Efficient Sparse LU Factorization with Partial Pivoting on Distributed Memory Architectures

Cong Fu, Xiangmin Jiao* and Tao Yang
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

A sparse LU factorization based on Gaussian elimination with partial pivoting (GEPP) is important to many scientific applications, but it is still an open problem to develop a high performance GEPP code on distributed memory machines. The main difficulty is that partial pivoting operations dynamically change computation and nonzero fill-in structures during the elimination process. This paper presents an approach called S^* for parallelizing this problem on distributed memory machines. The S^* approach adopts static symbolic factorization to avoid run-time control overhead, incorporates 2-D L/U supernode partitioning and amalgamation strategies to improve caching performance, and exploits irregular task parallelism embedded in sparse LU using asynchronous computation scheduling. The paper discusses and compares the algorithms using 1-D and 2-D data mapping schemes, and presents experimental studies on Cray-T3D and T3E. The performance results for a set of nonsymmetric benchmark matrices are very encouraging and S^* has achieved up to 6.878 GFLOPS on 128 T3E nodes. This is the highest performance known for this challenging problem and the previous record was 2.583 GFLOPS on shared memory machines [8].

Index Terms: Sparse LU factorization, Gaussian Elimination with partial pivoting, Dense structures, Asynchronous computation scheduling, Irregular parallelism.

1 Introduction

Sparse matrix factorization is an important approach to solving a sparse system of linear equations. If a matrix is symmetric and positive definite, Cholesky factorization can be used, for which fast sequential and parallel algorithms have been developed in [24, 30, 31]. However in many applications, the associated equation systems involve nonsymmetric matrices and pivoting may be required to maintain numerical stability for such nonsymmetric linear systems [6, 23]. Because

*Currently with the Computer Science Department, University of Illinois at Urbana-Champaign.

pivoting operations interchange rows based on the numerical values of matrix elements during the elimination process, it is impossible to predict the precise structures of L and U factors without actually performing the numerical factorization. The adaptive and irregular nature of sparse LU data structures makes an efficient implementation of this algorithm very hard even on a modern sequential machine with memory hierarchies.

There are several approaches that can be used for solving nonsymmetric systems. One approach is the unsymmetric-pattern multi-frontal method [5, 25] that uses elimination graphs to model irregular parallelism and guide the parallel computation. Another approach [19] is to restructure a sparse matrix into a bordered block upper triangular form and use a special pivoting technique which preserves the structure and maintains numerical stability at acceptable levels. This method has been implemented on Illinois Cedar multi-processors based on Aliant shared memory clusters. This paper focuses on parallelization issues for a given column ordering with row interchanges to maintain numerical stability. Parallelization of sparse LU with partial pivoting is also studied in [21] on a shared memory machine by using static symbolic LU factorization to overestimate nonzero fill-ins and avoid dynamic variation of LU data structures. This approach leads to good speedups for up to 6 processors on a Sequent machine and further work is needed to assess the performance of the sequential code.

As far as we know, there are no published results for parallel sparse LU on popular commercial distributed memory machines such as Cray-T3D/T3E, Intel Paragon, IBM SP/2, TMC CM-5 and Meiko CS-2. One difficulty in the parallelization of sparse LU on these machines is how to utilize a sophisticated uniprocessor architecture. The design of a sequential algorithm must take advantage of caching, which makes some previously proposed techniques less effective. On the other hand, a parallel implementation must utilize the fast communication mechanisms available on these machines. It is easy to get speedups by comparing a parallel code to a sequential code which does not fully exploit the uniprocessor capability, but it is not as easy to parallelize a highly optimized sequential code. One such sequential code is SuperLU [7] which uses a supernode approach to conduct sequential sparse LU with partial pivoting. The supernode partitioning makes it possible to perform most of the numerical updates using BLAS-2 level dense matrix-vector multiplications, and therefore to better exploit memory hierarchies. SuperLU performs symbolic factorization and generates supernodes on the fly as the factorization proceeds. UMFPACK is another competitive sequential code for this problem and neither SuperLU nor UMFPACK is always better than the other [3, 4, 7]. MA41 is a code for sparse matrices with symmetric patterns. All of them are regarded as of high quality and deliver excellent megaflop performance. In this paper we focus on the performance analysis and comparison with SuperLU code since the structure of our code is closer to that of SuperLU.

In this paper, we present an approach called S^* that considers the following key strategies together in parallelizing the sparse LU algorithm:

1. Adopt a static symbolic factorization scheme to eliminate the data structure variation caused by dynamic pivoting.

2. Identify data regularity from the sparse structure obtained by the symbolic factorization

scheme so that efficient dense operations can be used to perform most of computation and the impact of nonzero fill-in overestimation on overall elimination time is minimized.

3. Develop scheduling techniques for exploiting maximum irregular parallelism and reducing memory requirements for solving large problems.

We observe that on most current commodity processors with memory hierarchies, a highly optimized BLAS-3 subroutine usually outperforms a BLAS-2 subroutine in implementing the same numerical operations [6, 9]. We can afford to introduce some extra BLAS-3 operations in re-designing the LU algorithm so that the new algorithm is easy to be parallelized but the sequential performance of this code is still competitive to the current best sequential code. We use the static symbolic factorization technique first proposed in [20, 21] to predict the worst possible structures of L and U factors without knowing the actual numerical values, then we develop a 2-D L/U supernode partitioning technique to identify dense structures in both L and U factors, and maximize the use of BLAS-3 level subroutines for these dense structures. We also incorporate a supernode amalgamation [1, 10] technique to increase the granularity of the computation.

In exploiting irregular parallelism in the re-designed sparse LU algorithm, we have experimented with two mapping methods, one of which uses 1-D data mapping and the other uses 2-D data mapping. One advantage of using 1-D data mapping is that the corresponding LU algorithm can be easily modeled by directed acyclic task graphs (DAGs). Graph scheduling techniques and efficient run-time support are available to schedule and execute DAG parallelism [15, 16]. Scheduling and executing DAG parallelism is a difficult job because parallelism in sparse problems is irregular and execution must be asynchronous. The important optimizations are overlapping computation with communication, balancing processor loads and eliminating unnecessary communication overhead. Graph scheduling can do an excellent job in exploiting irregular parallelism but it leads to extra memory space per node to achieve the best performance. Also the 1-D data mapping can only expose limited parallelism. Due to these restrictions, we have also examined a 2-D data mapping method and an asynchronous execution scheme which exploits parallelism under memory constraints. We have implemented our sparse LU algorithms and conducted experiments with a set of nonsymmetric benchmark matrices on Cray-T3D and T3E. Our experiments show that our approach is quite effective in delivering good performance in terms of high megaflop numbers. In particular, the 1-D code outperforms the current 2-D code when processors have sufficient memory. But the 2-D code has more potential to solve larger problems and produces higher megaflop numbers.

The rest of the paper is organized as follows. Section 2 gives the problem definition. Section 3 describes structure prediction and 2-D L/U supernode partitioning for sparse LU factorization. Section 4 describes program partitioning and data mapping schemes. Section 5 addresses the asynchronous computation scheduling and execution. Section 6 presents the experimental results. Section 7 concludes the paper.

```
for k = 1 to n-1
        Find m such that |a_mk| = max_{k<=i<=n}{|a_ik|};
        if a_mk = 0, then A is singular, stop;
        Swap row k with row m;
        for i = k+1 to n with a_ik != 0
              a_ik = a_ik / a_kk;
        endfor
        for j = k+1 to n with a_kj != 0
              for i = k+1 to n with a_ik != 0
                    a_ij = a_ij - a_ik * a_kj;
              endfor
        endfor
endfor
```

Figure 1: Sparse Gaussian elimination with partial pivoting for LU factorization.

2 Preliminaries

Figure 1 shows how a nonsingular matrix A can be factored into two matrices L and U using GEPP. The elimination steps are controlled by loop index k. For elements manipulated at step k, we use i for row indexing and j for column indexing. This convention will be used through the rest of this paper. During each step of the elimination process, a row interchange may be needed to maintain numerical stability. The result of LU factorization process can be expressed by: $PA = LU$, where L is a unit lower triangular matrix, U is a upper triangular matrix, and P is a permutation matrix which contains the row interchange information. The solution of a linear system $Ax = b$ can hence be solved by two triangular solvers: $Ly = Pb$ and $Ux = y$. The triangular solvers are much less time consuming than the Gaussian elimination process.

Caching behavior plays an important role in achieving good performance for scientific computations. To better exploit memory hierarchy in modern architectures, supernode partitioning is an important technique to exploit the regularity of sparse matrix computations and utilize BLAS routines to speed up the computation. It has been successfully applied to Cholesky factorization [26, 30, 31]. The difficulty for the nonsymmetric factorization is that supernode structure depends on pivoting choices during the factorization thus cannot be determined in advance. SuperLU performs symbolic factorization and identifies supernodes on the fly. It also maximizes the use of BLAS-2 level operations to improve the caching performance of sparse LU. However, it is challenging to parallelize SuperLU on distributed memory machines. Using the precise pivoting information at each elimination step can certainly optimize data space usage, reduce communication and improve load balance, but such benefits could be offset by high run-time control and communication overhead.

The strategy of static data structure prediction in [20] is valuable in avoiding dynamic symbolic factorization, identifying the maximum data dependence patterns and minimizing dynamic control overhead. We will use this static strategy in our S^* approach. But the overestimation does introduce extra fill-ins and lead to a substantial amount of unnecessary operations in the numerical factorization. We observe that in SuperLU [7] the `DGEMV` routine (the BLAS-2 level dense matrix vector multiplication) accounts for 78% to 98% of the floating point operations (excluding the symbolic factorization part). It is also a fact that BLAS-3 routine `DGEMM` (matrix-matrix multiplication) is usually much faster than BLAS-1 and BLAS-2 routines [6]. On Cray-T3D with a matrix of size 25×25, `DGEMM` can achieve 103 MFLOPS while `DGEMV` only reaches 85 MFLOPS. Thus the key idea of our approach is that if we could find a way to maximize the use of `DGEMM` after using static symbolic factorization, even with overestimated nonzeros and extra numerical operations, the overall code performance could still be competitive to SuperLU which mainly uses `DGEMV`.

3 Storage prediction and dense structure identification

3.1 Storage prediction

The purpose of symbolic factorization is to obtain structures of L and U factors. Since pivoting sequences are not known until the numerical factorization, the only way to allocate enough storage space for the fill-ins generated in the numerical factorization phase is to overestimate. Given a sparse matrix A with a zero-free diagonal, a simple solution is to use the Cholesky factor L_c of $A^T A$. It has been shown that the structure of L_c can be used as an upper bound for the structures of L and U factors regardless of the choice of the pivot row at each step [20]. But it turns out that this bound is not very tight. It often substantially overestimates the structures of the L and U factors (refer to Table 1). Instead we consider another method from [20]. The basic idea is to statically consider all possible pivoting choices at each step. The space is allocated for all the possible nonzeros that would be introduced by *any* pivoting sequence that could occur during the numerical factorization. We summarize the symbolic factorization method briefly as follows.

The nonzero structure of a row is defined as a set of column indices at which nonzeros or fill-ins are present in the given $n \times n$ matrix A. Since the nonzero pattern of each row will change as the factorization proceeds, we use $\mathcal{R}_i^k$ to denote the structure of row i after step k of the factorization and A^k to denote the structure of the matrix A after step k. And a_{ij}^k denotes the element a_{ij} in A^k. Notice that the structures of each row or the whole matrix cover the structures of both L and U factors. In addition, during the process of symbolic factorization we assume that no exact numerical cancelation occurs. Thus, we have

$$\mathcal{R}_i^k = \{\ j \mid a_{ij}^k \text{ is structurally nonzero}\}.$$

We also define the set of candidate pivot rows at step k as follows:

$$\mathcal{P}_k = \{\ i \mid i \geq k, \text{and } a_{ik}^{k-1} \text{ is structurally nonzero}\}.$$

We assume that a_{kk} is always a nonzero. For any nonsingular matrix which does not have a zero-free diagonal, it is always possible to permute the rows of the matrix so that the permuted matrix has a zero-free diagonal [11]. Though the symbolic factorization does work on a matrix that contains zero entries in the diagonal, it is not preferable because it makes the overestimation too generous. The symbolic factorization process will iterate n steps and at step k, for each row $i \in \mathcal{P}_k$, its structure will be updated as:

$$\mathcal{R}_i^k = \mathcal{R}_i^{k-1} \bigcup (\bigcup_{j \in \mathcal{P}_k, j \neq i} \mathcal{R}_j^{k-1} - \{1, 2, \cdots, k-1\}).$$

Essentially the structure of each candidate pivot row at step k will be replaced by the union of the structures of all the candidate pivot rows except those column indices less than k. In this way it is guaranteed that the resulting structure A^n will be able to accommodate the fill-ins introduced by any possible pivot sequence. A simple example in Figure 2 demonstrates the whole process.

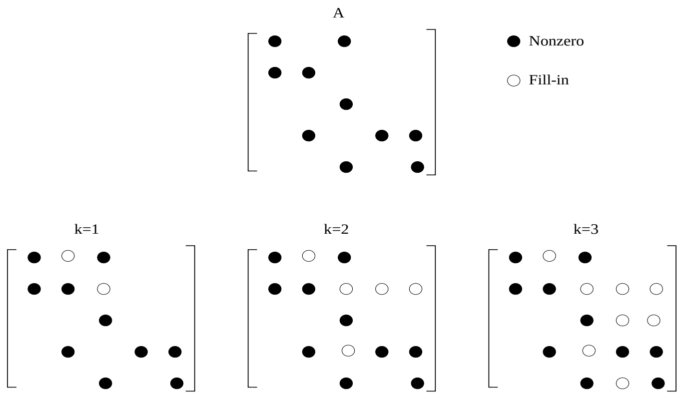

Figure 2: The first 3 steps of the symbolic factorization on a sample 5×5 sparse matrix. The structure remains unchanged at steps 4 and 5.

This symbolic factorization is applied after an ordering is performed on the matrix A to reduce fill-ins. The ordering we are currently using is the multiple minimum degree ordering for $A^T A$. We also permute the rows of the matrix using a transversal obtained from Duff's algorithm [11] to make A have a zero-free diagonal. The transversal can often help reduce fill-ins [12].

We have tested the storage impact of overestimation for a number of nonsymmetric testing matrices from various sources. The results are listed in Table 1. The fourth column in the table is original number of nonzeros, and the fifth column measures the symmetry of the structure of the original matrix. The bigger the symmetry number is, the more nonsymmetric the original matrix is. A unit symmetry number indicates a matrix is symmetric, but all matrices have nonsymmetric numerical values. We have compared the number of nonzeros obtained by the static approach and the number of nonzeros obtained by SuperLU, as well as that of the Cholesky factor of $A^T A$, for these matrices. The results in Table 1 show that the overestimation usually leads to less than 50% extra nonzeros than SuperLU scheme does. Extra nonzeros do imply additional computational cost. For example, one has to either check if a symbolic nonzero is an actual nonzero during a numerical factorization, or directly perform arithmetic operations which could be unnecessary. If we can aggregate these element-wise floating point operations and maximize the use of BLAS-3 subroutines, the sequential code performance will still be competitive. Even the fifth column of Table 1 shows that the floating operations from the overestimating approach can be as high as 5 times, the results in Section 6 will show that actual ratios of running times are much less. Thus it is necessary and beneficial to identify dense structures in a sparse matrix after the static symbolic factorization.

It should be noted that there are some cases that static symbolic factorization leads to excessive overestimation. For example, `memplus` matrix [7] is such a case. The static scheme produces 119 times as many nonzeros as SuperLU does. In fact, for this case, the ordering for SuperLU is applied based on $A^T + A$ instead of $A^T A$. Otherwise the overestimation ratio is 2.34 if using $A^T A$ for SuperLU also. For another matrix `wang3` [7], the static scheme produces 4 times as many nonzeros as SuperLU does. But our code can still produce 1 GFLOPS for it on 128 nodes of T3E. This paper focuses on the development of a high performance parallel code when overestimation ratios are not too high. Future work is to study ordering strategies that minimize overestimation ratios.

					factor entries/$\|A\|$			S^*/SuperLU	
Matrix	Identifier	Order	$\|A\|$	$\frac{\|A+A^T\|}{\|A\|}$	SuperLU	S^*	$A^T A$	$\|L\|+\|U\|$	ops
1	sherman5	3312	20793	1.220	12.03	15.70	20.42	1.31	3.41
2	lnsp3937	3937	25407	1.131	17.87	27.33	36.76	1.53	5.04
3	lns3937	3937	25407	1.131	18.07	27.92	37.21	1.55	4.74
4	sherman3	5005	20033	1.000	22.13	31.20	39.24	1.41	3.57
5	jpwh991	991	6027	1.053	23.55	34.02	42.57	1.44	3.47
6	orsreg1	2205	14133	1.000	29.34	41.44	52.19	1.41	3.76
7	saylr4	3564	22316	1.000	30.01	44.19	56.40	1.47	3.85
8	goodwin	7320	324772	1.358	9.63	10.80	16.00	1.12	3.11
9	e40r0100	17281	553562	1.000	14.76	17.32	26.48	1.17	3.11
10	ex11	16614	1096948	1.000	23.89	24.43	31.05	1.02	1.76
11	raefsky4	19779	1316789	1.000	20.36	28.06	35.68	1.38	2.81
12	inaccura	16146	1015156	1.000	9.79	12.21	16.47	1.25	2.90
13	af23560	23560	460598	1.054	30.39	44.39	57.40	1.46	3.16
14	vavasis3	41092	1683902	1.999	29.21	32.03	38.75	1.10	1.43

Table 1: Testing matrices and their statistics.

3.2 2-D L/U supernode partitioning and dense structure identification

Supernode partitioning is a commonly used technique to improve the caching performance of sparse code [2]. For a symmetric sparse matrix, a supernode is defined as a group of consecutive columns that have nested structure in the L factor of the matrix. Excellent performance has been achieved in [26, 30, 31] using supernode partitioning for Cholesky factorization. However, the above definition is not directly applicable to sparse LU with nonsymmetric matrices. A good analysis for defining unsymmetric supernodes in an L factor is available in [7]. Notice that supernodes may need to be further broken into smaller ones to fit into cache and to expose more parallelism. For the SuperLU approach, after L supernode partitioning, there are no regular dense structures in a U factor that could make it possible to use BLAS-3 routines (see Figure 3(a)). However in the S^* approach, there are dense columns (or subcolumns) in a U factor that we can identify after the static symbolic factorization (see Figure 3(b)). The U partitioning strategy is explained as follows. After an L supernode partition has been obtained on a sparse matrix A, i.e., a set of column blocks with possible different block sizes, the same partition is applied to the rows of the matrix to further break each supernode panel into submatrices. Now each off-diagonal submatrix in the L part is either a dense block or contains dense blocks. Furthermore, the following theorem identifies dense structure patterns in U factors. This is the key to maximizing the use of BLAS-3 subroutines in our algorithm.

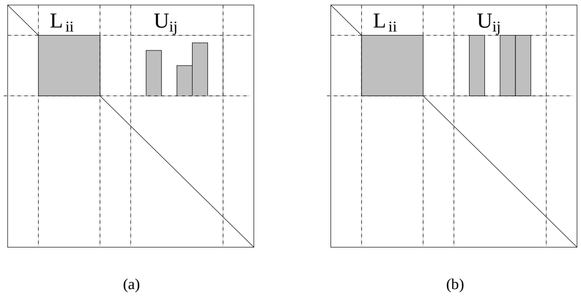

Figure 3: (a) An illustration of dense structures in a U factor in the SuperLU approach; (b) Dense structures in a U factor in the S^* approach.

In the following theorem, we show that the 2-D L/U partitioning strategy is successful and there is a rich set of dense structures to exploit. The following notations will be used through the rest of the paper.

- The L and U partitioning divides the columns of A into N column blocks and the rows of A into N row blocks so that the whole matrix is divided into $N \times N$ submatrices. For submatrices in the U factor, we denote them as U_{ij} for $1 \leq i < j \leq N$. For submatrices in the L factor, we denote them as L_{ij} for $1 \leq j \leq i \leq N$. If $i = j$, L_{ii} denotes the diagonal submatrix. We use A_{ij} to denote a submatrix when it is not necessary to distinguish between L and U factors.

- Define $S(i)$ as the starting column (or row) number of the i-th column (or row) block. For convenience, we define $S(N+1) = n+1$.

- A subcolumn (or subrow) is a column (or row) in a submatrix. For simplicity, we use a global column (or row) index to denote a subcolumn (or subrow) in a submatrix. For example, by subcolumn k in the submatrix block U_{ij}, it means the subcolumn in this submatrix with the global column index k where $S(j) \leq k < S(j+1)$. Similarly we use a_{ij} to indicate an individual nonzero element based on global indices. A *compound structure* in L or U is a submatrix, a subcolumn, or a subrow.

- A compound structure is nonzero if it contains at least one nonzero element or fill-in. We use $A_{ij} \neq 0$ to indicate that block A_{ij} is nonzero. Notice that an algorithm only needs to operate on nonzero compound structures. A compound structure is structurally dense if all of its elements are nonzeros or fill-ins. In the following we will not differentiate between nonzero and fill-in entries. They are all considered as nonzero elements.

Theorem 1 *Given a sparse matrix A with a zero-free diagonal, after the above static symbolic factorization and 2-D L/U supernode partitioning are performed on A, each nonzero submatrix in the U factor of A contains only structurally dense subcolumns.*

Proof: Recall that $\mathcal{P}_k$ is the set of candidate pivot rows at symbolic factorization step k. Given a supernode spanning from column k to $k+s$, from its definition and the fact that after step k the static symbolic factorization will only affect the nonzero patterns in submatrix $a_{k+1:n,k+1:n}$, and A has a zero-free diagonal, we have

$$\mathcal{P}_k \supset \mathcal{P}_{k+1} \supset \cdots \supset \mathcal{P}_{k+s}, \quad \text{and } \mathcal{P}_{k+s} \neq \phi.$$

Notice at each step k, the final structures of row i ($i \in \mathcal{P}_k$) are updated by the symbolic factorization procedure as

$$\mathcal{R}_i^k = \mathcal{R}_i^{k-1} \bigcup \Big(\bigcup_{j \in \mathcal{P}_k, j \neq i} \mathcal{R}_j^{k-1} - \{1, 2, \cdots, k-1\} \Big).$$

For the structure of a row i where $k \leq i \leq k+s$, we are only interested in nonzero patterns of the U part (excluding the part belonging to L_{kk}). We call this partial structure as $\mathcal{UR}_i$. Thus for $i \in \mathcal{P}_k$:

$$\mathcal{UR}_i^k = \bigcup_{j \in \mathcal{P}_k} \mathcal{UR}_j^{k-1}.$$

It can be seen that after the k-th step updating, $\mathcal{UR}_i^k = \mathcal{UR}_k^k$ for $i \in \mathcal{P}_k$. Knowing that the structure of row k is unchanged after step k, we only need to prove that $\mathcal{UR}_k^k = \mathcal{UR}_{k+1}^{k+1} = \cdots = \mathcal{UR}_{k+s}^{k+s}$ as shown below. Then we can infer that the nonzero structures of rows from k to $k+s$ are same and subcolumns at the U part are either structurally dense or zero. Now since $\mathcal{P}_k \supset \mathcal{P}_{k+1}$, and $\mathcal{P}_{k+1} \neq \phi$, it is clear that:

$$\mathcal{UR}_{k+1}^{k+1} = \bigcup_{j \in \mathcal{P}_{k+1}} \mathcal{UR}_j^k = \bigcup_{j \in \mathcal{P}_{k+1}} \mathcal{UR}_k^k = \mathcal{UR}_k^k.$$

Similarly we can show that $\mathcal{UR}_{k+s}^{k+s} = \mathcal{UR}_{k+s-1}^{k+s-1} = \cdots = \mathcal{UR}_k^k$. ■

The above theorem shows that the L/U partitioning can generate a rich set of structurally dense subcolumns or even structurally dense submatrices in a U factor. We also further incorporate this result with supernode amalgamation in Section 3.3 and our experiments indicate that more than 64% of numerical updates is performed by the BLAS-3 routine `DGEMM` in S^*, which shows the effectiveness of the L/U partitioning method. Figure 4 demonstrates the result of a supernode partitioning on a 7×7 sample sparse matrix. One can see that all the submatrices in the upper triangular part of the matrix only contain structurally dense subcolumns.

Based on the above theorem, we can further show a structural relationship between two submatrices in the same supernode column block, which will be useful in implementing our algorithm to detect nonzero structures efficiently for numerical updating.

Corollary 1 *Given two nonzero submatrices U_{ij}, $U_{i'j}$ where $i < i' < j$ and $L_{i'i} \neq 0$, if subcolumn k in U_{ij} is structurally dense, then subcolumn k in $U_{i'j}$ is also structurally dense.*

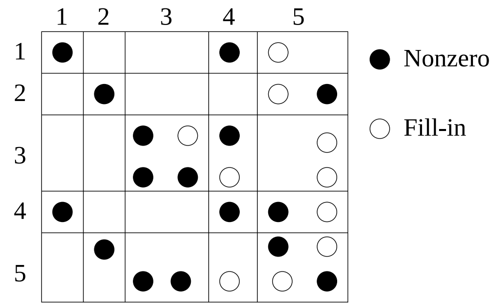

Figure 4: An example of L/U supernode partitioning.

Proof: The corollary is illustrated in Figure 5. Since $L_{i'i}$ is nonzero, there must be a structurally dense subrow in $L_{i'i}$. This will lead to a nonzero element in the subcolumn k in $U_{i'j}$ because the subcolumn k of U_{ij} is structurally dense. According to Theorem 1, subcolumn k in $U_{i'j}$ is structurally dense. ■

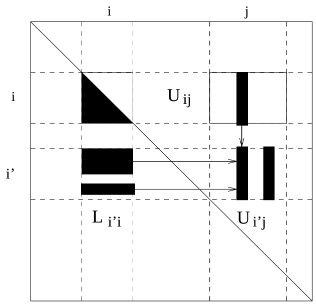

Figure 5: An illustration for Corollary 1.

Corollary 2 *Given two nonzero submatrices U_{ij}, $U_{i'j}$ where $i < i' < j$ and $L_{i'i}$ is nonzero. If U_{ij} is structurally dense, $U_{i'j}$ must be structurally dense.*

Proof: That is straightforward using Corollary 1. ■

3.3 Supernode amalgamation

For most tested sparse matrices, the average size of a supernode after L/U partitioning is very small, about 1.5 to 2 columns. This results in very fine grained tasks. Amalgamating small supernodes can lead to great performance improvement for both parallel and sequential sparse codes because it can improve caching performance and reduce interprocessor communication overhead.

There could be many ways to amalgamate supernodes [7, 30]. The basic idea is to relax the restriction that all the columns in a supernode must have exactly the same nonzero structure below diagonal. The amalgamation is usually guided by a supernode elimination tree. A parent could be merged with its children if the merging does not introduce too many extra zero entries into a supernode. Row and column permutations are needed if the parent is not consecutive with its children. However, a column permutation introduced by the above amalgamation method could undermine the correctness of the static symbolic factorization. We have used a simpler approach that does not require any permutation. This approach only amalgamates consecutive supernodes if their nonzero structures only differ by a small number of entries and it can be performed in a very efficient manner which only has a time complexity of $O(n)$ [27]. We can control the maximum allowed differences by an amalgamation factor r. Our experiments show that when r is in the range of $4-6$, it gives the best performance for the tested matrices and leads to $10-55\%$ improvement on the execution times of the sequential code. The reason is that by getting bigger supernodes, we are getting larger dense structures, although there may be a few zero entries in them, and we are taking more advantage of BLAS-3 kernels. Notice that after applying the supernode amalgamation, the dense structures identified in the Theorem 1 are not strictly dense any more. We call them *almost-dense* structures and can still use the result of Theorem 1 with a minor revision. That is summarized in the following corollary. All the results presented in Section 6 are obtained using this amalgamation strategy.

Corollary 3 *Given a sparse matrix A, if supernode amalgamation is applied to A after the static symbolic factorization and 2-D L/U supernode partitioning are performed on A, each nonzero submatrix in the U factor of A contains only almost-structurally-dense subcolumns.*

4 Program partitioning, task dependence and processor mapping

After dividing a sparse matrix A into submatrices using the L/U supernode partitioning, we need to partition the LU code accordingly and define coarse grained tasks that manipulate on partitioned dense data structures.

Program partitioning. Column block partitioning follows supernode structures. Typically there are two types of tasks. One is $Factor(k)$, which is to factorize all the columns in the k-th column block, including finding the pivoting sequence associated with those columns. The other is $Update(k, j)$, which is to apply the pivoting sequence derived from $Factor(k)$ to the j-th column block, and modify the j-th column block using the k-th column block, where $k < j$ and $U_{kj} \neq 0$. Instead of performing the row interchange to the right part of the matrix right after each pivoting search, a technique called "delayed-pivoting" is used [6]. In this technique, the pivoting sequence is held until the factorization of the k-th column block is completed. Then the pivoting sequence is applied to the rest of the matrix, i.e., interchange rows. Delayed-pivoting is important, especially to the parallel algorithm, because it is equivalent to aggregating multiple small messages into a larger one. Here the owner of the k-th column block sends the column block packed together with the pivoting information to other processors.

An outline of the partitioned sparse LU factorization algorithm with partial pivoting is described in Figure 6. The code of $Factor(k)$ is summarized in Figure 7. It uses BLAS-1 and BLAS-

2 subroutines. The computational cost of the numerical factorization is mainly dominated by *Update*() tasks. The function of task $Update(k, j)$ is presented in Figure 8. The lines `(05) and (12)` are using dense matrix multiplications.

```
for k = 1 to N
    Perform task Factor(k);
    for j = k+1 to N with U_kj ≠ 0
        Perform task Update(k, j);
    endfor
endfor
```

Figure 6: A partitioned sparse LU factorization with partial pivoting.

```
Factor(k)
  for m = S(k) to S(k+1) - 1
    Find the pivoting row t in column m;
    Interchange row t and row m of the column block k;
    Scale column m and update rest of columns in this column block;
  endfor
```

Figure 7: The description of task $Factor(k)$.

We use directed acyclic task graphs (DAGs) to model irregular parallelism arising in this partitioned sparse LU program. The DAGs are constructed statically before numerical factorization. Previous work on exploiting task parallelism for sparse Cholesky factorization has used elimination trees (e.g. [28, 30]), which is a good way to expose the available parallelism because pivoting is not required. For sparse LU, an elimination tree of $A^T A$ does not directly reflect the available parallelism. Dynamically created DAGs have been used for modeling parallelism and guiding run-time execution in a nonsymmetric multi-frontal method [5, 25].

Given the task definitions in Figures 6, 7 and 8 we can define the structure of a sparse LU task graph in the following.

These four properties are necessary.

- There are N tasks $Factor(k)$, where $1 \leq k \leq N$.
- There is a task $Update(k, j)$ if $U_{kj} \neq 0$, where $1 \leq k < j \leq N$. For a dense matrix, there will be a total of $N(N-1)/2$ updating tasks.
- There is a dependence edge from $Factor(k)$ to task $Update(k, j)$, where $k < j$ and $U_{kj} \neq 0$.

```
Update(k, j)
    Interchange rows according to the pivoting sequence;
    Let L_t be the lower triangular part of L_kk;
    if the submatrix U_kj is dense
       U_kj = L_t^{-1} * U_kj;
    else for each dense subcolumn c_u of U_kj
            c_u = L_t^{-1} * c_u;
         endfor

    for each nonzero submatrix A_ij, i > k in column block j
       if the submatrix U_kj is dense
          A_ij = A_ij - L_ik * U_kj;
       else for each dense subcolumn c_u of U_kj
                Let c_b be the corresponding dense subcolumn of A_ij;
                c_b = c_b - L_ik * c_u;
            endfor
    endfor
```

Figure 8: A description of task $Update(k, j)$.

- There is a dependence from $Update(k, k')$ to $Factor(k')$, where $k < k'$, $U_{kk'} \neq 0$ and there exists no task $Update(t, k')$ such that $k < t < k'$.

We add one more property, that while not necessary, simplifies implementation. This property essentially does not allow exploiting commutativity among $Update()$ tasks. However, according to our experience with Cholesky factorization [16], the performance loss due to this property is not substantial, about 6% in average when graph scheduling is used.

- There is a dependence from $Update(k, j)$ to $Update(k', j)$, where $k < k'$ and there exists no task $Update(t, j)$ such that $k < t < k'$.

Figure 9(a) shows the nonzero pattern of the partitioned matrix shown in Figure 4. Figure 9(b) is the corresponding task dependence graph.

1-D data mapping. In the 1-D data mapping, all submatrices, from both L and U part, of the same column block will reside in the same processor. Column blocks are mapped to processors in a cyclic manner or based on other scheduling techniques such as graph scheduling. Tasks are assigned based on owner-compute rule, i.e., tasks that modify the same column block are assigned to the same processor that owns the column block.

One disadvantage of this mapping is that it serializes the computation in a single $Factor(k)$ or $Update(k, j)$ task. In other words, a single $Factor(k)$ or $Update(k, j)$ task will be performed by

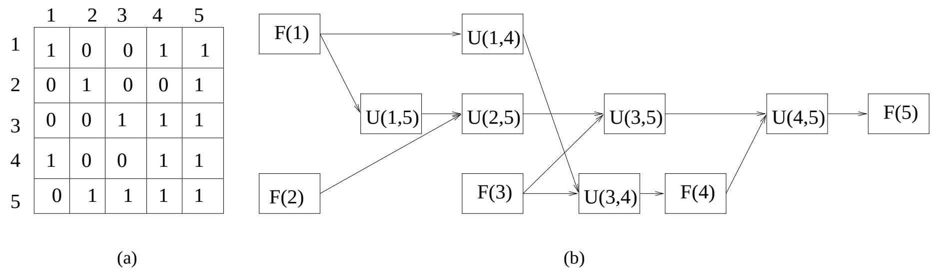

Figure 9: (a) The nonzero pattern for the example matrix in Figure 4. (b) The dependence graph derived from the partitioning result. For convenience, $F()$ is used to denote $Factor()$, $U()$ is used to denote $Update()$.

one processor. But this mapping strategy has an advantage that both pivot searching and subrow interchange can be done locally without any communication. Another advantage is that parallelism modeled by the above dependence structure can be effectively exploited using graph scheduling techniques.

2-D data mapping. In the literature 2-D mapping has been shown more scalable than 1-D for sparse Cholesky [30, 31]. However there are several difficulties to apply the 2-D block-oriented mapping to the case of sparse LU factorization even the static structure is predicted. Firstly, pivoting operations and row interchanges require frequent and well-synchronized interprocessor communication when submatrices in the same column block are assigned to different processors. Effective exploitation of limited irregular parallelism in the 2-D case requires a highly efficient asynchronous execution mechanism and a delicate message buffer management. Secondly, it is difficult to utilize and schedule all possible irregular parallelism from sparse LU. Lastly, how to manage a low space complexity is another issue since exploiting irregular parallelism to a maximum degree may need more buffer space.

Our 2-D algorithm uses a simple standard mapping function. In this scheme, p available processors are viewed as a two dimensional grid: $p = p_r \times p_c$. A nonzero submatrix block A_{ij} (could be an L block or a U block) is assigned to processor $P_{i \bmod p_r,\, j \bmod p_c}$. The 2-D data mapping is considered more scalable than 1-D data mapping because it enables parallelization of a single $Factor(k)$ or $Update(k, j)$ task on p_r processors. We will discuss how 2-D parallelism is exploited using asynchronous schedule execution.

5 Parallelism exploitation

5.1 Scheduling and run-time support for 1-D methods

We discuss how 1-D sparse LU tasks are scheduled and executed so that parallel time can be minimized. George and Ng [21] used a dynamic load balancing algorithm on a shared memory machine. For distributed memory machines, dynamic and adaptive load balancing works well for problems with very coarse grained computations, but it is still an open problem to balance the benefits of dynamic scheduling with the run-time control overhead since task and data migration cost is too expensive for sparse problems with mixed granularities. We use task dependence graphs to guide scheduling and have investigated two types of scheduling schemes.

- **Compute-ahead scheduling (CA).** This is to use block-cyclic mapping of tasks with a compute-ahead execution strategy, which is demonstrated in Figure 10. This idea has been used to speed up parallel dense factorizations [23]. It executes the numerical factorization layer by layer based on the current submatrix index. The parallelism is exploited for concurrent updating. In order to overlap computation with communication, the $Factor(k+1)$ task is executed as soon as $Factor(k)$ and $Update(k, k+1)$ (if exists) finish so that the pivoting sequence and column block $k+1$ for the next layer can be communicated as early as possible.
- **Graph scheduling**. We order task execution within each processor using the graph scheduling algorithms in [36]. The basic optimizations are balancing processor loads and overlapping computation with communication to hide communication latency. These are done by utilizing global dependence structures and critical path information.

```
if column block 1 is local
    Perform task Factor(1);
    Broadcast column block 1 and the pivoting sequence;
for k = 1 to N − 1
    if column block k + 1 is local
       if U_{k,k+1} ≠ 0
          Receive column block k and the pivoting choices;
          Interchange rows according to the pivoting sequence;
          Perform task Update(k, k + 1);
       Perform task Factor(k + 1);
       Broadcast column block k + 1 and the pivoting sequence;
    for j = k + 2 to N with U_{kj} ≠ 0
       if column block j is local
          if column block k has not been received
             Receive column block k and the pivoting choices;
          Interchange rows according to the pivoting sequence;
          Perform task Update(k, j);
    endfor
endfor
```

Figure 10: The 1-D code using compute-ahead schedule.

Graph scheduling has been shown effective in exploiting irregular parallelism for other applications (e.g. [15, 16]). Graph scheduling should outperform the CA scheduling for sparse LU because it does not have a constraint in ordering $Factor()$ tasks. We demonstrate this point using the LU task graph in Figure 9. For this example, the Gantt charts of the CA schedule and the schedule derived by our graph scheduling algorithm are listed in Figure 11. It is assumed that each task has a computation weight 2 and each edge has communication weight 1. It is easy to see that our scheduling approach produces a better result than the CA schedule. If we look at the CA schedule carefully, we can see that the reason is that CA can look ahead only one step so that the execution of task $Factor(3)$ is placed after $Update(1, 5)$. On the other hand, the graph scheduling algorithm detects that $Factor(3)$ can be executed before $Update(1, 5)$ which leads to better overlap of communication with computation.

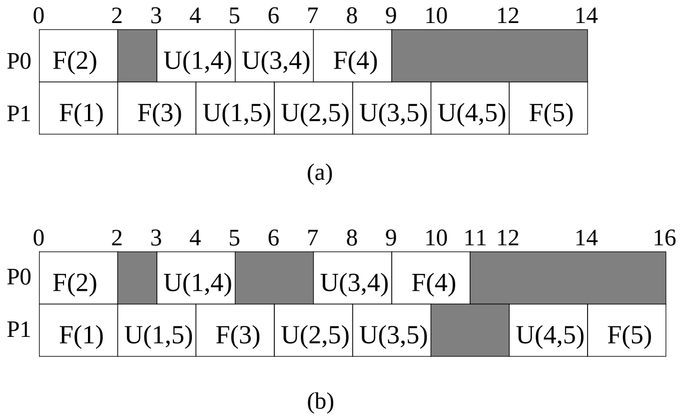

Figure 11: (a) A schedule derived by our graph scheduling algorithm. (b) A compute-ahead schedule. For convenience $F()$ is used to denote $Factor()$, $U()$ is used to denote $Update()$.

However the implementation of the CA algorithm is much easier since the efficient execution of a sparse task graph schedule requires a sophisticated run-time system to support asynchronous communication protocols. We have used the RAPID run-time system [16] for the parallelization of sparse LU using graph scheduling. The key optimization is to use Remote Memory Access(RMA) to communicate a data object between two processors. It does not incur any copying/buffering during a data transfer since low communication overhead is critical for sparse code with mixed granularities. RMA is available in modern multi-processor architectures such as Cray-T3D [34], T3E [32] and Meiko CS-2 [15]. Since the RMA directly writes data to a remote address, it is possible that the content at the remote address is still being used by other tasks and then the execution at the remote processor could be incorrect. Thus for a general computation, a permission to write the remote address needs to be obtained before issuing a remote write. However in the RAPID system, this hand-shaking process is avoided by a carefully designed task communication protocol [16]. This property greatly reduces task synchronization cost. As shown in [17], the RAPID sparse code can deliver more than 70% of the speedup predicted by the scheduler on Cray-T3D. In addition, using RAPID system greatly reduces the amount of implementation work to parallelize sparse LU.

```
Let (my_rno, my_cno) be the 2-D coordinates of this processor;
if my_cno = 1 then Perform task Factor(1);
for k = 1 to N - 1
    Perform ScaleSwap(k);
    if my_cno = (k + 1) mod p_c
        Perform Update_2D(k, k + 1);
        Perform Factor(k + 1);
    for j = k + 2 to N
        if my_cno = j mod p_c
            Perform Update_2D(k, j);
    endfor
endfor
```

Figure 12: The SPMD code of 2-D asynchronous code.

5.2 Asynchronous execution for the 2-D code

As we discussed previously, 1-D data mapping can not expose parallelism to a maximum extent. Another issue is that a time-efficient schedule may not be space-efficient. Specifically, to support concurrency among multiple updating stages in both RAPID and CA code, multiple buffers are needed to keep pivoting column blocks of different stages on each processor. Therefore for a given problem, the per processor space complexity of the 1-D codes could be as high as $O(S_1)$, where S_1 is the space complexity for a sequential algorithm. For sparse LU, each processor in the worst case may need a space for holding the entire matrix. The RAPID system [16] also needs extra memory space to hold dependence structures.

Based on the above observation, our goal for the 2-D code is to reduce memory space requirement while exploiting a reasonable amount of parallelism so that it can solve large problem instances in an efficient way. In this section, we present an asynchronous 2-D algorithm which can substantially overlap multi-stages of updating but its memory requirement is much smaller than that of 1-D methods. Figures 12 shows the main control of the algorithm in an SPMD coding style. Figure 13 shows the SPMD code for $Factor(k)$ which is executed by processors of column $k \bmod p_c$. Recall that the algorithm uses 2-D block-cyclic data mapping and the coordinates for the processor that owns submatrix A_{ij} are $(i \bmod p_r,\ j \bmod p_c)$. Also we divide the function of *Update*() (in Figure 8) into two parts: $ScaleSwap()$ which does scaling and delayed row interchange for submatrix $A_{k:N,\ k+1:N}$ as shown in Figure 14; *Update_2D*() which does submatrix updating as shown in Figure 15. In all figures, the statements which involve interprocessor communication are marked with $*$.

It can be seen that the **computation flow** of this 2-D code is still controlled by the pivoting tasks $Factor(k)$. The order of execution for $Factor(k)$, $k = 1, 2, \cdots, N$ is sequential, but *Update_2D*() tasks, where most of the computation comes from, can execute in parallel among all processors. The asynchronous parallelism comes from two levels. First a single stage of tasks $Update\_2D(k,\ k{+}1 : N)$

(01) $Factor(k)$
(02) **for** $m = S(k)$ to $S(k+1)-1$
(03) Find out local maximum element of column m;
(04) **if** $my\_rno \neq k$ mod p_r
(05)* Send the subrow within column block k containing the local maximum to processor $P_{k \bmod p_r,\ k \bmod p_c}$;
(06) **if** this processor owns L_{kk}
(07)* Collect all local maxima and find the pivot row t;
(08)* Broadcast the subrow t within column block k along this processor column and interchange subrow t and subrow m if necessary.
(09) Scale local entries of column m;
(10) Update local subcolumns from column $m+1$ to $S(k+1)-1$;
(11) **endfor**
(12)* Multicast the pivot sequence along this processor row;
(13)* **if** this processor owns L_{kk} **then** Multicast L_{kk} along this processor row;
(14)* Multicast the part of nonzero blocks in $L_{k+1:N,\ k}$ owned by this processor along this processor row;

Figure 13: Parallel execution of $Factor(k)$ for the 2-D asynchronous code.

can be executed concurrently on all processors. In addition, different stages of $Update\_2D()$ tasks from $Update\_2D(k,\ k+1:N)$ and $Update\_2D(k',\ k'+1:N)$, where $k \neq k'$, can also be overlapped. The idea of compute-ahead scheduling is also incorporated, i.e., $Factor(k+1)$ is executed as soon as $Update\_2D(k,\ k+1)$ finishes.

Some detailed explanation for pivoting, scaling and swapping is given below. In line (5) of Figure 13, the whole subrow is communicated when each processor reports its local maximum to $P_{k \bmod p_r,\ k \bmod p_c}$, i.e., the processor that owns the L_{kk} block. Let m be the current global column number on which the pivoting is conducted, then without further synchronization, processor $P_{k \bmod p_r,\ k \bmod p_c}$ can locally swap subrow m with subrow t which contains the selected pivoting element. This shortens the waiting time to conduct further updating with a little more communication volume. However in line (08), processor $P_{k \bmod p_r,\ k \bmod p_c}$ must send the original subrow m to the owner of subrow t for swapping, and the selected subrow t to other processors as well for updating. In $Factor()$ tasks, synchronizations take place at lines (05), (07) and (08) when each processor reports its local maximum to $P_{k \bmod p_r,\ k \bmod p_c}$, and $P_{k \bmod p_r,\ k \bmod p_c}$ broadcasts the subrow containing global maximum along the processor column. For task ScaleSwap(), the main role is to scale $U_{k,\ k+1:N}$ and perform delayed row interchanges for remaining submatrices $A_{k+1:N,\ k+1:N}$.

We examine the degree of parallelism exploited in this algorithm by determining number of updating stages that can be overlapped. Using this information we can also determine the extra buffer space needed per processor to execute this algorithm correctly. We define the **stage overlapping degree**

```
ScaleSwap(k)
    if my_cno ≠ k mod p_c then receive the pivot sequence from P_{my_rno, k mod p_c};
    for m = S(k) to S(k+1) − 1
        if This processor own a part of row m or the pivot row t for column m
            Interchange nonzero parts of row t and row m owned by this processor;
    endfor
    if my_rno = k mod p_r
        if my_cno ≠ k mod p_c then receive L_kk from P_{my_rno, k mod p_c};
        Scale nonzero blocks in U_{k, k:N} owned by this processor;
        Multicast the scaling results along this processor column;
    if my_cno ≠ k mod p_c then receive L_{k:N, k} from P_{my_rno, k mod p_c};
    if my_rno ≠ k mod p_r then receive U_{k, k:N} from P_{k mod p_r, my_cno};
```

Figure 14: Task $ScaleSwap(k)$ for the 2-D asynchronous code.

```
Update_2D(k, j)
    if j mod p_c = my_cno
        for i = k+1 to N
            if i mod p_r = my_rno
                Update A_ij using L_ik and U_kj;
        endfor
```

Figure 15: $Update\_2D(k, j)$ for the 2-D asynchronous code.

for updating tasks as

$$max\{|k - k'| \mid \text{There exist tasks } Update\_2D(k,*) \text{ and } Update\_2D(k',*) \text{ executed concurrently.}\}$$

Here $Update\_2D(k,*)$ denotes a set of $Update\_2D(k,j)$ tasks where $k < j \leq N$.

Theorem 2 *For the asynchronous 2-D algorithm on p processors where $p > 1$ and $p = p_r \times p_c$, the reachable upper bound of overlapping degree is p_c among all processors; and the reachable upper bound of overlapping degree within a processor column is $min(p_r - 1, p_c)$.*

Proof: We will use the following facts in proving the theorem:

- **Fact 1.** $Factor(k)$ is executed at processors with column number $k \bmod p_c$. Processors on this column are synchronized. When a processor completes $Factor(k)$, this processor can still do $Update\_2D(k-1,*)$ as shown in Figure 13, but all $Update\_2D(k-t,*)$ tasks belonging to this processor where $t > 1$ must have been completed on this processor.

- **Fact 2.** $ScaleSwap(k)$ is executed at processors with row number $k \bmod p_r$. When a processor completes $ScaleSwap(k)$, all $Update\_2D(k-t,*)$ tasks belonging to this processor where $t > 0$ must have been completed on this processor.

Part 1. First we show that $Update\_2D()$ tasks can be overlapped to a degree of p_c among all processors.

When $p_c = 1$, it is trivial based on Fact 1. When $p_c > 1$, we can imagine a scenario in which all processors in column 0 have just finished task $Factor(k)$, and some of them are still working on $Update\_2D(k-1,*)$ tasks, and processors in column 1 could go ahead and execute $Update\_2D(k,*)$ tasks. After processors in column 1 finish $Update\_2D(k,k+1)$ task, they will execute $Factor(k+1)$. Then after finishing $Update\_2D(k,*)$ tasks, processors in column 2 could execute $Update\_2D(k+1,k+2)$ task and start $Factor(k+2)$ task and so on. Finally, processors in column $p_c - 1$ could execute $Factor(k+p_c-1)$ and then start $Update\_2D(k+p_c-1,*)$. At this moment, processors in column 0 may be still working on $Update\_2D(k-1,*)$. Thus the overlapping degree is p_c.

Now we will show by contradiction that the maximum overlapping degree is p_c. Assume that at some moment, there exist two updating stages being executed concurrently: $Update\_2D(k,*)$ and $Update\_2D(k',*)$, where $k' > k + p_c$. Then $Factor(k')$ must have been completed. Without loss of generality, assuming that processors in column 0 execute $Factor(k')$, then according to Fact 1 all $Update\_2D(j,*)$ tasks where $j < k'-1$ should be completed before this moment. Since block cyclic mapping is used, it is easy to see each processor column has performed one of the $Factor(j)$ tasks where $k' - p_c + 2 \leq j \leq k'$. Then $Update\_2D(k' - p_c + 1)$ should be completed on all processors. Then for any concurrent stage $Update\_2D(k,*)$, k must satisfy $k \geq k' - p_c$, which is a contradiction.

Part 2. First, we show that overlapping degree $min(p_r - 1, p_c)$ can be achieved within a processor column. For the convenience of illustration, we consider a scenario in which all delayed row interchanges in $ScaleSwap()$ take place locally without any communication within a processor column. Therefore there is no interprocessor synchronization going on within a processor column except in $Factor()$ tasks. Assuming $p_r - 1 < p_c$, we can imagine at some moment, processors in column 0 have completed $Factor(s)$, and $P_{0,0}$ has just finished $ScaleSwap(s)$, and starts executing $Update\_2D(s,*)$, where $s \bmod p_r = 0$ and $s \bmod p_c = 0$. Then processors in column 1 will execute $Update\_2D(s,s+1)$ and then $Factor(s+1)$, after which $P_{1,0}$ can start $ScaleSwap(s+1)$ and then $Update\_2D(s+1,*)$. Following this reasoning, after $Update\_2D(s+p_r-2, s+p_r-1)$ and $Factor(s+p_r-1)$ have been finished on processors of column $p_r - 1$, $P_{p_r-1,0}$ could complete previous $Update\_2D()$ tasks and $ScaleSwap(s+p_r-1)$, and start $Update\_2D(s+p_r-1,*)$. Now $P_{0,0}$ may be still working on $Update\_2D(s,*)$. Thus the overlapping degree is $p_r - 1$. If $p_r \geq p_c + 1$, obviously the above reasoning will stop when processors of column $p_c - 1$ finish $Update\_2D(s+p_c-2, s+p_c-1)$ and $Factor(s+p_c-1)$. In that case when $P_{p_c-1,0}$ is to start $Update\_2D(s+p_c-1,*)$, the processor $P_{p_r-1,0}$ could be still working on $Update\_2D(s-1,*)$ because of the compute ahead scheduling. Hence the overlapping degree is p_c.

Now we need to show that the upper bound of overlapping degree within a processor column is $min(p_r - 1, p_c)$. We have already shown in the proof of Part 1 that the overall overlapping degree is less than p_c, so is the overlapping degree within a processor column. To prove it is also less than $p_r - 1$, we can use the similar proof as that for part 1, except using $ScaleSwap(k)$ to replace $Factor(k)$, and using Fact 2 instead of Fact 1.

■

Knowing degree of overlapping is important in determining the amount of memory space needed to accommodate those communication buffers on each processor for supporting asynchronous execution. Buffer space is additional to data space needed to distribute the original matrix. There are four types of communication that needs buffering:

1. Pivoting along a processor column (lines (05), (07), and (08) in Figure 13), which includes communicating pivot positions and multicasting pivot rows. We call the buffer for this purpose *Pbuffer*.
2. Multicasting along a processor row (line (12), (13) and (14) in Figure 13). The communicated data includes L_{kk}, local nonzero blocks in $L_{k+1:N,\ k}$, and pivoting sequences. We call the buffer for this purpose *Cbuffer*.
3. Row interchange within a processor column (line (05) in Figure 14). We call this buffer *Ibuffer*.
4. Multicasting along a processor column (line (10) in Figure 14). The data includes local nonzero blocks of a row panel. We call the buffer *Rbuffer*.

Here we assume that $p_r \leq p_c + 1$, because based on our experimental results, setting $p_r \leq p_c + 1$ always leads to better performance. Thus the overlapping degree of *Update_2D*() tasks within a processor row is at most p_c, and the overlapping degree within a processor column is at most $p_r - 1$. Then we need p_c separate *Cbuffer*'s for overlapping among different columns and $p_r - 1$ separate *Rbuffer*'s for overlapping among different rows.

We estimate the size of each *Cbuffer* and *Rbuffer* as follows. Assuming that the sparsity ratio of a given matrix is s after fill-in and the maximum block size is $BSIZE$, each *Cbuffer* is of size:

$$C = max\{\text{space for local nonzero blocks of } L_{k:N,k}, 1 \leq k \leq N\} \approx n \cdot BSIZE \cdot s/p_r.$$

Similarly each *Rbuffer* is of size:

$$R = max\{\text{space for local nonzero blocks of } U_{k,k+1:N}, 1 \leq k \leq N\} \approx n \cdot BSIZE \cdot s/p_c.$$

We ignore the buffer size for *Pbuffer* and *Ibuffer* because they are very small (the size of *Pbuffer* is only about $BSIZE \cdot BSIZE$ and the size of *Ibuffer* is about $s \cdot n/p_c$). Thus the total buffer space needed for the asynchronous execution is: $C \cdot p_c + R \cdot (p_r - 1) \approx n \cdot BSIZE \cdot s \cdot (p_c/p_r + p_r/p_c)$.

Notice that the sequential space complexity $S_1 = n^2 s$. In practice, we set $p_c/p_r = 2$. Therefore the buffer space complexity for each processor is $2.5 \cdot n \cdot BSIZE \cdot s$, or $\frac{2.5 \cdot BSIZE}{n} \cdot S_1$, which is very small for a large matrix. For all the benchmark matrices we have tested, the buffer space is less than 100 K words. Given a sparse matrix, if the matrix data is evenly distributed onto p processors, the total memory requirement per processor is $S_1/p + O(1)$ considering $n \gg p$ and $n \gg BSIZE$. This leads us to conclude that the 2-D asynchronous algorithm is very space scalable.

6 Experimental studies

Our experiments were originally conducted on a Cray-T3D distributed memory machine at San Diego Supercomputing Center. Each node of the T3D includes a DEC Alpha EV4(21064) processor with 64 Mbytes of memory. The size of the internal cache is 8 Kbytes per processor. The BLAS-3 matrix-matrix multiplication routine `DGEMM` can achieve 103 MFLOPS, and the BLAS-2 matrix-vector multiplication routine `DGEMV` can reach 85 MFLOPS. These numbers are obtained assuming all the data is in cache and using cache read-ahead optimization on T3D, and the matrix block size is chosen as 25. The communication network of the T3D is a 3-D torus. Cray provides a shared memory access library called `shmem` which can achieve 126 Mbytes/s bandwidth and $2.7\mu s$ communication overhead using `shmem_put()` primitive [34]. We have used `shmem_put()` for the communications in all the implementations.

We have also conducted experiments on a newly acquired Cray-T3E at San Diego Supercomputing Center. Each T3E node has a clock rate of 300 MHZ, an 8Kbytes internal cache, 96Kbytes second level cache, and 128 Mbytes main memory. The peak bandwidth between nodes is reported as 500 Mbytes/s and the peak round trip communication latency is about 0.5 to 2 μs [33]. We have observed that when block size is 25, `DGEMM` achieves 388 MFLOPS while `DGEMV` reaches 255 MFLOPS. We have used block size 25 in our experiments since if the block size is too large, the available parallelism will be reduced. In this section we mainly report results on T3E. In some occasions that the absolute performance is concerned, we also list the results on T3D to see how our approach scales when the underline architecture is upgraded. All the results are obtained on T3E unless explicitly stated.

In calculating the MFLOPS achieved by our parallel algorithms, we do not include extra floating point operations introduced by the overestimation. We use the following formula:

$$\text{Achieved MFLOPS} = \frac{\text{Operation count obtained from SuperLU}}{\text{Parallel time of our algorithm on T3D or T3E}}.$$

The operation count for a matrix is reported by running SuperLU code on a SUN workstation with large memory since SuperLU code cannot run for some large matrices on a single T3D or T3E node due to memory constraint. We also compare the S^* sequential code with SuperLU to make sure that the code using static symbolic factorization is not too slow and will not prevent the parallel version from delivering high megaflops.

6.1 Impact of static symbolic factorization on sequential performance

We study if the introduction of extra nonzero elements by the static factorization substantially affects the time complexity of numerical factorization. We compare the performance of the S^* sequential code with SuperLU code performance in Table 2 [1] for those matrices from Table 1 that

[1]The times for S^* in this table do not include symbolic preprocessing cost while the times for SuperLU include symbolic factorization because SuperLU does it on the fly. Our implementation for static symbolic preprocessing is

can be executed on a single T3D or T3E node. We also introduce two other matrices to show how well the method works for larger matrices and denser matrices. One of the two matrices is `b33_5600` which is truncated from `BCSSTK33` because of the current sequential implementation is not able to handle the entire matrix due to memory constraint, and the other one is `dense1000`.

Matrix	S^* Approach				SuperLU				Exec. Time Ratio S^*/SuperLU	
	Seconds		Mflops		Seconds		Mflops			
	T3D	T3E	T3D	T3E	T3D	T3E	T3D	T3E	T3D	T3E
sherman5	2.87	0.94	8.81	26.9	2.39	0.78	10.57	32.4	1.21	1.22
lnsp3937	6.12	2.0	6.90	21.1	4.94	1.73	8.56	24.4	1.27	1.16
lns3937	6.67	2.19	6.71	20.4	5.30	1.84	8.45	24.3	1.26	1.19
sherman3	6.06	2.03	10.18	30.4	4.27	1.68	14.46	36.7	1.56	1.21
jpwh991	2.11	0.69	8.24	25.2	1.62	0.56	10.66	31.0	1.34	1.23
orsreg1	6.03	2.04	10.37	30.7	4.09	1.53	15.25	40.9	1.66	1.31
saylr4	10.38	3.53	10.31	30.3	7.18	2.69	14.95	39.8	1.60	1.31
goodwin	43.72	17.0	15.3	39.4	-	-	-	-	-	-
b33_5600	110.5	45.0	22.9	56.2	-	50.0	-	50.6	-	0.9
dense1000	10.48	4.04	63.6	165.0	19.6	8.39	34.0	79.4	0.53	0.48

Table 2: Sequential performance: S^* versus SuperLU. A "-" implies the data is not available due to insufficient memory.

Though the static symbolic factorization introduces a lot of extra computation as shown in Table 1, the performance of S^* after 2-D L/U partitioning is consistently competitive to that of highly optimized SuperLU. The absolute single node performance that has been achieved by the S^* approach on both T3D and T3E is consistently in the range of $5-10\%$ of the highest `DGEMM` performance for those matrices of small or medium sizes. Considering the fact that sparse codes usually suffer poor cache reuse, this performance is reasonable. In addition, the amount of computation for the testing matrices in Table 2 is small, ranging from 18 to 107 million double precision floating operations. Since the characteristic of the S^* approach is to explore more dense structures and utilize BLAS-3 kernels, better performance is expected on larger or denser matrices. This is verified on a matrix `b33_5600`. For even larger matrices such as `vavasis3`, we cannot run S^* on one node, but as shown later, the 2-D code can achieve 32.8 MFLOPS per node on 16 T3D processors. Notice that the megaflops performance per node for sparse Choleksy reported in [24] on 16 T3D nodes is around 40 MFLOPS, which is also a good indication that S^* single-node performance is satisfactory.

We present a quantitative analysis to explain why S^* can be competitive to SuperLU. Assume the speed of BLAS-2 kernel is ω_2 $second/flop$ and the speed of BLAS-3 kernel is ω_3 $second/flop$. The total amount of numerical updates is $\mathcal{C}$ $flops$ for SuperLU and $\mathcal{C}'$ $flops$ for the S^*. Apparently $\mathcal{C}' \geq \mathcal{C}$. For simplicity, we ignore the computation from the scaling part within each column because it contributes very little to the total execution time. Hence we have:

very efficient. For example, the preprocessing time is only about 2.76 seconds on a single node of T3E for the largest matrix we tested (`vavasis3`).

$$T_{SuperLU} = T_{symbolic} + \omega_2 \cdot \mathcal{C}, \quad and \tag{1}$$

$$T_{S^*} = (1-\rho) \cdot \omega_2 \cdot \mathcal{C}' + \rho \cdot \omega_3 \cdot \mathcal{C}', \tag{2}$$

where $T_{symbolic}$ is the time spent on dynamic symbolic factorization in the SuperLU approach, ρ is the percentage of the numerical updates that are performed by `DGEMM` in S^*. Let η be the ratio of symbolic factorization time to numerical factorization time in SuperLU, then we simplify Equation (1) to the following:

$$T_{SuperLU} = (1+\eta) \cdot \omega_2 \cdot \mathcal{C}. \tag{3}$$

We estimate that $\eta \approx 0.82$ for the tested matrices based on the results in [7]. In [17], we have also measured ρ as approximately $\rho \approx 0.67$. The ratios of the number of floating point operations performed in S^* and SuperLU for the tested matrices are available in Table 1. In average, the value of $\frac{\mathcal{C}'}{\mathcal{C}}$ is 3.98. We plug in these typical parameters in Equation 2 and 3, and we have:

$$\frac{T_{S^*}}{T_{SuperLU}} \approx \frac{(0.33 \cdot \omega_2 + 0.67 \cdot \omega_3) \cdot 3.98}{1.82 \cdot \omega_2}. \tag{4}$$

For T3D, $\omega_2 = \frac{1}{85}\ \mu s/flop$ and $\omega_3 = \frac{1}{103}\ \mu s/flop$. Then we can get $\frac{T_{S^*}}{T_{SuperLU}} \approx 1.93$. For T3E, $\omega_2 = \frac{1}{255}\ \mu s/flop$ and $\omega_3 = \frac{1}{388}\ \mu s/flop$. And we get $\frac{T_{S^*}}{T_{SuperLU}} \approx 1.68$. These estimations are close to the ratios obtained in Table 2. The discrepancy is caused by the fact that the submatrix sizes of supernodes are non-uniform, which leads to different caching performance. If submatrices are of uniform sizes, we expect our prediction is more accurate. For instance, in the dense case, $\frac{\mathcal{C}'}{\mathcal{C}}$ is exactly 1. The ratio $\frac{T_{S^*}}{T_{SuperLU}}$ is calculated as 0.48 for T3D and 0.42 for T3E, which are almost the same as the ratios listed in Table 2.

The above analysis shows that using BLAS-3 as much as possible makes S^* competitive to SuperLU. Suppose in a machine that `DGEMM` outperforms `DGEMV` substantially and the ratio of the computation that is performed by `DGEMM` is high enough, S^* could be faster than SuperLU for some matrices. The last two entries in Table 2 have already shown this.

6.2 Parallel performance of 1-D codes

In this subsection, we report a set of experiments conducted to examine the overall parallel performance of 1-D codes, the effectiveness of scheduling and supernode amalgamation.

Overall Performance: We list the MFLOPS numbers of the 1-D RAPID code obtained on various number of processors for several testing matrices in Table 3 (A "-" entry implies the data is not available due memory constraint, same below). We know that the megaflops of `DGEMM` on T3E is about 3.7 times as large as that on T3D, and the RAPID code after using a upgraded machine is speeded up about 3 times in average, which is satisfactory. For the same machine, the performance of the RAPID code increases when the number of processors increases and speedups compared to the pure S^* sequential code (if applicable) can reach up to 17.7 on 64 T3D nodes and 24.1 on 64 T3E nodes. From 32 to 64 nodes, the performance gain is small except for matrices `goodwin`, `e40r0100` and `b33_5600`, which are much larger problems than the rest. The reason is that those small tested matrices do not have enough amount of computation and parallelism to saturate a large number of processors when the elimination process proceeds toward the end. It is our belief that better and more scalable performance can be obtained on larger matrices. But currently the available memory on each node of T3D or T3E limits the problem size that can be solved with the current version of the RAPID code.

Matrix	P=2		P=4		P=8		P=16		P=32		P=64	
	T3D	T3E	T3D	T3E	T3D	T3E	T3D	T3E	T3D	T3E	T3D	T3E
sherman5	14.7	44.4	25.8	79.0	40.8	133.1	53.8	168.6	64.9	210.7	68.4	229.9
lnsp3937	11.1	34.3	19.6	62.1	29.7	93.8	45.4	145.5	57.1	183.5	63.0	201.0
lns3937	10.2	32.6	17.6	55.2	28.5	93.1	43.5	135.4	50.3	148.9	52.1	165.5
sherman3	16.4	51.4	30.0	90.7	45.7	143.5	61.1	192.8	64.3	199.0	66.3	212.7
jpwh991	13.3	41.4	23.2	75.6	40.5	124.2	51.2	173.9	58.0	193.2	60.0	217.3
orsreg1	17.4	53.4	30.6	90.6	51.2	160.3	68.7	215.6	75.3	223.3	75.3	231.6
goodwin	29.6	73.6	54.0	135.7	87.9	238.0	136.4	373.7	182.0	522.6	218.1	655.8
e40r0100	-	-	-	133.1	-	242.2	-	426.1	-	649.8	-	699.4
b33_5600	44.5	105.0	80.4	199.0	135.7	365.0	217.6	642.0	341.8	1029.0	405.3	1353.2

Table 3: Absolute performance (MFLOPS) of the 1-D RAPID code.

Effectiveness of Graph Scheduling: We compare the performance of 1-D CA code with 1-D RAPID code in Figure 16. The Y axis is $1 - PT_{RAPID}/PT_{CA}$, where PT stands for parallel time. For 2 and 4 processors, in certain cases, the compute-ahead code is slightly faster than the RAPID code. But for the number of processors more than 4, the RAPID code runs $10 - 72\%$ faster. The more processors involved, the bigger the performance gap tends to be. The reason is that for a small number of processors, there are sufficient tasks making all processors busy and the compute-ahead schedule performs well while the RAPID code suffers a certain degree of system overhead. For a larger number of processors, schedule optimization becomes important since there is limited parallelism to exploit.

Effectiveness of supernode amalgamation: We have examined how effective our supernode amalgamation strategy is using the 1-D RAPID code. Let PT_a and PT be the parallel time with and without supernode amalgamation respectively. The parallel time improvement ratios $(1 - PT_a/PT)$ on T3E for several testing matrices are listed in Table 4 and similar results on T3D are in [17]. Apparently the supernode amalgamation has brought significant improvement due to the increase of supernode size which implies an increase of the task granularities. This is important to obtaining good parallel performance [22].

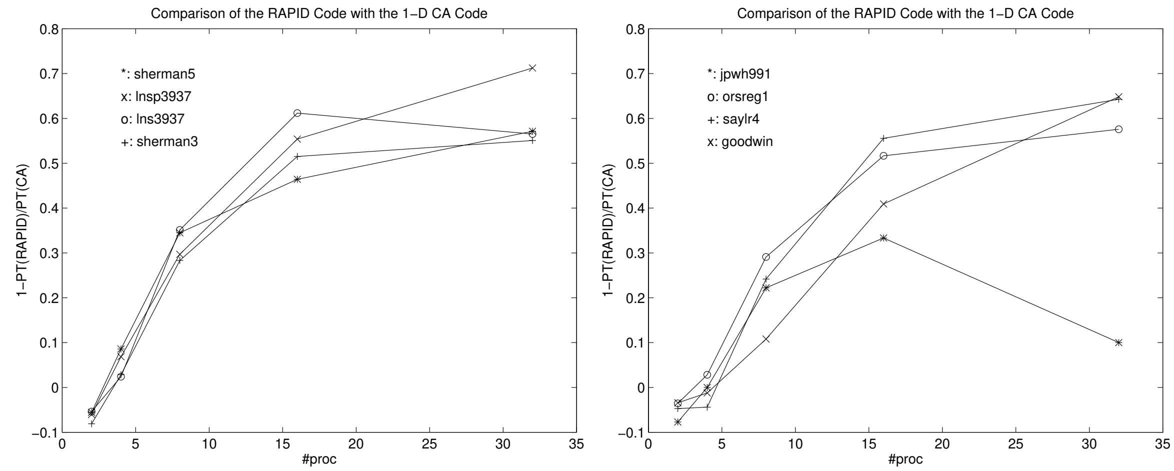

Figure 16: Impact of different scheduling strategies on 1-D code approach.

Matrix	P=1	P=2	P=4	P=8	P=16	P=32
sherman5	47%	47%	46%	50%	40%	43%
lnsp3937	50%	51%	53%	53%	51%	39%
lns3937	55%	54%	54%	54%	51%	35%
sherman3	20%	25%	23%	28%	22%	14%
jpwh991	48%	48%	48%	50%	47%	40%
orsreg1	16%	18%	18%	26%	15%	10%
saylr4	21%	22%	23%	23%	30%	18%

Table 4: Parallel time improvement obtained by supernode amalgamation.

6.3 2-D code performance

As mentioned before, our 2-D code exploits more parallelism but maintains a lower space complexity, and has much more potential to solve large problems. We show the absolute performance obtained for some large matrices on T3D in Table 5. Since some matrices cannot fit for a small number of processors, we only list results on 16 or more processors. The maximum absolute performance achieved on 64 nodes of T3D is 1.48 GFLOPS, which is translated to 23.1 MFLOPS per node. For 16 nodes, the per-node performance is 32.8 MFLOPS.

Table 6 shows the performance numbers on T3E for the 2-D code. We have achieved up to 6.878 GFLOPS on 128 nodes. For 64 nodes, megaflops on T3E are from 3.1 to 3.4 times as large as that on T3D. Again considering that `DGEMM` megaflops on T3E is about 3.7 times as large as that on T3D, our code performance after using a upgraded machine is good.

Notice that 1-D codes cannot solve the last six matrices of Table 6. For those matrices solvable using both 1-D RAPID and 2-D codes, we compare the average parallel time differences by computing the ratio: $1 - PT_{RAPID}/PT_{2D}$ and the result is in Figure 17. The 1-D RAPID code achieves

Matrix	P=16		P=32		P=64	
	Time(Sec)	Mflops	Time(Sec)	Mflops	Time(Sec)	Mflops
goodwin	6.0	110.7	4.6	145.2	3.6	184.8
e40r0100	21.2	116.6	15.2	162.5	10.5	235.4
ex11	87.9	305.0	53.4	501.8	33.4	802.6
raefsky4	129.8	242.9	76.0	413.8	43.2	719.2
vavasis3	169.8	525.4	100.8	885.1	60.3	1480.2

Table 5: Performance results of the 2-D code for large matrices on T3D.

Matrix	P=8		P=16		P=32		P=64		P=128	
	Time	Mflops	Time	Mflops	Time	Mflops	Time	Mflops	Time	Mflops
goodwin	3.1	215.2	1.9	344.6	1.3	496.3	1.1	599.2	0.9	715.2
e40r0100	12.1	205.1	7.2	342.9	4.8	515.8	3.3	748.0	2.7	930.8
ex11	50.7	528.8	28.3	946.2	16.2	1654.2	9.9	2703.1	6.4	4182.2
raefsky4	79.4	391.2	43.2	718.9	24.1	1290.7	13.9	2233.3	8.6	3592.9
inaccura	16.8	244.6	9.9	415.2	6.3	655.8	3.9	1048.0	3.0	1391.4
af23560	22.3	285.4	12.9	492.9	8.12	784.3	5.7	1123.2	4.2	1512.7
vavasis3	109.7	813.4	58.7	1519.0	33.6	2651.9	19.8	4505.5	13.0	6878.1

Table 6: Performance results of 2-D asynchronous algorithm on T3E. All times are in seconds.

better performance because it uses sophisticated graph scheduling technique to guide the mapping of column blocks and ordering of tasks, which results in better overlapping of communication with computation. The performance difference is larger for the matrices listed in the left of Figure 17 compared to the right of Figure 17. We partially explain the reason by analyzing load balance factors of the 1-D RAPID code and the 2-D code in Figure 18. The load balance factor is defined as $work_{total}/(P \cdot work_{max})$ [31]. Here we only count the work from the updating part because it is the major part of the computation. The 2-D code has better load balance, which can make up for the impact of lacking of efficient task scheduling. This is verified by Figure 17 and Figure 18. One can see that when the load balance factor of the 2-D code is close to that of the RAPID code (e.g., `lnsp3937`), the performance of the RAPID code is much better than the 2-D code; when the load balance factor of the 2-D code is significantly better than that of the RAPID code (e.g., `jpwh991` and `orserg1`), the performance differences are smaller.

Synchronous versus asynchronous 2-D code. Using a global barrier in the 2-D code at each elimination step can simplify the implementation, but it cannot overlap computations among different updating stages. We have compared parallel time reductions between the asynchronous code and the synchronous code for some testing matrices in Table 7. It shows that asynchronous design improves performance significantly, especially on large number of processors on T3E. It demonstrates the importance of exploiting parallelism using asynchronous execution. The experiment

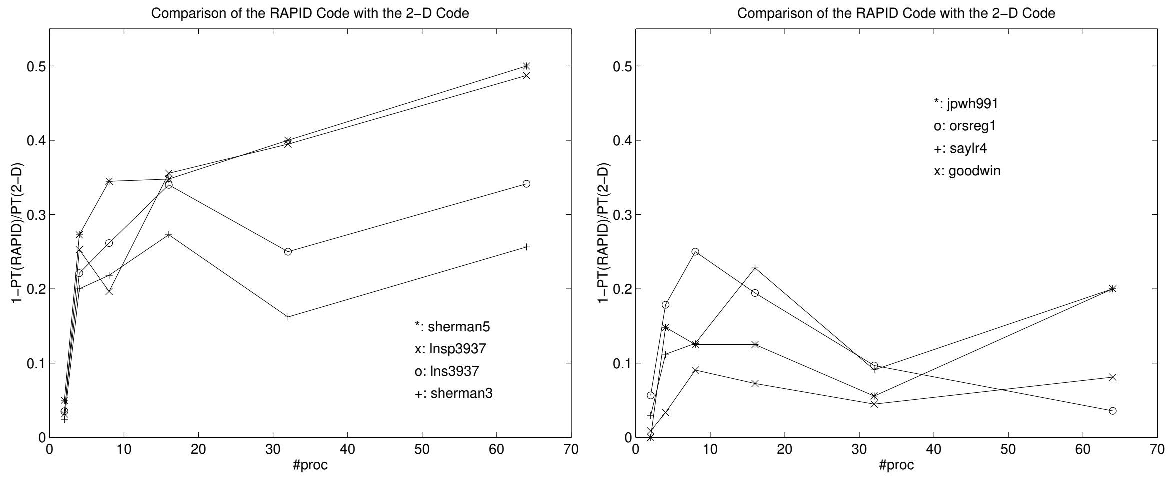

Figure 17: Performance improvement of 1-D RAPID over 2-D code: $1 - PT_{RAPID}/PT_{2D}$.

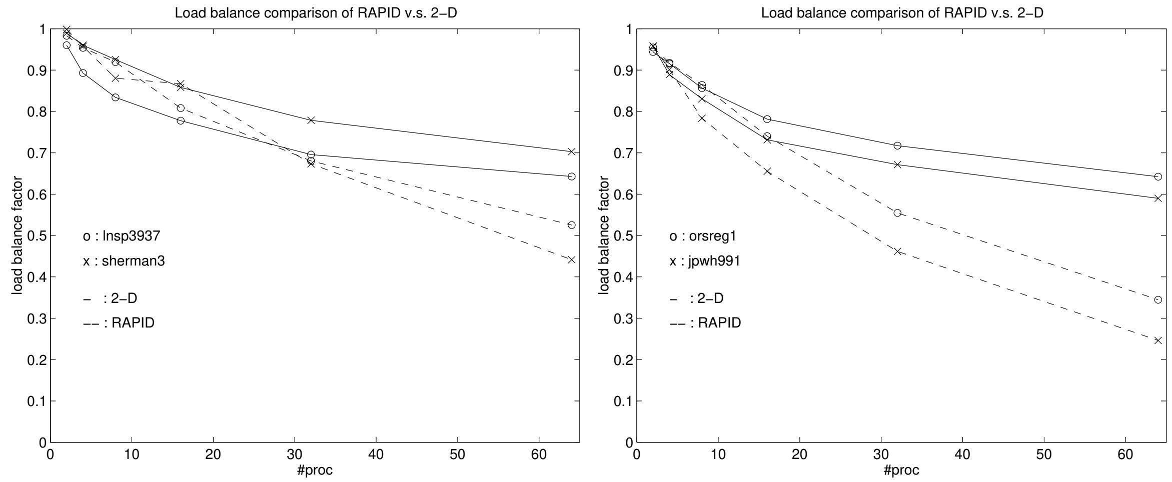

Figure 18: Comparison of load balance factors of 1-D RAPID code and 2-D code.

data on T3D is in [14].

7 Concluding remarks

In this paper we present an approach for parallelizing sparse LU factorization with partial pivoting on distributed memory machines. The major contribution of this paper is that we integrate several techniques together such as static symbolic factorization, scheduling for asynchronous parallelism, 2-D L/U supernode partitioning techniques to effectively identify dense structures, and maximize the use of BLAS-3 subroutines in the algorithm design. Using these ideas, we are able to exploit more data regularity for this open irregular problem and achieve up to 6.878 GFLOPS on 128 T3E nodes. This is the highest performance known for this challenging problem and the previous record was 2.583 GFLOPS on shared memory machines [8].

Matrix	P=2	P=4	P=8	P=16	P=32	P=64
sherman5	7.7%	6.4%	19.4%	28.1%	25.9%	24.1%
lnsp3937	7.3%	7.1%	22.2%	28.57%	26.9%	27.9%
lns3937	6.6%	2.8%	18.8%	26.5%	28.6%	26.8%
sherman3	10.2%	12.4%	20.3%	22.7%	26.0%	25.0%
jpwh991	8.7%	10.0%	23.8%	33.3%	35.7%	28.6%
orsreg1	6.1%	7.7%	17.5%	28.0%	20.5%	28.2%
saylr4	8.0%	10.7%	21.0%	29.6%	30.2%	27.4%
goodwin	5.4%	14.1%	14.2%	24.6%	26.0%	30.2%
e40r0100	2.4%	8.7%	8.1%	16.8%	18.1%	29.9%
ex11	-	9.0%	6.9%	14.9%	12.6%	24.5%
raefsky4	-	9.4%	8.1%	16.2%	13.5%	27.1%
vavasis3	-	-	7.8%	17.4%	15.2%	29.0%

Table 7: Performance improvement of 2-D asynchronous code over 2-D synchronous code.

The comparison results show that the 2-D code has a better scalability than 1-D codes because 2-D mapping exposes more parallelism with a carefully designed buffering scheme. But the 1-D RAPID code still outperforms the 2-D code if there is sufficient memory since the scheduling and execution techniques for the 2-D code are simple, and are not competitive to graph scheduling. Recently we have conducted research on developing space efficient scheduling algorithms while retaining good time efficiency [18]. It is still an open problem to develop advanced scheduling techniques that better exploit parallelism for 2-D sparse LU factorization with partial pivoting. There are other issues which are related to this work and need to be further studied, for example, alternative for parallel sparse LU based on Schur complements [13] and static estimation and parallelism exploitation for sparse QR [29, 35].

It should be noted that the static symbolic factorization could fail to be practical if the input matrix has a nearly dense row because it will lead to an almost complete fill-in of the whole matrix. It might be possible to use different matrix reordering to avoid that. Fortunately, this is not the case in most of matrices we have tested. Therefore our approach is applicable to a wide range of problems using a simple ordering strategy. It is interesting in the future to study ordering strategies that minimize overestimation ratios so that S^* can consistently deliver good performance for various classes of sparse matrices.

Acknowledgment

This work is supported by NSF RIA CCR-9409695, NSF CDA-9529418, the UC MICRO grant with a matching from SUN, NSF CAREER CCR-9702640, and ARPA DABT-63-93-C-0064 through the Rutgers HPCD project.

We would like to thank Kai Shen for the efficient implementation of the static symbolic factorization algorithm, Xiaoye Li and Jim Demmel for helpful discussions and providing us their testing matrices and SuperLU code, Cleve Ashcraft, Tim Davis, Apostolos Gerasoulis, Esmond Ng, Ed Rothberg, Rob Schreiber, Horst Simon, Chunguang Sun, Kathy Yelick and anonymous referees for their valuable comments.

References

[1] C. Ashcraft and R. Grimes. The Influence of Relaxed Supernode Partitions on the Multifrontal Method. *ACM Transactions on Mathematical Software*, 15(4):291–309, 1989.

[2] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon. Progress in Sparse Matrix Methods for Large Sparse Linear Systems on Vector Supercomputers. *International Journal of Supercomputer Applications*, 1:10–30, 1987.

[3] T. Davis. User's guide for the Unsymmetric-pattern Multifrontal Package (UMFPACK). Technical Report TR-93-020, Computer and Information Sciences Department, University of Florida, June 1993.

[4] T. Davis. Personal Communication, 1997.

[5] T. Davis and I. S. Duff. An Unsymmetric-pattern Multifrontal Method for Sparse LU factorization. *SIAM Matrix Analysis & Applications*, January 1997.

[6] J. Demmel. Numerical Linear Algebra on Parallel Processors. Lecture Notes for NSF-CBMS Regional Conference in the Mathematical Sciences, June 1995. To be published as a book by SIAM.

[7] J. Demmel, S. Eisenstat, J. Gilbert, X. Li, and J. Liu. A Supernodal Approach to Sparse Partial Pivoting. Technical Report CSD-95-883, UC Berkeley, September 1995.

[8] J. Demmel, J. Gilbert, and X. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. Technical Report CSD-97-943, UC Berkeley, February 1997.

[9] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of Basic Linear Algebra Subroutines. *ACM Trans. on Mathematical Software*, 14:18–32, 1988.

[10] I. Duff and J. Reid. The Multifrontal Solution of Indefinite Sparse Symmetric Systems of Equations. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[11] I. S. Duff. On Algorithms for Obtaining a Maximum Transversal. *ACM Transactions on Mathematical Software*, 7(3):315–330, September 1981.

[12] I. S. Duff. Personal Communication, 1996.

[13] S. Eisenstat and J. W. H. Liu. *Structural Representations of Schur Complements in Sparse Matrices*, volume 56 of *Graph Theory and Sparse Matrix Computation (Edited by Alan George and John R. Gilbert and Joseph W.H. Liu)*, pages 85–100. Springer-Verlag, New York, 1993.

[14] C. Fu, X. Jiao, and T. Yang. A Comparison of 1-D and 2-D Data Mapping for Sparse LU Factorization with Partial Pivoting. In *Proc. of Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[15] C. Fu and T. Yang. Efficient Run-time Support for Irregular Task Computations with Mixed Granularities. In *Proceedings of IEEE International Parallel Processing Symposium*, pages 823–830, Hawaii, April 1996.

[16] C. Fu and T. Yang. Run-time Compilation for Parallel Sparse Matrix Computations. In *Proceedings of ACM International Conference on Supercomputing*, pages 237–244, Philadelphia, May 1996.

[17] C. Fu and T. Yang. Sparse LU Factorization with Partial Pivoting on Distributed Memory Machines. In *Proceedings of ACM/IEEE Supercomputing*, Pittsburgh, November 1996.

[18] C. Fu and T. Yang. Space and Time Efficient Execution of Parallel Irregular Computations. In *Proc. of 6th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 57–68, June 1997.

[19] K. Gallivan, B. Marsolf, and H. Wijshoff. The Parallel Solution of Nonsymmetric Sparse Linear Systems Using H* Reordering and an Associated Factorization. In *Proc. of ACM International Conference on Supercomputing*, pages 419–430, Manchester, July 1994.

[20] A. George and E. Ng. Symbolic Factorization for Sparse Gaussian Elimination with Partial Pivoting. *SIAM J. Scientific and Statistical Computing*, 8(6):877–898, November 1987.

[21] A. George and E. Ng. Parallel Sparse Gaussian Elimination with Partial Pivoting. *Annals of Operations Research*, 22:219–240, 1990.

[22] A. Gerasoulis and T. Yang. On the Granularity and Clustering of Directed Acyclic Task Graphs . *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[23] G. Golub and J. M. Ortega. *Scientific Computing: An Introduction with Parallel Computing Compilers* . Academic Press, 1993.

[24] A. Gupta and V. Kumar. Highly Scalable Parallel Algorithms for Sparse Matrix Factorization. Technical Report TR 94-63, Computer Science, Univ. of Minnesota, August 1994.

[25] S. Hadfield and T. Davis. A Parallel Unsymmetric-pattern Multifrontal Method. Technical Report TR-94-028, Computer and Information Sciences Department, University of Florida, August 1994.

[26] M. Heath, E. Ng, and B. Peyton. Parallel Algorithms for Sparse Linear Systems . *SIAM Review*, 33(3):420–460, September 1991.

[27] X. Jiao. Parallel sparse gaussian elimination with partial pivoting and 2-d data mapping. Master's thesis, Dept. of Computer Science, University of California at Santa Barbara, August 1997.

[28] J. W. H. Liu. Computational Models and Task Scheduling for Parallel Sparse Cholesky Factorization. *Parallel Computing*, 18:327–342, 1986.

[29] P. Raghavan. Distributed Sparse Gaussian Elimination and Orthogonal Factorization. *SIAM Journal on Scientific Computing*, 16(6):1462–1477, November 1995.

[30] E. Rothberg. *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization.* PhD thesis, Dept. of Computer Science, Stanford, December 1992.

[31] E. Rothberg and R. Schreiber. Improved Load Distribution in Parallel Sparse Cholesky Factorization. In *Proc. of Supercomputing'94*, pages 783–792, November 1994.

[32] S. L. Scott. Synchronization and Communication in the T3E Multiprocess. In *ASPLOS-VII*, October 1996.

[33] S. L. Scott and G. M. Thorson. The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus. In *Proceedings of HOT Interconnects IV*, August 1996.

[34] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers. In *Proc. of International Conference on Supercomputing*, pages 1–10, Barcelona, July 1995.

[35] C. Sun. Parallel Sparse Orthogonal Factorization on Distributed-memory Multiprocessors. *SIAM Journal on Scientific Computing*, 17(3):666–685, May 1996.

[36] T. Yang and A. Gerasoulis. PYRROS: Static Task Scheduling and Code Generation for Message-Passing Multiprocessors . In *Proc. of 6th ACM International Conference on Supercomputing*, pages 428–437, 1992.