrix A by sequentially processing each column (k) in 4 steps: 1) symbolic factorization, 2) numeric factorization, 3) partial pivoting, and 4) storing x in to U and L (with normalization). The first 3 steps are introduced in the below, with the example of factorizing column k.

```
Algorithm: Left-looking Gilbert-Peierls algorithm
L = I;
for k = 1:N do
    Symbolic factorization: determine the structure of the
    kth column, and which columns will update the kth column;
    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
```

(a)

```
Algorithm: Solving Lx=b, where b=A(:, k)
x = b;
for j<k where U(j, k)!=0, in topological order do
    x(j+1:n) = x(j+1:n) - x(j)*L(j+1:n, j);
end for
```

(b)

Fig. 2. The left-looking Gilbert-Peierls algorithm [6], [7]

A. Symbolic factorization

The purpose of the symbolic factorization is to determine the nonzero structure of the kth column, and which columns will update column k. [20] has presented the theory of symbolic factorization, and the detailed method and a DFS (Depth First Search)-based implementation can be found in [7], [21].

B. Numeric factorization

This step is the most time-consumed step, which is shown in Fig. 2(b). It indicates that the numeric factorization of column k refers to the data in 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**. This is the basic column-level dependency in sparse left-looking LU factorization. In the next section, we will introduce the parallel LU factorization algorithm based on this column-level dependency.

C. Partial pivoting

Although we utilize MC64 algorithm in the pre-processing step to enhance numeric stability, it is still possible to generate small pivots on the diagonal during the numeric factorization (leading to unstable algorithm), especially in digital circuit simulation (the matrix entries may vary dramatically according to the digital circuit operation). So partial pivoting is still needed. It is done by finding the largest element in each column, swapping it to the diagonal when the original diagonal entry is small enough [6], [21]. However, off-diagonal pivoting can increase fill-ins dramatically.

Even worse, when adopting partial pivoting, the nonzero structure of L and U depends on pivot choices (partial pivoting can interchange the rows). So the exact dependency cannot be determined before the factorization. The challenge is to dynamically determine the exact dependency during factorization.

IV. Parallel LU Factorization Strategies

In this section, we introduce the column-level parallel LU factorization algorithm, which is developed based on the left-looking G-P algorithm [6] and KLU algorithm [7].

As mentioned above, the column-level dependency is determined by the structure of U. In [17], since partial pivoting is not adopted, the structure of L and U can be calculated by a static symbolic factorization before numeric factorization, the exact dependency is obtained easily. An *Elimination Scheduler* (*EScheduler*) is 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 numeric pivot choices. So the exact dependency cannot be determined before

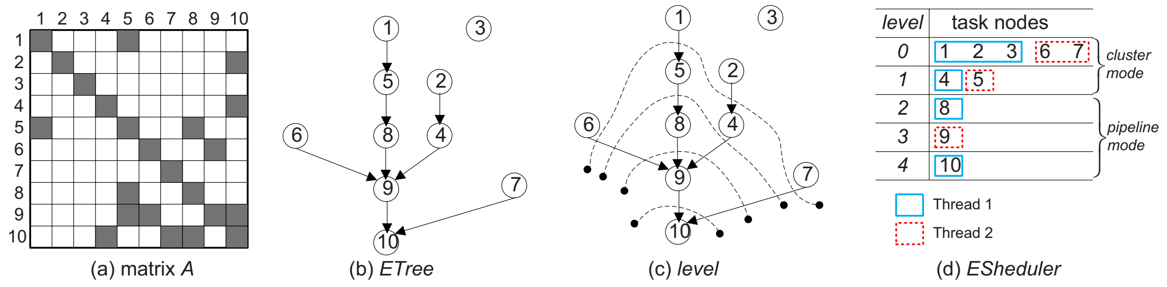

Fig. 3. An example to illustrate *ETree*, *level*, and *EScheduler*

the factorization. This is the primary difference between this work and [17], which also leads to the differences of the parallel algorithms.

In this paper, we still use *EScheduler* to schedule the parallel tasks by *cluster mode* and *pipeline mode*. Different from [17], *EScheduler* in this paper is built from an *Elimination Tree* (*ETree*), which is used in SpuerLU_MT [4].

A. Elimination Tree

In SuperLU_MT, *ETree* is adopted to represent the column-level dependency [4], [21]. It's not an exact representation, but contains all potential dependency regardless of the actual pivoting choices, so it overestimates the actual dependency. The definition of *ETree* and how to calculate the *ETree* can be found in [4], [21], here we only present an example, as shown in Fig. 3(b). Node k in the *ETree* corresponds to column k in the matrix, so in the following content, "node" and "column" are equivalent. The edges in the *ETree* represent the potential dependency.

B. Elimination Scheduler

Actually the parallel tasks can be directly scheduled by the *ETree*. However, since the circuit matrices are extremely sparse, and so are their LU factors, the computational time of one node is so little that the scheduling time of one node may be larger than its computational time. We attempt to reduce the scheduling time as much as possible. To explore some commonness from *ETree*, we define *level* of each node in the *ETree*.

***Level* Definition:** *Given the ETree, the level of each node is the length of the longest path from the leaf nodes to the node itself. Leaf node is the node which has no incoming edges.*

It's simple to calculate *level* of each node by topological order:

$$level(k) = \max(-1, level(c_1), level(c_2), \cdots) + 1 \qquad (1)$$

where $c_1, c_2, \cdots$ are the *children* of node k (*child* definition: if there is an edge $i \rightarrow j$ in the *ETree*, i is a *child* of j).

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 $V = \{1, 2, \cdots, n\}$ corresponds to the nodes in the ETree, and $LV = \{level(k) | 1 \le k \le n\}$.*

Fig. 3(d) shows an example of the *EScheduler*, corresponding to the *ETree* in Fig. 3(b). The above definitions indicate that the nodes in one level are independent. Although the *EScheduler* is coarser-grained than *ETree*, it is sufficient to implement the column-level parallelism and reduce the scheduling time.

C. EScheduler guided parallel LU factorization

In this paper, we still use *cluster mode* and *pipeline mode* to perform the parallel LU factorization. The primary difference from [17] is that when adopting partial pivoting, the symbolic factorization cannot be separated from numeric factorization, so the exact dependency is determined dynamically.

We use Fig. 3(d) as an example to illustrate the parallel algorithm. Since the nodes in one level are independent, these nodes can be calculated in parallel effectively, with little scheduling time. It's *cluster mode*. However, when the nodes in one level become fewer, *cluster mode* is ineffective. In this case, *pipeline mode* which **exploits parallelism between dependent levels** will be used.

We set a threshold N_{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*. $N_{th} = 1$ is adopted in this example. In practice, we find that $N_{th} = 10 \sim 20$ is the optimal value for the cases with less than 12 threads. For the above example, level $0 \sim 1$ are factorized by *cluster mode*, and then the rest nodes are factorized in parallel by *pipeline mode*.

1) Parallelism in 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 nodes in each *cluster*. Each thread performs the same code (*i.e.* the left-looking G-P algorithm) to factorize the nodes which are assigned to it. 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 finish factorizing the current level, and then the nodes in the next level will be processed by the same strategy. Fig. 3(d) shows an example of node allocation to 2 threads in *cluster mode*.

2) Parallelism in 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's different from the *pipeline mode* in [17].

We firstly sort all the nodes belonging to *pipeline mode* into a sequence (*i.e.* in the above example, the sequence is $8, 9, 10$),

TABLE I
RESULTS OF OUR ALGORITHM, COMPARED WITH KLU

Matrix benchmark	N	NNZ_A	KLU				Our algorithm								
			time(s)	fill-in	flops	residual	$P=1$		$P=4$		$P=8$		fill-in	flops	residual
	$\times 10^3$	$\times 10^3$					time(s)	speedup	time(s)	speedup	time(s)	speedup			
Set 1 benchmarks															
rajat03	7.6	32.7	0.024	**4.79**	**6.55E+06**	1.52E-20	0.023	**1.02**	0.012	**1.99**	0.013	**1.89**	4.89	6.97E+06	**1.49E-20**
coupled	11.3	98.5	0.074	**3.68**	**2.37E+07**	2.79E-19	0.074	**1.00**	0.027	**2.67**	0.026	**2.80**	3.79	2.47E+07	**2.34E-19**
onetone1	36.1	341.1	16.373	32.83	1.08E+10	8.48E-13	2.373	**6.90**	0.601	**27.24**	0.319	**51.38**	**8.90**	**1.34E+09**	**1.72E-17**
onetone2	36.1	227.6	0.620	9.36	4.66E+08	1.39E-13	0.265	**2.34**	0.116	**5.35**	0.081	**7.71**	**5.46**	**1.98E+08**	**1.37E-17**
ckt11752_dc_1	49.7	333.0	0.086	**3.17**	**3.51E+07**	5.18E-18	0.417	**0.21**	0.252	**0.34**	0.154	**0.56**	6.47	3.04E+08	**4.55E-18**
ASIC_100ks	99.2	578.9	2.924	7.38	2.19E+09	2.52E-23	1.793	**1.63**	0.636	**4.60**	0.398	**7.35**	**6.29**	**1.39E+09**	**9.37E-24**
ASIC_100k	99.3	954.2	2.342	4.58	1.73E+09	3.89E-23	1.501	**1.56**	0.629	**3.72**	0.433	**5.41**	**4.20**	**1.11E+09**	**3.45E-23**
twotone	120.8	1224.2	80.717	37.08	5.57E+10	6.53E-12	14.069	**5.74**	4.832	**16.70**	2.962	**27.25**	**9.41**	**1.09E+10**	**9.52E-19**
G2_circuit	150.1	726.7	13.153	**27.48**	**1.00E+10**	**1.34E-18**	12.989	**1.01**	3.970	**3.31**	2.377	**5.53**	**27.48**	**1.00E+10**	1.39E-18
transient	178.9	961.8	0.417	2.17	2.60E+08	**3.73E-20**	0.358	**1.16**	0.260	**1.60**	0.228	**1.83**	**2.09**	**2.27E+08**	1.79E-19
mac_econ_fwd500	206.5	1273.4	13414.872	726.41	5.63E+12	8.42E-16	46.125	**290.84**	18.883	**710.41**	11.129	**1205.41**	**52.96**	**3.81E+10**	**5.00E-19**
Raj1	263.7	1302.5	110.815	77.55	8.23E+10	3.81E-15	1.108	**100.01**	0.641	**172.93**	0.561	**197.68**	**5.60**	**7.18E+08**	**6.63E-21**
ASIC_320ks	321.7	1827.8	31.342	**2.65**	**1.37E+09**	8.33E-36	31.061	**1.01**	8.946	**3.50**	4.717	**6.64**	**2.65**	**1.37E+09**	**7.02E-36**
ASIC_320k	321.8	2635.4	27.632	**2.10**	**1.21E+09**	8.64E-35	27.433	**1.01**	7.585	**3.64**	4.202	**6.58**	2.14	1.23E+09	**6.08E-35**
mc2depi	525.8	2100.2	85.328	**25.88**	**2.04E+10**	**6.74E-17**	84.513	**1.01**	23.643	**3.61**	12.810	**6.66**	**25.88**	**2.04E+10**	6.75E-17
rajat30	644.0	6175.4	27.498	5.13	1.98E+10	3.03E-12	6.661	**4.13**	2.879	**9.55**	2.074	**13.26**	**3.10**	**4.72E+09**	**1.50E-22**
pre2	659.0	5959.3	FAIL				505.565	$--$	159.423	$--$	102.698	$--$	**17.96**	**4.17E+11**	**3.21E-27**
ASIC_680ks	682.7	2329.2	2.915	**2.13**	**9.18E+08**	**2.99E-28**	1.752	**1.66**	0.764	**3.82**	0.522	**5.59**	**2.13**	**9.18E+08**	6.16E-28
Hamrle3	1447.4	5514.2	FAIL				1073.550	$--$	392.018	$--$	249.118	$--$	**62.20**	**8.49E+11**	**1.61E-05**
G3_circuit	1585.5	7660.8	760.998	**49.21**	**5.99E+11**	**1.38E-18**	747.161	**1.02**	229.002	**3.32**	146.063	**5.21**	**49.21**	**5.99E+11**	1.44E-18
memchip	2707.5	14810.2	462.144	**14.79**	**3.64E+11**	4.41E-20	456.546	**1.01**	140.346	**3.29**	87.854	**5.26**	**14.79**	**3.64E+11**	**4.07E-20**
Freescale1	3428.8	18920.3	16.219	**3.24**	**1.17E+10**	8.59E-21	15.646	**1.04**	6.948	**2.33**	4.857	**3.34**	**3.24**	**1.17E+10**	**8.21E-21**
circuit5M_dc	3523.3	19194.2	90.940	**4.11**	**2.71E+10**	3.89E-34	89.704	**1.01**	27.081	**3.36**	15.473	**5.88**	**4.11**	**2.71E+10**	**3.55E-34**
rajat31	4690.0	20316.3	581.478	17.97	4.88E+11	**5.41E-20**	535.335	**1.09**	159.805	**3.64**	100.917	**5.76**	**17.28**	**4.27E+11**	5.51E-20
arithmetic-average								**19.43**		**45.08**		**71.77**			
geometric-average								**2.11**		**5.60**		**8.38**			
Set 2 benchmarks															
add20	2.4	17.3	0.002	**1.00**	**1.31E+05**	1.53E-17	0.002	**1.05**	0.011	**0.15**	0.004	**0.38**	**1.00**	**1.31E+05**	**1.46E-17**
circuit_1	2.6	35.8	0.004	**1.18**	**6.64E+05**	2.44E-20	0.004	**1.00**	0.004	**1.20**	0.026	**0.16**	1.21	9.09E+05	**1.34E-20**
circuit_2	4.5	21.2	0.003	1.68	4.45E+05	1.27E-16	0.003	**1.06**	0.003	**1.09**	0.020	**0.17**	**1.54**	**4.16E+05**	**1.06E-21**
add32	5.0	23.9	0.002	**1.00**	**4.87E+04**	1.94E-17	0.002	**0.71**	0.005	**0.36**	0.018	**0.10**	**1.00**	**4.87E+04**	**1.83E-17**
circuit_3	12.1	48.1	0.007	**1.38**	**2.12E+05**	1.13E-18	0.006	**1.22**	0.010	**0.69**	0.013	**0.55**	1.42	2.47E+05	**2.70E-21**
circuit_4	80.2	307.6	0.032	1.44	6.93E+06	**3.85E-20**	0.025	**1.27**	0.068	**0.47**	0.063	**0.50**	**1.40**	**5.36E+06**	9.99E-20
hcircuit	105.7	513.1	0.046	**1.22**	**2.30E+06**	1.92E-19	0.036	**1.28**	0.076	**0.60**	0.077	**0.59**	1.23	2.42E+06	**1.88E-19**
dc1	116.8	766.4	0.114	1.49	3.83E+07	**4.35E-19**	0.102	**1.11**	0.146	**0.78**	0.147	**0.77**	**1.47**	**3.59E+07**	1.72E-18
trans4	116.8	766.4	0.118	1.48	3.82E+07	**2.03E-20**	0.108	**1.09**	0.148	**0.79**	0.144	**0.82**	**1.47**	**3.59E+07**	3.72E-20
ASIC_680k	682.9	3871.8	2.989	1.72	1.07E+09	**1.68E-28**	2.047	**1.46**	1.361	**2.20**	1.141	**2.62**	**1.70**	**9.99E+08**	3.44E-28
circuit5M	5558.3	59524.3	4.346	**1.04**	**1.00E+09**	**4.57E-19**	3.299	**1.32**	6.129	**0.71**	6.390	**0.68**	**1.04**	1.10E+09	6.64E-19
arithmetic-average								**1.14**		**0.82**		**0.67**			
geometric-average								**1.13**		**0.67**		**0.46**			

The speedups and the best values of fill-in, flops and residual are shown in **boldface**
speedup: the runtime compared with KLU
N: matrix dimension NNZ_A: the number of nonzeros in A
fill-in: it means relative fill-in in this paper, *i.e.* $\frac{NNZ_F}{NNZ_A}$, where NNZ_F is the number of nonzeros in $L+U$
flops: the total number of floating-point operations
residual: it's defined as $\frac{||Ax-b||}{||A||||x||+||b||}$, where $||\cdot||$ is 1-norm

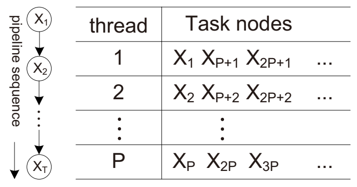
pipeline sequence
X_1
X_2
X_T
thread
Task nodes
1
X_1 X_{P+1} X_{2P+1} ...
2
X_2 X_{P+2} X_{2P+2} ...
P
X_P X_{2P} X_{3P} ...

Fig. 4. The static scheduling method in *pipeline mode* parallelism

and then perform a static scheduling, which is illustrated in Fig. 4. The task nodes of each thread are completely determined when the *pipeline mode* starts, which also reduces the scheduling time.

In *pipeline mode*, each thread also performs the same code, as shown in Fig. 5. The code can be partitioned into 2 parts: pre-factorization and post-factorization.

In pre-factorization, the thread (factorizing node k) performs incomplete symbolic factorization and numeric factorization repeatedly. Symbolic factorization is to determine which columns will update column k. However, currently some dependent columns are not finished, they will be skipped. And then it uses the finished columns to perform numeric factorization, to update column k. The code of numeric accumulation is in Fig. 2(b), the difference is that currently it only uses part of the dependent columns (*i.e.* the finished columns) to update column k. The pre-factorization will be repeated until the previous column in the pipeline sequence is finished, then it performs the post-factorization.

In post-factorization, the thread performs a complete symbolic factorization without skipping any columns (all the dependent columns are finished currently), to determine the exact structure of the kth column, and then uses the rest columns (*i.e.* un-finished columns in the pre-factorization) to do the numeric update.

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

```
Algorithm: The pipeline parallelism of each thread
1  while (the pipeline tail is not achieved) do
2      Get a new column k;
       pre-factorization:
3      while (the previous column in the pipeline
       sequence is not finished) do
4          Symbolic factorization, skip all
           un-finished columns;
5          Numeric factorization: using the finished
           columns to update column k;
6      end while
       post-factorization:
7      Symbolic factorization: determine the exact
       structure of the kth column of L and U;
8      Numeric factorization: using the rest columns
       to update column k;
9      Partial pivoting;
10 end while
```

Fig. 5. The *pipeline* parallelism of each thread

TABLE II
THE RELATIVE SPEEDUPS AND BENCHMARK CATEGORIZATION

Matrix benchmark	relative speedup		NNZ_S	Matrix benchmark	relative speedup		NNZ_S
	$P=4$	$P=8$	NNZ_A		$P=4$	$P=8$	NNZ_A
Set 1 benchmarks				**Set 2 benchmarks**			
rajat03	1.95	1.84	4.89	add20	0.14	0.36	1.00
coupled	2.68	2.80	3.79	circuit_1	1.20	0.16	1.21
onetone1	3.94	7.45	8.44	circuit_2	1.03	0.16	1.54
onetone2	2.28	3.29	5.46	add32	0.51	0.14	1.00
ckt11752_dc_1	1.65	2.70	3.33	circuit_3	0.56	0.45	1.42
ASIC_100ks	2.82	4.51	6.29	circuit_4	0.37	0.40	1.40
ASIC_100k	2.39	3.47	4.20	hcircuit	0.47	0.46	1.23
twotone	2.91	4.75	9.44	dc1	0.70	0.69	1.47
G2_circuit	3.27	5.46	27.48	trans4	0.73	0.75	1.47
transient	1.38	1.57	2.09	ASIC_680k	1.50	1.79	1.70
mac_econ_fwd500	2.44	4.14	47.54	circuit5M	0.54	0.52	1.04
Raj1	1.73	1.98	5.60				
ASIC_320ks	3.47	6.59	2.65				
ASIC_320k	3.62	6.53	2.14				
mc2depi	3.57	6.60	25.88				
rajat30	2.31	3.21	3.10				
pre2	3.17	4.92	17.12				
ASIC_680ks	2.29	3.36	2.13				
Hamrle3	2.74	4.31	43.71				
G3_circuit	3.26	5.12	49.21				
memchip	3.25	5.20	14.79				
Freescale1	2.25	3.22	3.24				
circuit5M_dc	3.31	5.80	4.11				
rajat31	3.35	5.30	17.28				
arithmetic-average	**2.75**	**4.34**	**13.09**	**arithmetic-average**	**0.70**	**0.53**	**1.32**
geometric-average	**2.66**	**4.01**	**7.63**	**geometric-average**	**0.60**	**0.41**	**1.30**

NNZ_S: the number of nonzeros in $L+U$ after a static symbolic factorization
relative speedup: the runtime compared with our sequential algorithm

V. EXPERIMENTAL RESULTS AND ANALYSIS

A. Experimental Setup

The experiments are implemented by C on a 64-bit Linux server with 2 Xeon5670 CPUs (12 cores in total) and 24GB RAM. 35 circuit matrices from University of Florida Sparse Matrix Collection [22] are used to evaluate our algorithm. We also test KLU [7] as a baseline for comparison. KLU is implemented with the default configurations. We define two types of speedups: the pure **speedup** is defined as the speedup compared with KLU, and the **relative speedup** is defined as the speedup compared with the sequential algorithm.

The size of the benchmarks we used is up to 5.5 million, which is larger than the available test results of other software [3]–[5], [7]–[9], [11], iterative method implementations for circuit simulation [13], [14], and FPGA [15], [16] implementations. Our solver can be scaled to bigger problem sizes.

B. Experimental Results

Table I shows the results of our parallel LU factorization algorithm, and the comparison with KLU. The time listed in the table is the factorization time, excluding the time of pre-processing and right-hand-solving. These two steps are not time-consumed, and for the 35 matrices, they only cost 4.6% and 0.16% (on average) of the sequential factorization time, respectively. We also compare the fill-ins, flops, and residual of KLU and our algorithm. The speedups over KLU and the best values of fill-ins, flops and residual are shown in boldface.

We categorize all the benchmarks into two sets, named Set1 and Set2, by our proposed predictive method. We suggest using parallel algorithm on Set1 and sequential algorithm on Set2, and the results support our suggestion. We will discuss this point in the next subsection. Our parallel algorithm is generally faster than KLU on Set1 and slower on Set2. The parallel algorithm achieves speedups of 2.11×∼8.38× on geometric-average over KLU, when $P = 1 \sim 8$ (P is the thread number), on Set1 benchmarks. Our sequential algorithm ($P = 1$) is a little faster than KLU, since the MC64 algorithm leads to fewer off-diagonal pivots. From the comparison of fill-ins, flops, and residual, our algorithm performs better than KLU on most matrices.

We get very high speedups over KLU on some matrices (onetone1, onetone2, twotone, Raj1, mac_econ_fwd500, and rajat30), and low speedups on ckt11752_dc_1. This is because of the pre-processing step: KLU uses BTF, by default; our algorithm does not use BTF but uses MC64. These differences lead to differences of the matrix structure and pivoting choices, and then lead to big differences of the fill-ins. Specifically, *BTF* can dramatically affect matrix structure and MC64 algorithm can dramatically affect pivot choices.

C. Benchmark Categorization

In this subsection, we show the relative speedups and discuss the benchmark categorization. On Set1 benchmarks, our parallel algorithm can always get effective relative speedups but not so on Set2. We claim that not every matrix is suitable for parallel algorithm, and this is because of the characteristics of the matrix itself.

If the numeric computation of one matrix is somewhat little, the parallel algorithm cannot achieve effective speedups. That's what we find from the results. As mentioned in Section II, we calculate $\delta = \frac{NNZ_S}{NNZ_A}$, which can be regarded as the predictive relative numeric computation. We suggest that if $\delta < 2.0$, the sequential algorithm should be utilized, otherwise the parallel algorithm should be used. Table II shows the $\frac{NNZ_S}{NNZ_A}$ values. On Set2 benchmarks, $\frac{NNZ_S}{NNZ_A}$ values are all near 1.0, which means they only increase few fill-ins after LU factorization, so the numeric computation of these matrices is very little. On the contrary, the parallel overhead, such as scheduling time, waiting time, memory and cache conflicts, will dominate in the total runtime.

To prove this finding, we also test SuperLU_MT [4], which is a general-purpose parallel solver by multi-thread

TABLE III
THE RESULTS OF SUPERLU_MT

Matrix benchmark	KLU		SuperLU_MT						
	fill-in	time(s)	fill-in	$P=1$		$P=4$		$P=8$	
				time(s)	speedup	time(s)	speedup	time(s)	speedup
Set 1 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
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
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						
twotone	37.08	80.717	**15.83**	15.976	5.05	3.879	20.81	**2.306**	35.00
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						
mac_econ_fwd500	726.41	13414.872	**134.64**	165.801	80.91	47.968	279.66	**29.038**	461.98
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						
mc2depi	**25.88**	85.328	36.61	82.607	1.03	22.999	3.71	**12.975**	6.58
rajat30	**5.13**	**27.498**	FAIL						
pre2	FAIL		**56.12**	720.134	−−	226.172	−−	**136.702**	−−
ASIC_680ks	**2.13**	**2.915**	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						
arithmetic-average					**7.27**		**25.22**		**41.28**
geometric-average					**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						
ASIC_680k	**1.72**	**2.989**	FAIL						
circuit5M	**1.04**	**4.346**	FAIL						
arithmetic-average					**0.13**		**0.21**		**0.13**
geometric-average					**0.03**		**0.06**		**0.06**

The best values of fill-in and runtime are shown in **boldface**
speedup: the runtime compared with KLU

parallelism. It's also implemented with the default configurations. The results are shown in Table III. SuperLU_MT can also achieve speedups on some matrices in Set1, over KLU (however, SuperLU_MT fails on many matrices); and on these matrices, it also achieves relative speedups over its sequential algorithm ($P = 1$). However, on Set2 benchmarks, neither speedups nor relative speedups can be effectively achieved by SuperLU_MT.

So the conclusion is that not every matrix is suitable for parallel algorithm. For the matrices which have little numeric computation, the sequential algorithm should be used rather than the parallel algorithm. So each matrix can achieve the optimal performance. Our proposed predictive method, which performs the static symbolic factorization and calculates $\frac{NNZ_S}{NNZ_A}$, is performed very fast. For the 35 benchmarks, the runtime of the static symbolic factorization is only 1.2% (on average) of the sequential factorization time, and can be ignored.

VI. CONCLUSIONS

This paper has proposed a parallel LU factorization algorithm (with partial pivoting) by multi-thread parallelism. Since not every matrix is suitable for parallel algorithm, we propose a predictive method to decide whether a matrix should use sequential or parallel algorithm. By our adaptive solver, each matrix achieves the optimal performance. The experimental results show that the proposed algorithm achieves effective speedups compared with KLU, on the matrices which are suitable for parallel algorithm.

REFERENCES

[1] L. W. Nagel, "SPICE 2: A computer program to stimulate semiconductor circuits," *Ph.D. dissertation, University of California, Berkeley*, 1975.

[2] N. Kapre and A. DeHon, "Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs," *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pp. 190–198, dec. 2009.

[3] 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 Analysis and Applications*, vol. 20, no. 3, pp. 720–755, 1999.

[4] J. W. Demmel, J. R. Gilbert, and X. S. Li, "An asynchronous parallel supernodal algorithm for sparse gaussian elimination," *SIAM J. Matrix Analysis and Applications*, vol. 20, no. 4, pp. 915–952, 1999.

[5] X. S. Li and J. W. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Trans. Mathematical Software*, vol. 29, no. 2, pp. 110–140, June 2003.

[6] 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.

[7] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*, vol. 37, pp. 36:1–36:17, September 2010.

[8] T. A. Davis, "Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method," *ACM Trans. Math. Softw.*, vol. 30, pp. 196–199, June 2004.

[9] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, "Hybrid scheduling for the parallel solution of linear systems," *Parallel Computing*, vol. 32, no. 2, pp. 136–156, 2006.

[10] J. W. H. Liu, "The multifrontal method for sparse matrix solution: Theory and practice," *SIAM Review*, vol. 34, no. 1, pp. 82–109, 1992.

[11] O. Schenk and K. Gartner, "Solving unsymmetric sparse systems of linear equations with pardiso," *Computational Science - ICCS 2002*, vol. 2330, pp. 355–363, 2002.

[12] O. Schenk, K. Gartner, and W. Fichtner, "Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors," *BIT Numerical Mathematics*, vol. 40, pp. 158–176, 2000.

[13] 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," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 12, pp. 2868 –2881, dec. 2006.

[14] H. Thornquist, E. Keiter, R. Hoekstra, D. Day, and E. Boman, "A parallel preconditioning strategy for efficient transistor-level circuit simulation," *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pp. 410 –417, nov. 2009.

[15] N. Kapre, "SPICE2 - a spatial parallel architecture for accelerating the spice circuit simulator," Ph.D. dissertation, California Institute of Technology, 2010.

[16] T. Nechma, M. Zwolinski, and J. Reeve, "Parallel sparse matrix solver for direct circuit simulations on FPGAs," *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pp. 2358–2361, 2010.

[17] X. Chen, W. Wu, Y. Wang, H. Yu, and H. Yang, "An escheduler-based data dependence analysis and task scheduling for parallel circuit simulation," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 58, no. 10, pp. 702 –706, oct. 2011.

[18] 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. and Applics*, no. 4, pp. 889–901, 1997.

[19] 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, pp. 381–388, September 2004.

[20] J. R. Gilbert, "Predicting structure in sparse matrix computations," *SIAM J. Matrix Anal. Appl.*, vol. 15, pp. 62–79, January 1994.

[21] X. S. Li, "Sparse gaussian elimination on high performance computers," Ph.D. dissertation, Computer Science Division, UC Berkeley, Sep. 1996.

[22] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 3, 2011.