A New Scheduling Algorithm for Parallel Sparse LU Factorization with Static Pivoting*†

Laura Grigori Xiaoye S. Li
Lawrence Berkeley National Laboratory, MS 50F-1650
One Cyclotron Road, Berkeley, CA 94720, USA.
email: {lgrigori, xsli}@lbl.gov

Abstract

In this paper we present a static scheduling algorithm for parallel sparse LU factorization with static pivoting. The algorithm is divided into mapping and scheduling phases, using the symmetric pruned graphs of L^T and U to represent dependencies. The scheduling algorithm is designed for driving the parallel execution of the factorization on a distributed-memory architecture. Experimental results and comparisons with SuperLU_DIST are reported after applying this algorithm on real world application matrices on an IBM SP RS/6000 distributed memory machine.

1 Introduction

To factorize unsymmetric and non-definite matrices, many solvers rely on *partial pivoting* to maintain numerical stability. In that case, the structures of the L and U factors depend both on the structure of the initial A and on the row interchanges induced by partial pivoting. Thus the L and U structures cannot be determined *before* the numerical computation of the factors. In particular, this implies that some dynamically changing data structures should be used, together with a dynamic scheduling algorithm that identifies data dependencies while the numerical computation is in progress. In a distributed memory environment this approach has been observed not to scale so well [1].

A possible way to address the scalability issue is to first evaluate to what extent replacing the partial pivoting with other, more *static* techniques, can help maintain numerical stability. Li and Demmel show that such techniques are indeed possible [12]. Their proposed SuperLU_DIST solver uses a 2D distribution of the sparse matrix on a 2D grid of processors and is intended for large-scale distributed-memory machines. This solver is highly parallel and its ability to scale proportionally with the matrix size has been tested on a set of large matrices from various application domains. For almost all the matrices, the 2D distribution led to a very good load balance. However, exceptions have been noted with some matrices, for which a significant load imbalance was observed on 64 processors. Experiments using

*This work was supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy. The work of the first author is also supported by an INRIA post-doc abroad fellowship.

the 64 processors of a CRAY T3E [11] showed that the time spent in communication or synchronization represents a significant percentage of the total execution time. Even for the matrices for which the algorithm scales well up to 128 processors, more than 50% of the factorization time was spent waiting to receive messages.

These experiments suggest that further refinements are needed in order to improve the parallel efficiency. Several possibilities exist, among which the most relevant are: use more accurate information about the dependencies between computations, develop more sophisticated functions to map blocks to processors, consider additional scheduling techniques overlapping communication and computation.

Several mapping and scheduling algorithms have been proposed in the literature, mainly in the context of the Cholesky factorization. These algorithms use the *elimination tree* [14] to model the parallelism due to the sparsity of the matrix. An example of such an algorithm is the *subtree to subcube mapping* [7] which leads to good performance if the elimination tree is balanced. A generalization of this algorithm for unbalanced elimination tree is the *proportional mapping* algorithm, proposed by Pothen and Sun [15]. These algorithms map the columns to processors during a top-down traversal of the elimination tree, balancing the load while minimizing the communication.

Starting from these algorithms, a static scheduling algorithm was developed in the PaStiX solver [9]. This algorithm is based on two distinct phases. The first phase (partitioning phase) assigns to each supernode j a set of *candidate* processors. They are obtained by using the proportional mapping algorithm of Pothen and Sun [15]. The second phase (mapping phase) simulates the parallel factorization to order the tasks associated with the computation of each supernode. These tasks are assigned to a subset of processors from the set of candidate processors. Thus, the computation of each supernode will be assigned to only one processor (in 1D distribution) or several processors (in 2D distribution), depending on the workload associated to this supernode and the number of processors in the set of candidate processors.

In this paper we present a static scheduling algorithm for the sparse LU factorization. The main goal is to reduce as much as possible the communication, while balancing the load. Similar to the PaStiX solver, the algorithm uses the first phase to map the supernodes to processors, followed by the second phase to schedule the tasks. Our contributions are as follows. In contrast to most of the existing algorithms which use a tree to represent the dependencies between tasks, we use the symmetric pruned graphs of L^T and U to represent accurate dependencies between tasks. For the first phase, we develop a generalization of the proportional mapping, and the resulting algorithm represent one of the main contributions of the paper. For the second phase, we use a list scheduling heuristic. We show that this scheduling algorithm is especially effective to improve the scalability on large number of processors for very sparse matrices. Compared with the 2D mapping, the number of messages is significantly reduced from between 9023 and 50682 to between 615 and 3856 on 64 processors. Moreover, the proposed mapping and scheduling algorithm is fast, with its time complexity linear in the size of the input DAG, i.e., the symmetric pruned graph of U.

The rest of the paper is organized as follows: Section 2 introduces the dependency graphs for the LU factorization with static pivoting, and describes the scheduling algorithm for the graph to minimize the execution time. Experimental results and comparisons with SuperLU_DIST solver are presented in Section 3, followed by the conclusions and future work in Section 4.

2 The mapping and scheduling algorithm

Let a square matrix be partitioned into blocks of submatrices. This is usually obtained by partitioning the columns using the unsymmetric supernodes (columns of L with the same nonzero structure [2]). After that, the same partitioning is applied to the rows of the matrix to further break each supernode into blocks of submatrices. We denote by U_{kj} (L_{kj}) a submatrix of U (L) at row block index k and column block index j. For each column block, we identify two types of tasks. Task *Factor(k)* factorizes the column block k and exists for each $1 \leq k \leq N$. Task *Update(j, k)* updates block column k by block column j and exists for $j < k$ and $U_{jk} \neq 0$.[1] The sparse LU factorization algorithm can be described as:

for $k := 1$ to N **do**
  for $j := 1$ to $k-1$ with $U_{jk} \neq 0$ **do**
    Perform task *Update(j, k)*;
  end for
  Perform task *Factor (k)*;
end for

We use two phases to distribute the data and to schedule the computations, as in the PaStiX solver. In the first phase we assign a set of candidate processors to the computation of each supernode. In the second phase we schedule the computations in order to minimize the execution time. During the second phase, the computation of each supernode is scheduled on a processor from its candidate processors set.

The two phases of the scheduling algorithm use the dependencies between tasks. In the case of an unsymmetric matrix A, several tools can be used to represent the dependencies:

- the elimination tree of $A + A^T$ [14];
- the elimination DAGs of L^T and U [8];
- the symmetric pruned graphs of L^T and U [4, 5].

The elimination tree of $A + A^T$ is a structure offering simple manipulation and access. This tree can be used to represent the dependencies between supernodes, but it overestimates these dependencies. The elimination DAGs of L and U are the transitive reductions of the graphs of L and U, and are a compact way of representing *all* dependencies between computations, and *only* those dependencies. But a major disadvantage is their significant amount of construction time.

A good tradeoff is to use the symmetric pruned graphs of L^T and U. These graphs can be built very efficiently [4, 5]. Even if they introduce redundant dependencies compared to the elimination DAGs, we experimentally observed that they contain few redundant edges. Thus, they can be used effectively. However, due to their simplicity, we still use the elimination DAGs to develop the theoretical results. Knowing that all these results are also valid if used with the symmetric pruned graphs, our experimental results are based on the symmetric pruned graphs of L^T and U.

[1] The details of the *Factor()* and *Update()* tasks are not important in describing the algorithms in this paper.

2.1 Finding a set of candidate processors

The goal of the first phase is to assign a set of processors to each supernode, such that these processors can participate efficiently in the factorization of the supernode.

For symmetric matrices, the proportional mapping algorithm [15] attempts to map dependent computations on the same processor while balancing the load. This algorithm uses the elimination tree to represent the dependencies between supernodes. It starts by assigning all processors to the root. Then it considers the subtrees of the root in descending workload order. It assigns to each subtree a subset of processors proportional to the workload of the subtree. The algorithm continues until only one processor is assigned to each subtree. The advantage of this algorithm is that the communication in a subtree is restricted to only between the processors assigned to this subtree, thus ensuring a low communication cost.

For unsymmetric matrices, we are interested in using more accurate information on the dependencies between supernodes, and in this case the elimination DAG of U is helpful. This led us to consider a generalization of the proportional mapping algorithm to the DAGs. At each step of the algorithm, we consider a node i of the graph and its set of processors. Our goal is to assign to each predecessor j of node i a subset of i's processors, such that the communication between node i and its predecessors is minimized.

When the proportional mapping algorithm is applied to the elimination tree, each node j has only one successor i, and assigning to node j a subset of processors from the set of processors of i will reduce the communication. In the case of the elimination DAG, a node j has several successors, and it can communicate data to all these successors. The basic idea of our approach is to assign to node j several processors from its successors. We assign more processors from the successors with whom node j communicates more. To do so, we include a proportion of the workload associated with j to the workload associated with each of its successor i. This proportion depends on the communication between j and i.

Since we only want to consider communication between j and its immediate successors, a new problem arises. Some of the edge $j \to k$ in the graph of U disappears and occurs as a path of length greater than one in the elimination DAG (e.g., $j \to i \to k$). So node j needs to update its immediate sucessors and some of its ancestors in the elimination DAG. How do we estimate the communication between node j and its immediate successor i in the elimination DAG? We propose to approximate this as follows. We first compute total communication volume to node i, including all the incoming edges to i in the graph of U. We then divide this amount equally among i's immediate predecessors in the elimination DAG. In other words, all the communication from i's descendents are treated as the communication from i's immediate predecessors, so that the global communication information is retained even in the presence of graph contraction.

Algorithm 1 considers the floating-point operations to factorize a supernode (FSN), the number of predecessors of each supernode in the graph ($noPred$), the volume of communication involved in the factorization of supernode i ($avgCommIn[i]$), and the volume of communication from supernode j to its successors ($commOut[j]$). Using this information, the algorithm computes the workload associated with each supernode (WSN), taking into account the workload of its predecessors and the communication with its predecessors.

After computing the workload associated with each supernode, we can compute the set of candidate processors using Algorithm 2. This algorithm takes as input the dependency graph G, annotated with the workload and communication, and returns for each supernode

Algorithm 1 Compute the communication and the workload of each supernode

Input G: graph representing the dependencies between supernodes
for $i := 1$ to N **do**
  compute $CommIn[i]$ and $FSN[i]$;
  $avgCommIn[i] := CommIn[i]/noPred[i]$;
  for each predecessor j of i in G **do**
    commOut[j] += avgCommIn[i]
  end for
end for
for $i := 1$ to N **do**
  $WSN[i] := FSN[i]$;
  for each predecessor j of i in G **do**
    $WSN[i] += avgCommIn[i]/commOut[j] * WSN[j]$;
  end for
end for

a set of candidate processors. We add to the dependency graph G a sink supernode $N+1$, which is used to start the algorithm.

Algorithm 2 is a generalization of the proportional mapping algorithm of Pothen and Sun [15]. The main difference is that for each node i, it considers a proportion of the workload of each predecessor to assign a subset of its processors, taking into account the volume of communication. Note that if the input graph G is a tree, the algorithm is identical to the proportional mapping algorithm. If G is a tree and i is the parent of j in this tree, then $commOut[j]$ will be equal to $avgCommIn[i]$. Thus the workload associated with supernode i, denoted as $WSN[i]$, is equal to the sum of the workloads associated with all the nodes belonging to the subtree rooted at i, similar to the proportional mapping algorithm of Pothen and Sun.

Now we illustrate the execution of this algorithm on an example matrix B in Figure 1, where each column corresponds to a supernode and each nonzero element corresponds to a block.

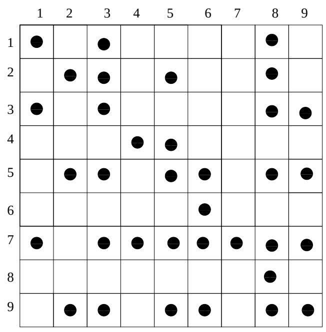

Figure 1: A supernodal matrix $B = L + U - I$

First, we illustrate the execution of the proportional mapping algorithm of Pothen and Sun in Figure 2. This figure shows the elimination tree of $B + B^T$, and each node of this tree corresponds to a supernode of the supernodal matrix B. At the right of each node i we label the information used by the proportional mapping algorithm, i.e. FSN represents the floating-point operations to compute supernode i, and WSN represents the load associated

Algorithm 2 Generalized proportional mapping algorithm for a DAG

Input: G: an annotated graph representing the dependencies and workload, E: set of processors
$proc[N+1] := E$;
for $i := 1$ to N **do**
  $proc[i] := \emptyset$;
end for
for $i := N+1$ to 1 **do**
  $S := proc[i]$, $\rho := |S|$;
  if i has predecessors **then**
    let $j_1, j_2, \ldots j_q$ be the predecessors of i, s.t. $WSN(j_1) \geq \ldots \geq WSN(j_q)$;
    $WSN[i] -= FSN[i]$;
    for $pred := j_1$ to j_q **do**
      $w := avgCommIn[i]/commOut[pred] \times WSN[pred]$;
      $k := w/WSN[i] \times \rho + 0.5$;
      if $k = 0$ or $S = \emptyset$ **then**
        find a processor p with the least workload from $proc[i]$;
        $proc[pred] \leftarrow proc[pred] \cup \{p\}$
      else
        if $k > |S|$ **then** $k := |S|$; **endif**
        $C :=$ a subset of k processors from S;
        $proc[pred] \leftarrow proc[pred] \cup C$
        $S \leftarrow S \setminus C$;
      end if
    end for
  end if
end for

with the subtree rooted at supernode i, which is the sum of the workloads of all the nodes belonging to the subtree rooted at i. At the left of each node i, we represent the set of candidate processors, which later can take part in the computation of supernode i. For example, node 5 disposes of 4 processors, namely P_0, P_1, P_2, P_3. This node has two sons, 3 and 4. The load of the subtree rooted at 3 represents 3/4 of the total load of the subtree rooted at 5, while the load of the subtree rooted at 4 represents 1/4 of this load. Thus, node 3 will receive three processors P_0, P_1, P_2, while node 4 will receive the processor P_3.

Second, we illustrate the execution of our proportional mapping algorithm on the same example matrix B in Figure 1. The elimination DAG of U is shown in Figure 3. Again, the nodes of this graph represent supernodes of B. In Figure 3 we label at the right of each node its corresponding values in the arrays WSN and FSN. These values are used in Algorithm 2. Moreover, $avgCommIn[i]$ is labeled on each edge to i, thus the value $commOut[j]$ is the sum of the outgoing edges from j. As an example, the value of the load WSN associated with node 8 is obtained by doing the sum $50 + 200 + (300 * 6/9) + (250 * 6/30) = 500$.

In Figure 4 we present the set of candidate processors assigned to each node by Algorithm 2. In this figure, node 10 was added as a starting point for the execution of the algorithm. Consider node 2. This node communicates mainly with its successor 3, thus it receives a processor from node 3 and no processor from node 5. Node 3 communicates roughly the same amount of data to its two successors, and thus it receives a processor from each one of its successors 8 and 9.

By comparing the results of the executions when using the elimination tree of $B + B^T$

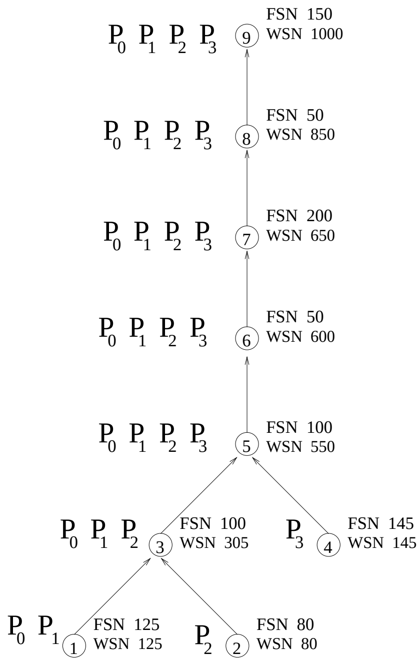

Figure 2: The elimination tree of $B + B^T$ corresponding to the matrix in Figure 1

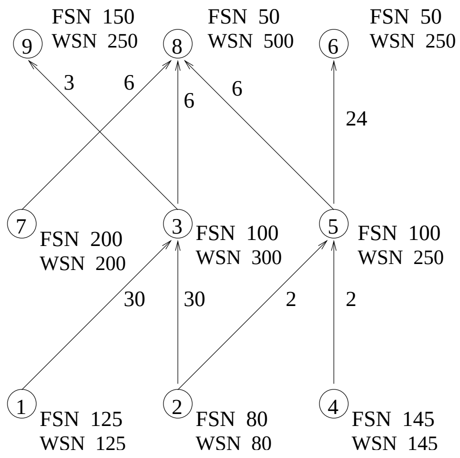

Figure 3: Elimination DAG of U corresponding to the matrix in Figure 1

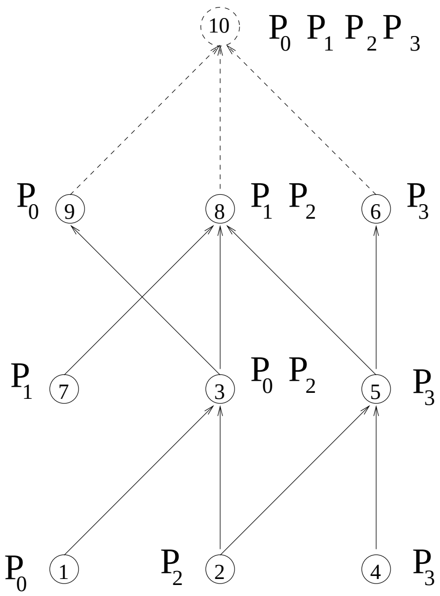

Figure 4: Illustration of Algorithm 2 on the elimination DAG from Figure 3

(Figure 2) and the elimination DAG of U (Figure 4), we notice that our approach using the elimination DAG tries to minimize the communication by considering the dependencies between supernodes and their communication. Node 3 in Figure 2 has received a subset of processors from node 5, even if there is no dependency between supernodes 5 and 3. This is due to the fact that the elimination tree of $A + A^T$ introduces false dependencies between supernodes.

2.2 Scheduling the computations

After the phase for mapping a supernode to a set of candidate processors, we perform scheduling. We use a list-based scheduling heuristic [6, 16] in this phase, which assigns the tasks associated with the computation of each supernode to a processor from the set of candidate processors.

A left-looking approach with a 1D distribution of the data is used: the computaion of a supernode k is assigned to one processor; the updates to supernode k (*Update(*, k)*) are executed just before the factorisation of this supernode (*Factor(k)*).

With this approach, the elimination DAG of U gives all the dependencies between supernodes computations, and hence it can be used as a dependency graph in the list scheduling algorithm. Each node of this graph corresponds to a supernode. An entry supernode is defined as a supernode with no incoming edges, while an exit supernode is defined as a supernode with no outgoing edges. During the scheduling, a supernode becomes ready when all its predecessors were scheduled.

Each supernode has a computation cost, which is the number of floating-point operations performed for this supernode. Each edge is assigned a communication cost, which is equal to the volume of data transferred between its two end supernodes. Using this information, we assign the priority of each supernode as the longest path from this supernode to an exit supernode. The length of the path is the sum of the communication and the computation costs of its constituent edges and supernodes.

For each processor we maintain a list of ready supernodes and the processor's start time. At each iteration of the scheduling loop, we select the processor with the earliest start time. From its list of ready supernodes, the supernode with the highest priority is selected (say supernode k), and needs to be assigned to one of its candidate processors that can execute it at the earliest time. Thus, we attempt to assign the supernode to each processor in its set of candidate processors, and determine on which processor the computation of this supernode can have the earliest start time. The supernode is then mapped on the selected processor and the tasks associated with its computation (tasks *Update(*, k)* and *Factor(k)* can be scheduled as soon as the computation of the last supernode assigned to that processor is finished. To overlap computations and communications, we consider a valid order between tasks *Update(*, k)*, and we schedule them in the order in which the data needed from its predecessors has arrived locally; we discuss the valid order further in this section.

The processor start time is updated as the finish time of the supernode scheduled on that processor. If a successor of the current supernode becomes ready for execution, then it is added to the ready list of every processor in its candidates set. The scheduling loop is repeated as long as there exist unscheduled supernodes.

A valid order between tasks *Update(j, k)* is given by the ascending order of the indices j. Using the elimination DAG of L^T, we can obtain a valid order which uses more accurate information on the dependencies between these tasks.

The following lemma gives the dependency between the updates from two supernodes i, i' to supernode k, where i, i' are linked by a path in the elimination DAG of L^T.

Lemma 1 *Consider supernodes i, i' and k such that i' is the successor of i in the elimination DAG of L^T and tasks Update(i, k), Update(i', k) exist. Then task Update(i, k) has to be completed before task Update(i', k) can start its execution.*

Proof As i' is the successor of i in the elimination DAG of L^T, then block $L_{i'i}$ is nonzero. As tasks *Update(i, k)*, *Update(i', k)* exist, we can deduce that blocks U_{ik}, $U_{i',k}$ are also nonzero.

Task *Update(i', k)* can begin its execution as soon as all the updates to block $U_{i'k}$ are finished. As task *Update(i, k)* is one of these updates, then this task must modify block $U_{i'k}$ before task *Update(i', k)* can start its execution. □

Let i, i' be two supernodes such that $i < i'$. If there is no path from i to i' in the elimination DAG of L^T, then block $L_{i'i}$ is zero. Consider another node k such that tasks *Update(i, k)*, *Update(i', k)* exist. Then there is no dependency between the two tasks, as task *Update(i, k)* does not modify block $U_{i'k}$ necessary for the execution of task *Update(i', k)*. From these results, the dependencies can be defined as follows :

- There is a task *Update(i, k)* for each $U_{ik} \neq 0$ and $1 \leq i < k \leq N$.
- There is a dependency from *Update(i, k)* to *Update(i', k)* if i' is the successor of i in the elimination DAG of L^T.
- There is a dependency from *Update(i, k)* to *Factor(k)* if i is an exit node, or the smallest successor j of i in the elimination DAG of L^T is no smaller than k.

We illustrate in Figure 5 these rules by computing the dependencies between the tasks associated with supernode 8 for the matrix in Figure 1. Before factorizing this supernode (task *Factor(8)* in the dependency graph), supernode 8 is updated by supernodes $1, 2, 3, 5$ and 7. A valid order of these updates is given by the ascending order of the source supernodes in the updates $1, 2, 3, 5, 7$. However, an order exhibiting more parallelism is the order using the elimination DAG. Hence, there is a dependency from *Update(1, 8)* to *Update(3, 8)* because 3 is the successor of 1 in the elimination DAG of L^T (the elimination DAG of L^T is presented in Figure 6). But there is no dependency between *Update(2, 8)* and *Update(3, 8)* because there is no path from 2 to 3 in the elimination DAG of L^T.

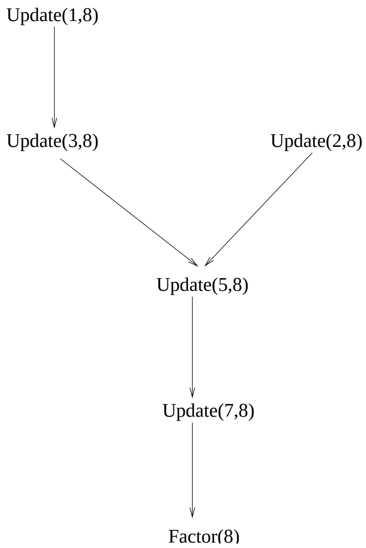

Figure 5: Illustration of several dependencies involved in the factorization of supernode 8 for the matrix in Figure 1

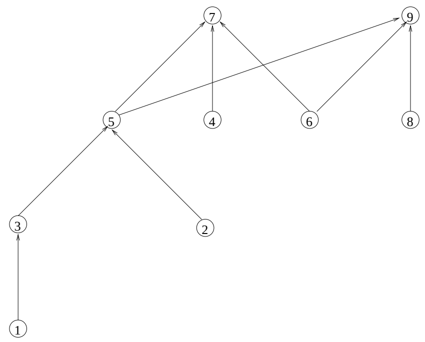

Figure 6: Elimination DAG of L^T corresponding to the matrix in Figure 1

3 Experimental results

In this section, we present the experimental results obtained when applying the new scheduling techniques on the real world matrices. We tested the new factorization method on an IBM SP RS/6000 distributed memory machine at NERSC. The system contains 2944 com-

Matrix	Order	$nnz(A)$	$nnz(L+U-I)$ $\times 10^6$	$Flops$ $\times 10^9$
af23560	23650	460598	12.8	5.41
bbmat	38744	1771722	36.2	27.80
ex11	16614	1096948	14.1	5.99
onetone1	62424	1717792	11.8	0.90
onetone2	36057	227628	1.3	0.23
rma10	46835	2374001	14.7	1.60
venkat01	62424	1717792	11.8	2.41
wang4	26064	177196	27.7	8.78

Table 1: Benchmark matrices.

Matrix	#snodes	#edges G(SNU)	#edges G(pr SNU)	#entries G(pr SNU)	#exits G(pr SNU)
af23560	10543	78962	10700	7482	1
bbmat	12726	214767	16944	5451	1
ex11	2597	23603	2596	678	1
onetone1	22370	123427	37323	10274	611
onetone2	21682	69098	27602	10426	611
rma10	7861	33265	7862	3045	1
venkat01	13570	61348	13569	4116	1
wang4	16302	89082	16301	11710	1

Table 2: Characteristics of the graphs. G(SNU) is the supernodal graph of U. G(pr SNU) is the symetric pruned supernodal graph of U.

pute processors distributed among 184 compute nodes. Each processor is clocked at 375 Mhz and has a peak performance of 1.5 GFlops. Each node has 16 to 64 Gbytes of shared memory. We used several medium and large matrices from a variety of application domains. These matrices and their characteristics are presented in table 1, which includes the matrix order, the number of nonzeros in the matrix A, the number of nonzeros in the factors L and U, and the number of floating-point operations.

Table 2 presents the size of the supernodal graph and its symmetric pruned graph. The second and the third columns list the number of nodes and edges in the supernodal graph of U. The fourth column lists the number of edges in the symmetric pruned graph of U. The fifth and the sixth columns list the number of entry and exit supernodes in the symmetric pruned graph of U. For all our test matrices, the supernodal symmetric pruned graph of U is much smaller than the supernodal graph of U. Very often, there are one order of magnitude fewer edges than in the supernodal graph of U. Three of the test matrices (`ex11, venkat01 and wang4`) are structurally symmetric, in which case the pruned graph is a tree. Computing the symmetric pruned graph takes very little time, and this time is included in the scheduling time overhead reported in Figure 7 and Table 5.

We now compare the performance of the new factorization algorithm (referred as SCHED) to the factorization algorithm in SuperLU_DIST (referred as SLUD). In particular, we compare the load balance, the amount of communication and the runtime.

		P = 4	P = 16	P = 32	P = 64	P = 128
af23560	SCHED	0.91	0.96	0.91	0.75	0.52
	SLUD	0.94	0.81	0.75	0.62	0.58
bbmat	SCHED	0.99	0.88	0.82	0.79	0.62
	SLUD	0.97	0.91	0.80	0.65	0.60
ex11	SCHED	0.98	0.85	0.69	0.54	0.32
	SLUD	0.97	0.93	0.83	0.67	0.52
onetone1	SCHED	0.94	0.74	0.49	0.48	0.24
	SLUD	0.86	0.83	0.66	0.50	0.45
onetone2	SCHED	0.83	0.46	0.40	0.31	0.16
	SLUD	0.81	0.64	0.59	0.40	0.21
rma10	SCHED	0.88	0.70	0.49	0.66	0.48
	SLUD	0.89	0.70	0.67	0.49	0.43
venkat01	SCHED	0.95	0.91	0.90	0.63	0.37
	SLUD	0.93	0.75	0.74	0.56	0.47
wang4	SCHED	0.98	0.90	0.76	0.87	0.52
	SLUD	0.99	0.91	0.86	0.75	0.60

Table 3: Load balance results.

For both algorithms, the preprocessing steps are the same. These include a step to permute large entries on the diagonal (using the routine MC64 [3]), followed by a symetric permutation to preserve the sparsity (using multiple minimum degree algorithm applied on $A + A^T$ [13]) and the symbolic factorization to get the structures of L and U. Only the numerical factorization phase is different in the two approaches. This includes the matrix distribution and the actual factorization. After the preprocessing steps, SLUD distributes the data among processors using a 2D block-cyclic distribution on a 2D grid of processors. Locally on the set of owner processors, each supernode of L is stored in a column oriented format, while each supernode of U is stored in a row oriented format. This storage scheme fits well the right-looking factorization. In SCHED, a master processor executes the scheduling algorithm, and then sends the necessary information to all the other processors to guide the numerical factorization. Each supernode is distributed on its owner processor, and is stored using a column oriented format, for both L and U. This storage is well adapted to the left-looking factorization. After the distribution, the numerical factotorization is performed using the valid task order established by the scheduling algorithm.

To evaluate the load balance, we consider the load associated with a processor as being the number of floating-point operations performed on this processor. As described in [11], the load balance factor can be computed as the average load divided by the maximum load among all the processors. Thus, the closer is this factor to 1, the better is the load balance. Table 3 shows the load balance factors. Compared with the 2D block-cyclic mapping, the proportional mapping algorithm usually improves load balance, with very few exceptions.

Table 4 compares the amount of communication of the two algorithms. For each matrix, we report the average communication volume and the average number of messages per processor. For all the test matrices, SCHED leads to a large reduction in the number of messages. Usually the average number of messages increases with the increasing number of processors up to 16 or 32, and then it starts decreasing. On the other hand, the volume of communication for SCHED is not always smaller than for SLUD. Sometimes SCHED is better and sometimes SLUD is better. But the difference is not dramatic. The worst

		P = 4	P = 16	P = 32	P = 64	P = 128
af23560	SCHED Vol	20.83	22.27	18.47	14.04	8.20
	SLUD Vol	27.27	21.25	16.38	13.01	8.13
	SCHED #Mess	5647	2474	1869	1465	1250
	SLUD #Mess	26185	36350	26632	33638	20301
bbmat	SCHED Vol	90.69	81.75	75.00	53.50	49.98
	SLUD Vol	81.53	62.67	51.23	38.09	27.77
	SCHED #Mess	9663	3931	3307	2469	2442
	SLUD #Mess	31719	45815	37640	46521	32686
ex11	SCHED Vol	25.72	25.98	23.15	13.57	9.71
	SLUD Vol	25.02	19.44	15.45	11.98	8.00
	SCHED #Mess	2435	1002	907	615	586
	SLUD #Mess	6486	9336	7247	9023	5639
onetone1	SCHED Vol	7.90	6.59	6.20	5.95	3.94
	SLUD Vol	7.56	5.82	4.16	3.56	2.26
	SCHED #Mess	46303	2859	3115	3856	2653
	SLUD #Mess	47639	57738	37322	50682	29359
onetone2	SCHED Vol	3.58	2.90	2.24	1.29	0.96
	SLUD Vol	4.16	3.25	2.20	2.05	1.20
	SCHED #Mess	14601	1934	2052	881	701
	SLUD #Mess	46173	53654	32701	45379	24176
rma10	SCHED Vol	13.64	8.37	6.69	5.13	3.33
	SLUD Vol	21.73	16.98	11.14	10.33	5.52
	SCHED #Mess	1861	839	719	566	469
	SLUD #Mess	19049	23743	14890	19717	10631
venkat01	SCHED Vol	14.95	9.86	8.50	7.29	4.04
	SLUD Vol	27.65	21.33	14.00	12.62	6.91
	SCHED #Mess	1505	545	632	627	428
	SLUD #Mess	33556	42675	26918	35343	19143
wang4	SCHED Vol	34.41	33.34	32.85	24.29	22.17
	SLUD Vol	24.59	18.91	14.78	11.47	8.07
	SCHED #Mess	12308	2703	2684	2038	2123
	SLUD #Mess	39470	50035	32983	43182	24582

Table 4: Average communication volume and number of messages per processor.

case is matrix `wang4`, for which the SCHED's communication volume is twice more than that of SLUD. These results imply that in SCHED, the message size is usually much bigger than that in SLUD. This is mainly because using the 1D distribution in SCHED, a message contains an entire supernode k of L. Whereas in SLUD, a message contains only a part of supernode k of L. Therefore, it is important to overlap computation with communication so to avoid the idle time waiting for the messages. This is well addressed in the new scheduling algorithm.

Finally, we compare the actual runtimes in Figure 7. (The runtimes are also tabulated in table 5.) Each plot in the figure corresponds to one matrix with varying number of processors. Since the two algorithms differ in matrix distribution and numerical factorization, we separately report the distribution time (labeled "dist") and the factorization time (labeled "fact"). We also report the total time which is the sum of the two. SCHED also needs to pay a small cost of scheduling overhead. We report this separately (labeled "schedule"). The reason we do not include this in the total time for SCHED is that when the matrices of the same nonzero structure are factorized multiple times, the scheduling algorithm will only be invoked once, and hence the cost is very small.

On smaller number of processors (less than 16), the distribution time for SCHED can be drastically smaller than that for SLUD, such as matrices `af23560, onetone1` and `onetone2`. This is due to the fact that storing both supernodes of L and U in a column oriented format can lead to a more efficient distribution algorithm. But with increasing number of processors, the distribution time for SCHED increases, while for SLUD it decreases. On one processor, the time difference is only in the distribution step. The factorization speed is about the same for both codes.

When comparing the total time of both distribution and factorization, SCHED is faster than SLUD for 6 matrices; it is more than twice faster for matrix `venkat01`. We observed more improvement on a large number of processors, where the total number of messages usually increases. In this case it is even more important to reduce the number of messages, and thus SCHED approach is effective. The time continues to decrease when increasing the number of processors up to 64. Beyond 64 processors, only the time for `bbmat` continues to decrease. This implies that this set of test matrices is not large enough to demonstrate scalability of the algorithms. In the future, we will test larger matrices.

For `ex11` and `bbmat`, SLUD is faster than SCHED. These two matrices are relatively denser than the other matrices. We suspect that these matrices exhibit limited amount of parallelism to be exploited in a left-looking algorithm with a 1D distribution. The 2D block-cyclic distribution used by SLUD can exploit more parallelism. As part of the future work, we will study the performance impact on denser matrices when using a 1D partition or a 2D partition (the 2D partition was shown to be more scalable for dense matrices). We will also evaluate the paralelism available in a left-looking algorithm versus a right-looking algorithm.

4 Conclusions and future work

In this paper we present a new assignment and static scheduling algorithm for sparse LU factorization with static pivoting. This algorithm uses the symmetric pruned graphs of L^T and U to represent the dependencies between computations, thus exploiting the parallelism due to the sparsity and asymmetry of the matrix. Experimental results show that our approach leads to a large reduction in the number of messages, and for very sparse matrices the performance compares favorably to that of the SuperLU_DIST solver on the IBM SP RS/6000 machine. Furthermore, the proposed scheduling algorithm is easy to implement and is fast, with its time complexity linear in the size of the input DAG, i.e., the symmetric pruned graph of U.

In an earlier work comparing SuperLU_DIST and MUMPS [1], it was found that MUMPS is faster for smaller numbers of processors (e.g., up to 64 on a Cray T3E), but SuperLU_DIST is faster for larger numbers of processors and shows better scalability. The new factorization algorithm SCHED in this paper usually performs better than SuperLU_DIST, especially for sparser matrices and larger machines. Therefore, it compares favorably with MUMPS for large numbers of processors.

Future work remains to improve the performance of the new approach and several avenues can be explored. A more accurate performance model should be developped in order to effectively use the list scheduling algorithm and to reduce the processor's idle time. More optimizations can be done to better overlap computation and communication. Methods for

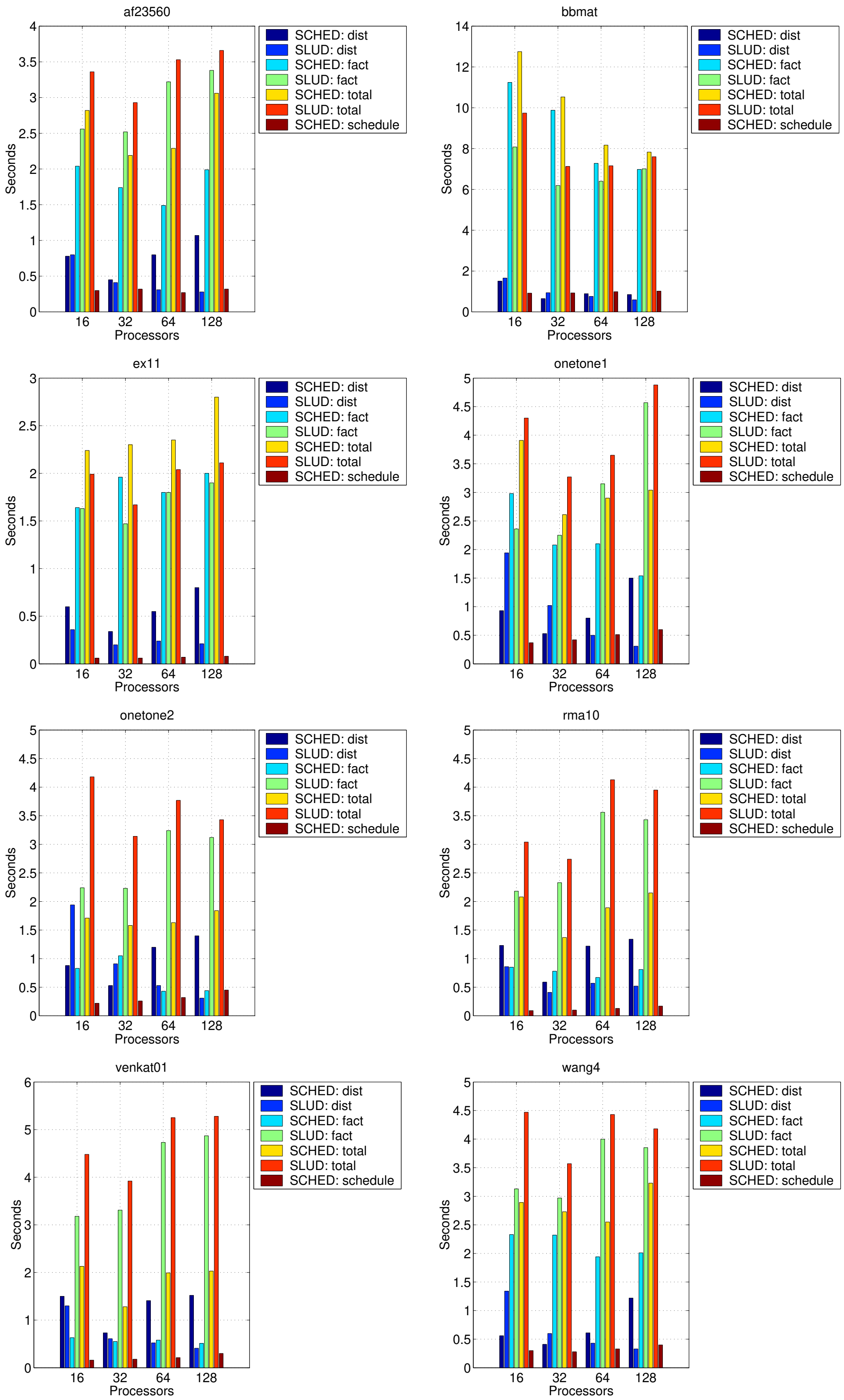

Figure 7: Comparison of the runtime between SCHED and SLUD.

		P = 1	P = 4	P = 16	P = 32	P = 64	P = 128
af23560	schedule	0.21	0.20	0.30	0.32	0.27	0.32
	SCHED	13.19	4.51	2.82	2.19	2.29	1.99
	SLUD	17.38	6.05	3.36	2.93	3.53	3.38
bbmat	schedule	0.90	0.88	0.92	0.93	0.99	1.02
	SCHED	75.75	34.28	12.75	9.53	8.17	7.83
	SLUD	82.78	24.27	9.74	7.13	7.16	7.60
ex11	schedule	0.10	0.05	0.06	0.06	0.07	0.08
	SCHED	11.55	4.20	2.24	2.30	2.35	2.80
	SLUD	11.14	3.96	1.99	1.67	2.04	2.11
onetone1	schedule	0.36	0.32	0.37	0.42	0.51	0.60
	SCHED	9.78	4.40	3.91	2.61	2.90	3.04
	SLUD	34.13	10.21	4.30	3.27	3.65	4.88
onetone2	schedule	0.27	0.20	0.22	0.26	0.32	0.45
	SCHED	3.07	1.73	1.71	1.58	1.63	1.84
	SLUD	27.27	8.88	4.18	3.14	3.77	3.43
rma10	schedule	0.19	0.09	0.09	0.10	0.13	0.17
	SCHED	6.45	2.38	2.08	1.37	1.89	2.15
	SLUD	8.43	3.94	3.04	2.74	4.13	3.95
venkat01	schedule	0.18	0.14	0.16	0.18	0.21	0.30
	SCHED	8.68	3.22	2.13	1.28	1.99	2.03
	SLUD	16.73	6.75	4.48	3.92	5.25	5.28
wang4	schedule	0.26	0.21	0.30	0.28	0.33	0.40
	SCHED	19.27	5.60	2.89	2.73	2.55	3.23
	SLUD	30.57	9.20	4.47	3.57	4.43	4.18

Table 5: Schedule time in seconds (schedule), total numerical factorization time in seconds (including data distribution time) on the IBM SP RS/6000.

controlling the memory requirement on each processor will be analyzed and implemented, which can improve the memory usage of the left-looking scheme.

To speed up the numerical factorization for denser matrices, we plan to extend our methods so that both 1D and 2D distributions will be used. During the list scheduling algorithm, the computation of each supernode will be assigned to one processor (in 1D distribution) or several processors (in 2D distribution), depending on the workload associated to this supernode and the number of processors in the set of candidate processors. Thus both task and data parallelism will be exploited in the program.

One final remark is about the choice of the algorithm – when to use SCHED and when to use SLUD, since both have merits. As we mentioned earlier, we think SCHED performs better for sparser problems. So we sort the matrices in terms of density, which is defined as $nnz(L + U)/n^2$, and plot the performance gain of SCHED over SLUD in figure 8. If our conjecture was true, each line would increase monotonously with increasing density. We somewhat see this trend, with some exceptions, such as matrices `af23560` and `wang4`, corresponding to the two dips in the plots. These two matrices are relatively dense, but SCHED performs better. We admit that the performance gain is a complex function of the input matrix and the two different algorithms. More factors than just sparsity affect the performance. It remains an open question to predict performance of different algorithms from the input matrix in order to help choose the right algorithm. For this, we plan to include a much larger set of matrices and analyze the global trend.

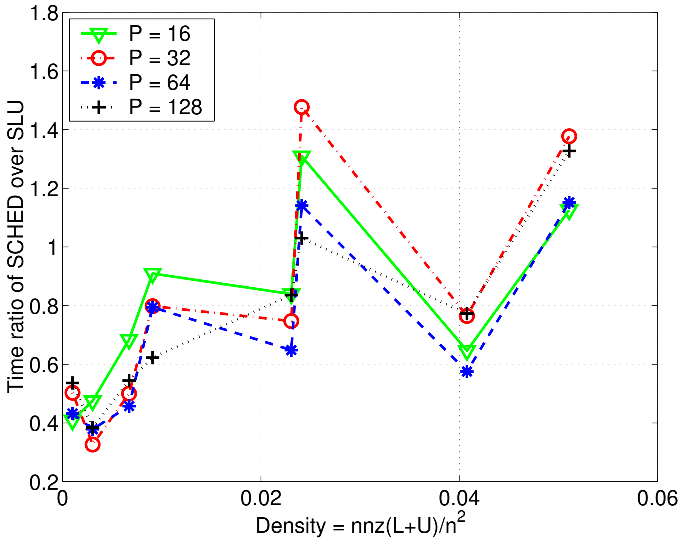

Figure 8: The ratio of the total time SCHED over SLUD.

References

[1] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Transactions on Mathematical Software*, 27(4):388–421, Dec. 2001.

[2] 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 Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[3] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Analysis and Applications*, 22(4):973–996, 2001.

[4] S. C. Eisenstat and J. W. H. Liu. Exploiting Structural Symmetry in Unsymmetric Sparse Symbolic Factorization. *SIAM J. Matrix Anal. Appl.*, 13(1):202–211, 1992.

[5] S. C. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in a sparse partial pivoting code. *SIAM Journal on Scientific Computing*, 14(1):253–257, January 1993.

[6] H. El-Rewini, H. H. Ali, and T. G. Lewis. Task scheduling in multiprocessing systems. *IEEE Computer*, pages 27–37, Dec. 1995.

[7] J. A. George, J. W. H. Liu, and E. G. Ng. Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Computing*, 10:287–298, 1989.

[8] J. R. Gilbert and J. W. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM J. Matrix Anal. Appl.*, 14(2):334–352, April 1993.

[9] P. Henon, P. Ramet, and J. Roman. Pastix: A parallel direct solver for sparse spd matrices based on efficient static scheduling and memory managment. In *SIAM Conference PPSC'2001, Portsmouth, Virginie, USA*, 2001.

[10] M. Joshi, G. Karypis, V. Kumar, A. Gupta, and F. Gustavson. PSPASES: Scalable Parallel Direct Solver Library for Sparse Symmetric Positive Definite Linear Systems. Technical report, University of Minnesota and IBM Thomas J. Watson Research Center, May 1999.

[11] X. S. Li and J. W. Demmel. Making Sparse Gaussian Elimination Scalable by Static Pivoting. *SuperComputing*, 1998.

[12] X. S. Li and J. W. Demmel. A Scalable Sparse Direct Solver Using Static Pivoting. *9th SIAM Conference on Parallel Processing and Scientific Computing*, 1999.

[13] J. W. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11:141–153, 1985.

[14] J. W. H. Liu. The Role of Elimination Trees in Sparse Factorization. *SIAM J. Matrix Annal. Appl.*, 11(1):134–172, 1990.

[15] A. Pothen and C. Sun. A Mapping Algorithm for Parallel Sparse Cholesky Factorization. *SIAM Journal on Scientific Computing*, pages 1253–1257, 1993.

[16] M. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. on Parallel and Distributed Systems*, 5(9):951–967, 1994.