Finding exact and approximate block structures for ILU preconditioning *

Yousef Saad[†]

August 17th, 2001

Abstract

Sparse matrices which arise in many applications often possess a block structure which can be exploited in iterative and direct solution methods. These block-matrices have as their entries small dense blocks with constant or variable dimensions. Block versions of incomplete LU factorizations which have been developed to take advantage of such structures give rise to a class of preconditioners that are among the most effective available. This paper presents general techniques for determining automatically block structures in sparse matrices. A standard 'graph compression' algorithm used in direct sparse matrix methods is considered along with two other algorithms which are also capable of unraveling approximate block structures.

Key words: Graph Compression, Incomplete LU factorization, Block ILU preconditioners, hash-based graph compression, cosine-based graph compression, indistinguishable nodes.
AMS subject classifications: 65F10, 65N06.

1 Introduction

ILU-type preconditioners combined with Krylov subspace accelerators are currently among the most effective iterative techniques for solving large sparse, irregularly structured, linear systems of equations. Incomplete LU factorizations, which consist of performing a Gaussian elimination and dropping some fill-in, give rise to an important class of efficient and relatively robust preconditioners. Among these, the subclass of "block ILU" techniques is known by practitioners to be extremely powerful for some types of problems, though this is seldom emphasized in the standard Numerical Linear Algebra literature. A block ILU method, is one which treats (small) dense submatrices of the matrix A as single entities. The classical case is when the coefficient matrix A is a discretization of a partial differential equation in which a fixed number of variables, say l, is associated with each mesh point. In this case, the matrix can be viewed as a "block-matrix", i.e., a matrix whose entries are dense $l \times l$ submatrices. This situation is fairly common, for example, in Computational Fluid Dynamics, where it is typical for l to be equal to 4 in two dimensions and 5 in three dimensions. In 2-D, the four variables may represent density (ρ), and scaled energy (ρE), and two scaled velocity components of the fluid ($\rho u, \rho v$). In 3-D an additional velocity component is added.

Block ILU preconditioning applied to such matrices gives rise to a fast and fairly robust iterative solution procedure, see for example [6]. It is known that it is always a good strategy to use a block version

*This work was supported in part by the Army Research Office under grant DAAD19-00-1-0485, and in part by NSF under grant NSF-INT 0003274

[†]Department of Computer Science and Engineering, University of Minnesota, 4-192 EE/CS Building, 200 Union Street S.E., Minneapolis, MN 55455. E-mail: saad@cs.umn.edu.

of ILU instead of a scalar version when this is possible. An obvious advantage of such techniques is the savings in storage, since most column indices and pointers for the block entries are avoided. Indeed, the usual Block Sparse Row format used in SPARSKIT [11], employs Compressed Sparse Row format for the resulting $(n/l) \times (n/l)$ block matrix, but each value in the CSR format becomes an array of size $l \times l$. A more important advantage is the potential gains obtained from computing with dense blocks and using BLAS3-based computations.

It is easy to generalize the constant block-size format to a variable block-size format, see, for example the SPARSKIT package [11]. Such generalizations are important because many applications lead to matrices with variable blocks.

The problem of finding a block structure for a matrix has been considered from a number of different angles in the past. For example, O'Neil and Szyld [10] proposed a scheme named PABLO which consists of reordering the matrix in such a way as to obtain diagonal blocks that are dense. During sparse direct solution methods, the eliminating column tends to propagate its pattern, leading to sets of columns referred to as "indistinguishable nodes". A technique for identifying these sets was proposed by Ashcraft [1]. It is clear that the sets of indistinguishable nodes provide a simple way of blocking a matrix. Similar algorithms are present in other packages for reordering in sparse matrices (e.g., the nested dissection ordering in Metis [8]) or in sparse direct solution packages such as MUMS [3].

What motivated the topic of this paper is that many of the matrices that have a relatively large number of nonzero elements per row have a 'near block structure'. This means that if we allow to expand the nonzero pattern a little, by making a few zero entries part of the nonzero pattern, we easily obtain a variable block matrix. Finding an "approximate" block structure is not as easy as finding an exact blocking. We propose two algorithms and show that the process of grouping the rows/columns is in general much less expensive than that of performing the standard block ILU factorization.

2 Graph compression and matrix blocking

In this section, we only consider matrices with symmetric patterns. For illustration, consider the matrix represented below, where an x represents a nonzero element:

$$A = \left[\begin{array}{cc|cc|ccc|c} x & x & 0 & 0 & x & x & x & 0 \\ x & x & 0 & 0 & x & x & x & 0 \\ \hline 0 & 0 & x & x & 0 & 0 & 0 & x \\ 0 & 0 & x & x & 0 & 0 & 0 & x \\ \hline x & x & 0 & 0 & x & x & x & 0 \\ x & x & 0 & 0 & x & x & x & 0 \\ x & x & 0 & 0 & x & x & x & 0 \\ \hline 0 & 0 & x & x & 0 & 0 & 0 & x \end{array}\right] \tag{1}$$

The above matrix has a clear variable block structure with blocks of size 2, 2, 3, and 1, respectively. To avoid confusion, we took an example where each block consists of contiguous columns but it is clear that this is not required (conceptually one can think of reordering the original matrix into one where the blocks are contiguous). The set of nodes V can be partitioned into 4 subsets $Y_1 = \{1, 2\}$, $Y_2 = \{3, 4\}$, $Y_3 = \{5, 6, 7\}$ and $Y_4 = \{8\}$. Let us call $\mathcal{P}$ this partition.

Since the matrix has a symmetric pattern its adjacency graph $G = (V, E)$ of the above matrix can be more succinctly represented by its quotient graph [7] denoted by $G/\mathcal{P} = \{V_\mathcal{P}, E_\mathcal{P}\}$ and defined by

$$V_\mathcal{P} = \{Y_1, \ldots, Y_p\}; \qquad E_\mathcal{P} = \{(Y_i, Y_j) \mid \exists v \in Y_i, w \in Y_j \; s.t. \; (v, w) \; \in V\}$$

In the above example, the adjacency matrix for the quotient graph is the 4×4 matrix

$$\begin{bmatrix} x & 0 & x & 0 \\ 0 & x & 0 & x \\ x & 0 & x & 0 \\ 0 & x & 0 & x \end{bmatrix}$$

One can consider that the entry in position (i, j) is a dense block of dimension $|Y_i| \times |Y_j|$, where $|X|$ is the cardinality of the set X. Finding automatically the partition $\mathcal{P}$ is useful in many different ways in sparse matrix computations. In sparse direct solution methods two vertices are said indistinguishable if they have the same pattern. Finding a blocking of the matrix in variable block format is clearly equivalent to grouping its vertices into subsets of indistinguishable nodes.

2.1 Hash-based algorithms

Blockings of the form shown in the previous subsection are relatively inexpensive to unravel. A technique used in several existing packages associates a key, or "checksum" value, to each row. The simplest such key used in [1] is the following:

$$key(u) = \sum_{(u,w) \in E} w \tag{2}$$

If the checksum of two rows is different then clearly the rows have a different pattern. If they are the same, then they may or may not have the same pattern, so an explicit comparison of the row patterns becomes necessary. Sorting the keys in a first step facilitates the process. At any given step of the block construction, a row will be compared with all rows succeeding it (in the order of the keys). For as long as the keys are the same a comparison of the patterns is made with the pattern of row_0. The algorithm is described in some detail below. In the description $K(i)$ is the checksum key shown above for each row i, and $Group(i)$ is a representative array: $Group(i) = k$ means i belongs to group k, unless k is equal to -1 in which case i is the representative of the group.

ALGORITHM **2.1** *Hash based compression*

1. *Compute the keys $K(u)$ for each vertex u*
2. *Set $Group(i) = -1$ for $i = 1, \ldots, n$*
3. *Sort the array $K(u)$ in increasing $K(u).key$*
4. *For $i = 1 : n$ Do*
5. *row = K(i).row ; key = K(i).key*
6. *For $j = i + 1 : n$ Do*
7. *rowj= K(j).row ; keyj= K(j).key*
8. *If $keyj! = key$ break;*
9. *If $Group(i) == -1$*
10. *If $pattern(rowj) == pattern(row)$ $Group(rowj) = row$*
11. *EndIf*
12. *EndDo*
13. *EndDo*

In order to give an idea on the performance of the above algorithm, we gathered statistics on a few runs for a sample of 14 matrices with different degrees of blocking and various sizes. Some generic information about the matrices is shown in Table 1. All matrices are available from the matrix-market [1]

Matrix	Type	n	nnz
BCSSTK10	RSA	1086	11578
BCSSTK11	RSA	1473	17857
BCSSTK12	RSA	1473	17857
BCSSTK16	RSA	4884	147631
RAEFSKY1	RUA	3242	294276
RAEFSKY2	RUA	3242	294276
RAEFSKY3	RUA	21200	1488768
RAEFSKY4	RUA	19779	1328611
BARTHT1A	RUA	14075	481125
BARTHT2A	RUA	14075	1311725
FIDAP006	RUA	1651	49479
FIDAP023	RUA	1409	43481
FIDAP028	RUA	2603	77653
PENALTY	RUA	35186	625828

Table 1: Information on th 14 matrices used for tests

except for Penalty, which is obtained from the Diffpack package [9, 5]. Note that the BCSSTK matrices are all symmetric whereas the solvers being used do not take advantage of symmetry. The other matrices are all nonsymmetric. In the table, n is the dimension of the matrix, nnz represents the total number of nonzero elements. The types "RSA" and "RUA" correspond to the labels "Real Symmetric Assembled" and "Real Unsymmetric Assembled" used in the Harwell-Boeing collection. For RSA matrices, only the lower triangular part is stored and nnz represents its number of nonzero elements.

Table 2 shows some statistics for the hash-based algorithm. The second and third columns of the table are the vertex and edge compression rates, respectively, achieved by the algorithm. These are the ratios $|V|/|V_{\mathcal{P}}|$ and $|E|/|E_{\mathcal{P}}|$. The next column shows the time required to execute the algorithm. The last two columns show, for reference, the time it takes to compute the variable block level-of-fill ILU with fill-levels of 0 and 2 respectively. As can be seen, the compression times are quite low – in fact in many cases negligible – relative to the time for the VBILU(k) factorization.

Before looking at alternative methods, we discuss potential improvements to checksum-based technique. Clearly, the idea of checksums is similar to that of hash functions, and the term hash was explicitly used in [2]. A row is 'hashed' into a given value. Two rows hashing into the same value is like a 'collision' in hash table and this makes it necessary to check the patterns. Similarly to hashing, it is possible to improve the performance of the method by clever choices of the hash function. An ideal hash function would assign a different value for each different row pattern which occurs. If we knew that the hash function had this property then there would be no need to compare patterns. Assuming that the vertices are labeled from 1 to n, a one-to-one hash function is

$$hash(u) = \sum_{(u,w)\ \in E} w 2^{w-1} \tag{3}$$

which is simply the integer represented by the pattern of the row viewed as a binary number (0 for a zero element, 1 for a nonzero element). The above 'perfect' hash function is not practical because it leads to huge numbers that are not machine-representable. If nothing is exploited about the pattern, then this is the only function that is one-to-one from the 2^n possible patterns to the set $1, \ldots, 2^n$. If we wish to reduce the comparisons to only those rows that have the same pattern, then clearly one can exploit what is known

[1] http://math.nist.gov/MatrixMarket/

	Compression statistics			VBILUK Time	
Matrix	V. Cmpr	E. Cmpr	Time	levf=0	levf = 2
BCSSTK10	2.30	5.52	0.00	0.04	0.05
BCSSTK11	1.89	3.46	0.01	0.07	0.17
BCSSTK12	1.89	3.46	0.01	0.06	0.17
BCSSTK16	2.73	7.45	0.08	0.49	2.83
RAEFSKY1	3.31	12.74	0.08	0.43	4.49
RAEFSKY2	3.31	12.74	0.14	0.47	4.61
RAEFSKY3	8.00	64.00	0.43	1.95	4.25
RAEFSKY4	2.93	8.75	0.38	2.07	25.28
BARTHT1A	5.00	25.00	0.16	0.63	1.15
BARTHT2A	5.01	25.07	0.35	1.71	10.56
FIDAP006	1.35	2.08	0.01	0.18	0.88
FIDAP023	1.20	1.47	0.01	0.21	1.20
FIDAP028	1.45	2.55	0.02	0.21	1.09
PENALTY	2.00	4.00	0.28	1.01	2.10

Table 2: Performance of the hash-based algorithm on a test sample of 14 sparse matrices

from constructing hash functions to reduce the number of collisions to a minimum. However, since for most matrices the time required to find indistinguishable nodes is negligible relative to the time it takes to solve the system, this is an issue that has not received much attention. This can be easily understood from the results of Table 2. Therefore, potential improvements to the simple choice (2) may exist but will not be explored in this paper.

2.2 The cosine algorithm

Consider now a slightly modified pattern obtained form the sample matrix A in (1). It can for example happen that the matrix in (1) was 'filtered' of its small elements and it lost three entries in the process, resulting in a matrix that is no longer block:

$$A = \left[\begin{array}{cc|cc|ccc|c} x & x & 0 & 0 & x & x & x & 0 \\ x & x & 0 & 0 & x & x & 0 & 0 \\ \hline 0 & 0 & x & x & 0 & 0 & 0 & x \\ 0 & 0 & x & x & 0 & 0 & 0 & x \\ \hline x & x & 0 & 0 & x & 0 & x & 0 \\ x & x & 0 & 0 & x & x & x & 0 \\ x & 0 & 0 & 0 & x & x & x & 0 \\ \hline 0 & 0 & x & x & 0 & 0 & 0 & x \end{array}\right] \tag{4}$$

The checksum algorithm would be able to find only one nontrivial block namely (2,3), all others being of size 1. However, in this case, it is clear that we still would like to consider the matrix as block – with the same blocking as in (1). By doing so, we make a small sacrifice in memory usage since a few zero entries are considered nonzeros to pad rows into desirable patterns. We refer to these "zero nonzero" entries as fill-ins for lack of a better term. Ideally, we would need a method which can find an 'approximate block structure'. It is not easy to extend the hash-based algorithm to handle this case. This would require a hash function which preserves proximity of patterns – in the sense that close patterns will result in close hash values. In addition, when rows which have the same key (or possibly nearby keys) must be compared we will need to count the number of matching column locations.

As an alternative to hash-based algorithms, we consider a technique which compares angles of rows (or columns). It may at first appear expensive to compare all the angles of the rows of a matrix between each other. However, a good implementation is key to making the method effective.

Let us call C the adjacency matrix related to A. This is a matrix which has the same pattern as A and whose nonzero values are equal to one. The main idea is to compute the i-th row of CC^T, or to be more accurate the upper triangular part of this row. Entry (i,j) in this row is the inner product of row i with row j. We need only consider those entries (i,j) with $j > i$. The inner product will give the cosine between row i and row j and if the corresponding angle is small enough, j will be added to the 'group' i. This is indicated by setting $group(j) = i$. The group of node i itself is set to -1 to indicate that this entry is a representative of a group. This is repeated for $i = 1, \ldots, n$, but when a row is already assigned ($group(i) > 0$) the row is skipped.

Key to this algorithm is the computation of $c_i C^T$ which is done row-wise in the usual way one proceeds when multiplying two sparse matrices in a row-wise structure, apart from a few important variations. This process is illustrated in Figure 1. Note that we describe the algorithm for general matrices with possibly

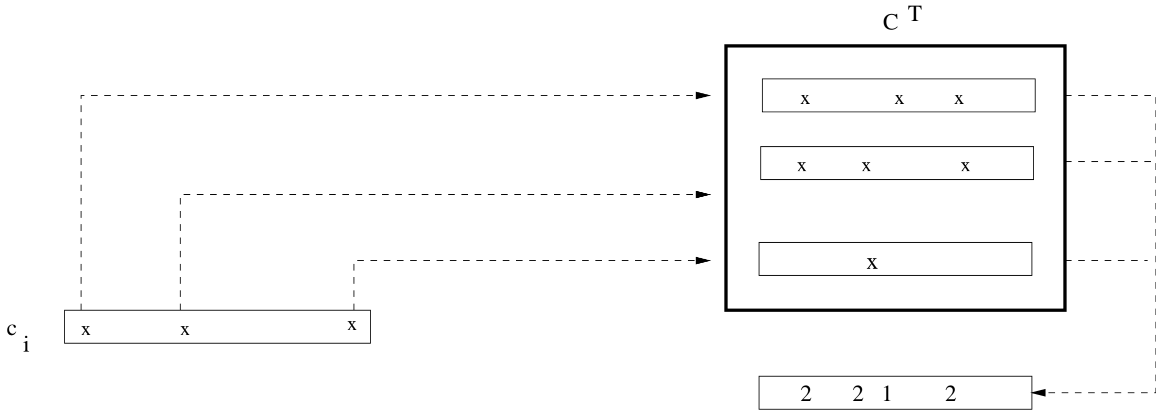

Figure 1: Computation of the inner product of c_i with all columns of A^T.

nonsymmetric patterns. The goal is to find groups of rows which have similar pattern. When seeking to find a block structure for a matrix, it is often assumed that the pattern is symmetric, although for block ILU, a symmetric pattern is not required. We will come back to this issue in a separate section.

The basic cost of computing all inner products of row i with all columns C^T (i.e., rows of C), is the sum of the number of nonzero entries of each row involved in this inner product:

$$\sum_{j=1}^{|c_i|} |c_j^T|$$

Here c_j^T denotes the j-th row of C^T and $|.|$ is the cardinality. This is basically the cost of computing the $i-th$ row of the upper triangular part of CC^T, except that there are no floating point operations made. However, there are several simplifications and improvements to this basic scheme which will substantially reduce the cost. The first is that row i is skipped if i has already been assigned to an existing group. As the algorithm progresses, many rows may already be assigned leading to nonnegligible potential savings. In addition, when computing the inner products, note that if column j with $j > i$ has already been assigned, it can be ignored in the calculation – since its inner product will not be of any use.

A final improvement comes from the fact that only entries $j > i$ of $c_i C^T$ need to be computed. This can be easily exploited provided the entries of each row of C^T are sorted by increasing columns. When C^T is actually obtained by transposing C with standard sparse transposition algorithms such as the one in SPARSKIT [11], for example, this property is true. Assume this is the case. Then when 'adding' row k to the current working row of $c_i C^T$, we will proceed backward from the rightmost column and move down until a column entry that is $\leq i$ is encountered, in which case, the loop is stopped. The final algorithm is sketched below. The notation $nz_A(i)$ means the number of nonzero entries in the i-th row of A.

ALGORITHM **2.2** *Cosine-based compression*

Input: pattern matrix C and tolerance τ; Output: set data structure for blocks.

1. *Compute the pattern matrix C^T.*
2. *Set Group$(i) = -1$ and Count$(j) = 0$ for $i = 1, \ldots, n$*
3. *For $i = 1 : n$ and if Group$(i) == -1$ Do :*
4.   *For $\{j \mid c_{ij} \neq 0\}$ Do*
5.   *Let row = j-th row of C^T*
6.   *For $k = nz_{C^T}(j)$ downto 1 Do :*
7.   *Let col = $row(k)$*
8.   *if col $\leq i$ break;*
9.   *if (Group(col) $== -1$)Count(col) $=$ Count(col) $+ 1$*
10.   *EndDo*
11.   *EndDo*
12.   *For each col such that Count(col) $\neq 0$ Do :*
13.   *If Count(col)$^2 > \tau * nz_A(i) * nz_A$(col)] Then*
14.   *Group(col) $== i$; Update the size of Group(i)*
15.   *EndIf*
16.   *Count(col) $= 0$*
17.   *EndDo*
18. *EndDo*

Table 3 shows statistics that are similar to those shown in Table 2 for the hash-based algorithm. A grouping tolerance of $\tau = 0.8$ was used for the tests. An additional column labeled "compression efficiency" is added, which represents the ratio of the original number of nonzero elements over the number of nonzero elements of the block matrix (taking fill-ins into account). An ideal compression which introduces no fill-in has an efficient of one. There are two observations worth noting. First, several matrices see a substantial increase in their edge compression rate, by introducing a moderate fill-in of less that 10%. For example, edge compression for BCSTK11 goes from 3.46 to 8.67 with an efficiency of 90.39%. A more substantial gain is seen for FIDAP023 whose edge compression rate goes from 1.47 to 6.11 with a similar efficiency. A second observation is that the cost of the algorithm increases substantially from that of the hash-based technique. In the worst case (BARTH2A) the time increases nearly fivefold. However, in most cases it still remains inferior to the cost of the least expensive block LU factorization, i.e., block ILU(0).

2.3 A hybrid algorithm

As is shown by the experiments seen so far, the angle algorithm is more expensive than hash-based techniques. It is easy to combine both algorithms into one whose cost will be closer to that of hash-based methods for

	Compression statistics				VBILUK Time	
Matrix	V. Cmpr	E. Cmpr	Cmpr eff	Time	levf=0	levf = 2
BCSSTK10	3.58	9.87	80.50%	0.02	0.04	0.06
BCSSTK11	3.02	8.67	90.39%	0.03	0.05	0.10
BCSSTK12	3.02	8.67	90.39%	0.03	0.06	0.11
BCSSTK16	4.31	15.56	79.24%	0.36	0.54	2.29
RAEFSKY1	3.64	14.11	93.61%	0.53	0.45	4.41
RAEFSKY2	3.64	14.11	93.61%	0.53	0.45	4.40
RAEFSKY3	8.64	69.16	95.23%	1.28	2.10	4.40
RAEFSKY4	4.31	13.85	79.49%	1.81	2.50	24.89
BARTHT1A	5.01	25.07	99.87%	0.41	0.63	1.14
BARTHT2A	6.20	33.52	88.35%	1.71	2.01	10.78
FIDAP006	3.02	6.70	93.37%	0.06	0.09	0.26
FIDAP023	3.02	6.11	90.11%	0.05	0.08	0.26
FIDAP028	3.18	7.10	87.81%	0.09	0.15	0.40
PENALTY	2.07	4.15	97.67%	0.71	0.99	2.06

Table 3: Performance of the cosine-based algorithm on a test sample of 14 sparse matrices

cases when a good blocking already exists. Since the quotient graph will be used, the technique introduced in this section is restricted to matrices with symmetric patterns. Simply put the idea is to do a first pass with the hash-based algorithm to detect any "exact" block structure and then to perform the angle algorithm on the quotient graph. The cost of this second pass which works on the typically smaller quotient graph is likely to be a small addition to the cost of the initial blocking by hashing. In the second pass, the algorithm scans each non-assigned row again to determine if it can be added to an existing group.

ALGORITHM **2.3** *Hybrid method for compression*

Input: pattern matrix C and tolerance τ; Output: set data structure for blocks.

1. *Call Algorithm 2.1 to find an initial group $Group_0$ assignment. Set $Group = Group_0$*
2. *Obtain the adjacency matrix C of the quotient graph $G_{\mathcal{P}}$, using the original labeling*
3. *For $i = 1 : n$ and if $Group(i) == -1$ Do :*
4.   *For $\{j \mid c_{ij} \neq 0\}$ Do*
5.   *Let $row = j$-th row of C and $s = |Group_0(j)|$*
6.   *For $k = nz_C(j)$ downto 1 Do :*
7.   *Let $col = row(k)$*
8.   *if $col \leq i$ break;*
9.   *if $(Group_0(col) == -1) Count(col) = Count(col) + s$*
10.   *EndDo*
11.   *EndDo*
12.   *For each col such that $Count(col) \neq 0$ Do :*
13.   *If $Count(col)^2 > \tau * nz_A(i) * nz_A(col)$ Then*
14.   *$Group(col) == i$; update the size of $Group(i)$*
15.   *EndIf*
16.   *$Count(col) = 0$*
17.   *EndDo*
18. *EndDo*

The second part of the algorithm is similar to that of Algorithm 2.2. One notable difference is that the increment in Line 9 is now s, the size of the j-th group obtained in the first path. The above algorithm works on the quotient graph but weighs the nonzero entries differently so that we will get the same inner products as in Algorithm 2.2.

	Compression statistics				VBILUK Time	
Matrix	V. Cmpr	E. Cmpr	Cmpr eff	Time	levf=0	levf = 2
BCSSTK10	3.58	9.87	80.50%	0.01	0.05	0.06
BCSSTK11	3.02	8.67	90.39%	0.02	0.06	0.11
BCSSTK12	3.02	8.67	90.39%	0.03	0.05	0.11
BCSSTK16	4.31	15.56	79.24%	0.17	0.55	2.30
RAEFSKY1	3.64	14.11	93.61%	0.15	0.45	4.36
RAEFSKY2	3.64	14.11	93.61%	0.16	0.45	4.36
RAEFSKY3	8.63	69.16	95.23%	0.60	2.11	4.39
RAEFSKY4	4.31	13.85	79.49%	0.77	2.52	24.90
BARTHT1A	5.01	25.07	99.87%	0.26	0.63	1.14
BARTHT2A	6.20	33.52	88.35%	0.58	2.00	10.69
FIDAP006	3.02	6.70	93.37%	0.05	0.09	0.26
FIDAP023	3.02	6.11	90.11%	0.06	0.08	0.26
FIDAP028	3.18	7.10	87.81%	0.07	0.14	0.41
PENALTY	2.07	4.15	97.67%	0.57	1.00	2.08

Table 4: Performance of the hybrid algorithm on a test sample of 14 sparse matrices

Table 4 shows statistics that are similar to those shown in Table 2 and in Table 3. The same grouping tolerance of $\tau = 0.8$ as in Table 3 was used. Observe that, as expected, the compression statistics are identical with those of the cosine-based algorithm. This is achieved at a cost that is, in most cases, much closer to the cost of the hash-based algorithm.

2.4 Fill-in Analysis

An important question regarding the angle-based compression algorithm, is related to its "memory efficiency". Blocking is useful in reducing overall execution times - by effectively exploiting BLAS3 computations and possibly by yielding better ILU techniques. On the other hand when inexact blocking is performed, we may have to introduce many nonzero elements to pad rows in order to obtain dense blocks. Consider the following two rows for example

$$\begin{matrix} 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{matrix}$$

where it is assumed that the first row (denoted by r) is the reference row in the cosine-based Algorithm and we denote by u the second row. Their squared 2-norms are their numbers of nonzero elements, or, $|r| = 6$ and $|u| = 4$, respectively. Their inner product $< r, u >$ which is equal to 4 in this case, is exactly the number of matching ones in the two patterns. The number of mismatching ones will represent 'fill-in' if the two rows are considered as a supernode. This is equal to

$$m(r, u) \equiv |r| + |u| - 2 < u, r >$$

which can be viewed as consisting of two parts: fill-in in the u row caused by nonmatching entries in r (totaling $|u| - < u, r >$) and fill-in in the r row caused by nonmatching entries in u (totaling $|r| - < u, r >$).

Because of the grouping threshold, we have $< u, r > \geq \tau\sqrt{|r|\ |u|}$ and therefore the maximum number of mismatches between the two rows, is such that

$$m(r,u) \leq |r| + |u| - 2\tau\sqrt{|u||r|}$$

In the simplest case when the two rows have the same length μ, this simplifies to

$$m(r,u) \leq 2\ \mu[1-\tau]$$

If a third row, say v, is now considered for addition to the group, the analysis becomes more complicated since it is difficult to predict whether the corresponding mismatch between v and r is all new or if part of it was already taken into account. The fill-in in v caused by nonmatching entries in r remains the same. On the other hand some of the fill-in in r caused by nonmatching entries in v, may already have been introduced by u and it is difficult to know whether or not this is the case. A worst case scenario is when all the v-linked fill in r is new, in which case we get the bound

$$m(r,u) + m(r,v)$$

for the fill-in introduced by the two rows. More generally, the fill-in occurring in the block represented by row r will be bounded by

$$\begin{aligned}
\text{Fill}(Group(r)) &\leq \sum_{u\ \in Group(r), u\neq r} m(r,u) && (5)\\
&\leq \sum_{u\ \in Group(r), u\neq r} |r| + |u| - 2\tau\sqrt{|u||r|} && (6)\\
&\leq \sum_{u\ \in Group(r), u\neq r} (\sqrt{|r|} - \sqrt{|u|})^2 + 2(1-\tau)\sqrt{|u||r|} && (7)
\end{aligned}$$

We now make the simplifying assumption that each row in the group has the same length μ. Then,

$$\text{Fill}(Group(r)) \leq \sum_{u\ \in Group(r), u\neq r} 2(1-\tau)\mu \leq 2(|Group(r)| - 1)\mu(1-\tau)$$

We can now add these bounds for all groups g,

$$\text{Fill}_{total} \leq 2(1-\tau)\sum_{g}(|g| - 1)\mu_g$$

Since $|g|\ \mu_g$ is the number of (original) nonzero elements in the block, this is close to $2(1-\tau)NZ(A)$. We can reformulate this slightly by introducing μ_a the average row-length, i.e., the average number of nonzero elements per row, and observing that the sum of μ_g over all n_b groups found is precisely $n_b \times \mu_a$, where $NZ(A)$ is $n \times \mu_a$. This leads to the following proposition.

Proposition 2.1 *Assume that during Algorithm 2.2 (and 2.3) each group is such that the number of nonzero elements in its member rows is constant, and let n_b be the total number of groups found, and μ_a the average number of nonzero elements per row. Then, the total fill-in introduced by Algorithm 2.2 (and 2.3) is such that*

$$Fill_{total} \leq 2(1-\tau)\mu_a\,(n - n_b) \tag{8}$$

Notice in particular that, as expected, the fill-in is void when the number of blocks is n, i.e., when no blocking is discovered. It is also important to realize that we obtained this upper bound by making a simplifying assumption based on ignoring the fact that some of the fill-in when $s > 2$ may be counted more than once. This provides a simple yet close enough bound for the fill-in introduced.

A consequence of the proposition is that the compression efficiency as defined in Table 3 and Table 4 satisfies:

$$\text{Eff} \leq \frac{1}{1 + 2(1-\tau)(1 - n_b/n)} .$$

3 Matrices with nonsymmetric patterns

Compression can also be used to unravel exploitable block structure for matrices with nonsymmetric patterns, though quotient graphs can no longer be used. For example, a column (only) block structure such as the one in the following matrix

$$A = \left[\begin{array}{cc|cc|ccc|c} x & x & 0 & 0 & x & x & x & 0 \\ x & x & x & x & x & x & x & 0 \\ 0 & 0 & x & x & 0 & 0 & 0 & x \\ x & x & 0 & 0 & 0 & 0 & 0 & x \\ x & x & x & x & x & x & x & 0 \\ 0 & 0 & x & x & x & x & x & 0 \\ x & x & 0 & 0 & x & x & x & 0 \\ x & x & 0 & 0 & 0 & 0 & 0 & x \end{array}\right]$$

could clearly be exploited in a row-oriented block-ILU.

There are however a few difficulties. Whereas in the symmetric case, a diagonal block is either a zero block or a square dense block, this is not guaranteed in the nonsymmetric case since the blocks are not necessarily square. As an example the second block column in the above matrix has as a diagonal block a (singular) 2×2 matrix with a zero second row. This situation does not arise if all the diagonal elements of A are non-zero and if exact blocking is performed ($\tau = 1$ for the angle and the hybrid algorithms). One way to avoid the difficulty is obviously to symmetrize the pattern, and this will be a good strategy in the rather common case when the pattern is nearly symmetric. If a small number of diagonal entries are zero, another strategy is to always treat diagonal entries as nonzero.

For matrices whose patterns are very far from being symmetric, compression is best be done in two stages. The column-compressed graph for the above matrix consists of considering each set of indistinguishable column as a node. This yields the following 8×4 adjacency matrix

$$\left[\begin{array}{cccc} x & 0 & x & 0 \\ x & x & x & 0 \\ 0 & x & 0 & x \\ x & 0 & 0 & x \\ x & x & x & 0 \\ 0 & x & x & 0 \\ x & 0 & x & 0 \\ x & 0 & 0 & x \end{array}\right]$$

Now a row-based compression can be applied to the above matrix, yielding the row-partition

$$Y_1 = \{1, 7\}, \quad Y_2 = \{2, 5\}, \quad Y_3 = \{3\}, \quad Y_4 = \{8\}\}$$

The row and column blockings may be quite different. In fact the number of groups may not even be the same as the number of groups found for the columns. However, now each block of the blocked matrix is a

Matrix	Hash	$\tau = .70$	$\tau = .75$	$\tau = .80$	$\tau = .90$	$\tau = .95$
			VBILU(0)			
BCSSTK10	119	46	40	37	54	54
BCSSTK11	300	80	86	80	85	140
BCSSTK12	300	80	86	80	85	140
BCSSTK16	47	42	39	40	47	44
RAEFSKY1	36	51	46	36	36	36
RAEFSKY2	45	56	49	45	45	45
RAEFSKY3	78	62	62	62	78	78
BARTHT1A	88	69	73	88	88	88
BARTHT2A	52	50	51	52	51	52
FIDAP006	300	300	300	300	300	300
FIDAP023	98	177	95	58	57	97
FIDAP028	106	300	300	300	300	127
VENKAT25	238	239	238	238	238	238
			VBILU(2)			
BCSSTK10	14	14	13	9	14	14
BCSSTK11	30	26	30	30	30	30
BCSSTK12	30	26	30	30	30	30
BCSSTK16	16	13	14	14	16	16
RAEFSKY1	19	21	21	19	19	19
RAEFSKY2	22	24	24	22	22	22
RAEFSKY3	50	49	49	49	50	50
BARTHT1A	30	30	30	30	30	30
BARTHT2A	23	19	21	22	23	23
FIDAP006	36	101	300	113	37	36
FIDAP023	25	22	17	17	18	19
FIDAP028	300	30	46	113	300	300
VENKAT25	87	87	87	87	87	87

Table 5: Iteration counts for GMRES(60) preconditioned with VBILU(k), for $k = 0, 2$, on a collection of linear systems

dense block in the case exact blocking is used. Computations can be reorganized in a complete or incomplete LU factorization to take advantage of this nonsymmetric blocking.

4 Numerical tests with VBILUK

The goal of the numerical experiments in this section is to illustrate how the methods described earlier can be integrated within a variable block ILU preconditioner with fill level (VBILUK). We have implemented and tested only a level-of-fill Block ILU technique, with variable blocks. All codes have been written in C, and unless otherwise stated, the experiments have been conducted on a PC with an Intel Pentium II processor with a clock speed of 450Mhz and 500MB of main memory. The timings in the following experiments were obtained with a $-O3$ optimization directive during compilation (in contrast with those of Tables 2, 3, and 4, of the previous sections where no optimization was used).

The principle of the block ILU preconditioner is straightforward so details will be omitted. It suffices to say that the ILU factorization is conceptually performed on the quotient matrix and the levels-of-fill are computed for the corresponding quotient graph. The inversion of the diagonal blocks is performed via a truncated SVD factorization, for better stability. Although this implementation does not seem to have been

Matrix	Hash	$\tau = .70$	$\tau = .75$	$\tau = .80$	$\tau = .90$	$\tau = .95$
VBILU(0)						
BCSSTK10	1.04	0.34	0.28	0.26	0.45	0.45
BCSSTK11	4.91	0.93	1.02	0.93	1.06	1.87
BCSSTK12	4.95	0.93	1.04	1.00	1.03	1.88
BCSSTK16	3.81	2.97	2.76	2.83	3.61	3.39
RAEFSKY1	2.31	3.21	3.07	2.27	2.29	2.40
RAEFSKY2	2.95	3.62	3.24	3.09	3.02	2.96
RAEFSKY3	22.76	18.81	18.61	19.38	22.60	23.02
BARTHT1A	10.72	8.66	9.12	10.90	10.93	10.72
BARTHT2A	13.29	14.77	13.88	13.71	13.26	13.42
FIDAP006	8.66	4.96	5.17	5.35	5.67	6.14
FIDAP023	2.95	2.78	1.51	0.92	0.97	1.79
FIDAP028	4.31	8.05	8.00	8.43	8.84	4.35
VENKAT25	157.59	155.76	151.38	152.91	157.96	153.56
VBILU(2)						
BCSSTK10	0.13	0.11	0.10	0.07	0.13	0.13
BCSSTK11	0.68	0.41	0.47	0.46	0.50	0.54
BCSSTK12	0.68	0.42	0.47	0.46	0.49	0.54
BCSSTK16	2.43	1.48	1.66	1.71	2.24	2.30
RAEFSKY1	2.88	2.85	3.00	3.11	2.87	2.95
RAEFSKY2	3.34	3.29	3.41	3.33	3.37	3.35
RAEFSKY3	19.30	18.96	18.77	19.17	19.04	19.29
BARTHT1A	4.55	4.50	4.62	4.55	4.68	4.54
BARTHT2A	10.74	10.35	10.22	10.06	10.73	10.81
FIDAP006	2.14	4.46	7.97	8.59	3.44	1.25
FIDAP023	1.61	0.52	0.41	0.41	0.48	0.62
FIDAP028	25.55	1.18	7.21	13.26	14.63	16.61
VENKAT25	67.33	65.82	65.41	65.89	66.26	65.84

Table 6: Iteration times for solving a collection of linear systems by GMRES(60) preconditioned with VBILU(k),for $k = 0, 2$

discussed in the literature, the simpler version of Block-ILU(k) with constant block size is well-established, see, for example, [6]. One problem with using a block ILU factorization with variable blocks, is precisely the fact that a blocking with variable blocks is not known in advance, or that it is difficult to handle practically. With automatic blocking, we only need to provide one parameter, namely the tolerance for grouping. If a hash-based algorithm is used, no parameter is required.

In this test we consider the impact of the grouping parameter τ on the performance of VBILUK. For this we took the same sample of the matrices tested in the previous tables. However we removed two matrices and added a new one. We removed `penalty` and `RAEFSKY4` because the methods did not converge for these two hard problems. We added VENKAT25, a matrix of dimension $n = 62,424$ with $nnz = 1,717,792$ nonzeros, which originates from a 2-D unstructured 2-D Euler solver[2]. VENKAT25 has a natural blocking with a block size of 4. All systems are solved by forming an artificial right-hand side and taking a random initial guess. The iteration was stopped when the residual norm was decreased by 10 orders of magnitude or when an iteration count of 300 was reached. The accelerator GMRES(60) was used in all tests.

Recall that as τ decreases down from 1, the blocks are likely to become larger. A rule of thumb is that we may expect the number of iterations to improve slightly. This is observed in general, but as Table

[2]Also available from the matrix market at `http://math.nist.gov/MatrixMarket/`

5 shows, not always. As can be seen, there is at least one case (FIDAP028) where the hash based algorithm (which is equivalent to setting $\tau = 1$), yields convergence whereas the other cases do not. For all other cases, the grouping provided by the hybrid algorithm provides better or comparable iteration counts for VBILU(0) and VBILU(2). Table 6 shows the corresponding times. As can be seen the fact ILU uses possibly much larger blocks does not penalize performance. In fact in many cases, the execution is at least marginally better than that obtained from the pure blocking yielded by the hash-based algorithm.

An interesting comment is in regards to matrices which already have a natural blocking. The times obtained for solving some of these systems hardly vary, because the number of GMRES iterations remains the same for all compressions. Two examples are `Venkat25` and `BARTH2A` when using $k = 2$. This is because the compressions obtained here are somewhat optimal, in that they are hard to improve by additional groupings.

To give an idea of the potential gains that can be achieved from a block version versus of point version of the same preconditioner, we compare in Table 7 the preprocessing and iteration times for VBILUK(2) and point ILUK(2) on two matrices under the same test conditions as those in the previous tests. The block version used the blocking provided by the hash-based algorithm (equivalent to setting $\tau = 1.0$ in the other two algorithms). As is known the point and block versions of ILU(k) are mathematically equivalent. As a result, the number of iterations and required memory should be identical for both, as the column "Its" in the table confirms. The column "Pr-Mem." reports the number of memory locations required by the preconditioners, which as expected are identical. The iterations times ("Pr-sec.") are close for both methods. In the current implementation of the code matrix-vector operations do not take advantage of blocking in the iteration phase. LAPACK [4] routines are extensively used in the construction phase and in the forward-backward sweeps. Thus, it is likely that additional gains in time can be made for the iteration phase by better code optimization. However, the gains made when constructing the preconditioner are substantial, reaching a reduction by a factor of about 4.6 for RAEFSKY3.

	VBILUK(2)				ILUK(2)			
Matrix	Pr-Mem.	Its	Its sec.	Pr-sec.	Pr-Mem.	Its	Its sec.	Pr-sec.
BARTHT1A	958775	30	4.53	1.02	958775	30	4.44	3.01
RAEFSKY3	2810240	50	19.33	3.41	2810240	50	21.57	15.84

Table 7: Performance of a block and point ILU(k) on two matrices

5 Conclusion

The methods presented in this paper have as their primary goal to automate blocking in block ILU factorizations. In order to allow for imperfect blocking, i.e., for blocking in which the entries in the blocks are allowed to be zeros, it is useful to be able to group rows according to the nearness of their patterns. This can be achieved by using the angle between the rows as a measure of nearness. The algorithms to accomplish this are inexpensive when compared for example to the cost of BILU(0), the least expensive factorization.

There are other possible applications of the blocking techniques presented here. In preconditioning methods, they can be combined with Algebraic Recursive Multilevel Solvers [12]. In a multilevel ILU context, the successive approximate Schur complements that are generated can become quite dense, and blocking can have a good performance pay-off. In fact, this is precisely the strategy utilized in sparse direct solution software to obtain flops rates that are far superior to those of iterative solvers.

Acknowledgements. The numerical experiments were conducted by Na Li, a graduate student in Computer Science at the University of Minnesota. The programs used in this paper were translated into C from FORTRAN-77 codes written by Andrew Chapman (see reference [6]). The author benefited from discussions on the topic of this paper with Patrick Amestoy and John Gilbert during a visit to CERFACS, France, supported by an NSF/INRIA grant. The Minnesota Supercomputer Institute provided computing resources and an excellent environment to conduct this research.

References

[1] C. Ahscraft. Compressed graphs and the minumum degree algorithm. *SIAM Journal on Scientific Computing*, 16:1404–1411, 1995.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[3] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. MUMPS Multifrontal Massively Parallel Solver version 2.0. Technical Report TR/PA/98/02, CERFACS, Toulouse, France, 1998.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. So ren sen. *LAPACK User's Guide*. SIAM, Philadelphia, PA, 1992.

[5] X. Cai and Å. Odegård. Parallel simulation of 3d nonlinear acoustic fields on a linux-cluster. In *Proceedings of CLUSTER 2000*, 2000. (http://www.ifi.uio.no/ xingca/xc_papers.shtml).

[6] A. Chapman, Y. Saad, and L. Wigton. High-order ILU preconditioners for CFD problems. *Int. J. Numer. Meth. Fluids*, 33:767–788, 2000.

[7] J. A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[8] G. Karypis and V. Kumar. A fast and high-quality multi-level scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1998.

[9] Å. Odegård, P. Fox, S. Holm, and A. Tveito. Finite element modelling of pulsed bessel beams and x-waves using diffpack. In *Proceedings of 25th International Imaging Symposium*, Bristol, United Kingdom, 2000.

[10] J. O'Neil and D. B. Szyld. A block ordering method for sparse matrices. *SIAM Journal on Scientific Computing*, 11:811–823, 1990.

[11] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[12] Y. Saad and B. Suchomel. ARMS: An algebraic recursive multilevel solver for general sparse linear systems. Technical Report umsi-99-107, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 1999.