Hypergraph-based Unsymmetric Nested Dissection Ordering for Sparse LU Factorization

Laura Grigori, Erik G. Boman, Simplice Donfack, Timothy A. Davis

R INRIA

Hypergraph-based Unsymmetric Nested Dissection Ordering for Sparse LU Factorization

Laura Grigori — Erik G. Boman — Simplice Donfack — Timothy A. Davis

N° 6520

Avril 2008

Thème NUM

R apport de recherche

Hypergraph-based Unsymmetric Nested Dissection Ordering for Sparse LU Factorization

Laura Grigori[*] , Erik G. Boman[†] , Simplice Donfack[‡] , Timothy A. Davis[§]

Abstract: In this paper we present HUND, a hypergraph-based unsymmetric nested dissection ordering algorithm for reducing the fill-in incurred during Gaussian elimination. HUND has several important properties. It takes a global perspective of the entire matrix, as opposed to local heuristics. It takes into account the assymetry of the input matrix by using a hypergraph to represent its structure. It is suitable for performing Gaussian elimination in parallel, with partial pivoting. This is possible because the row permutations performed due to partial pivoting do not destroy the column separators identified by the nested dissection approach. Experimental results on 27 medium and large size highly unsymmetric matrices compare HUND to four other well-known reordering algorithms. The results show that HUND provides a robust reordering algorithm, in the sense that it is the best or close to the best (often within 10%) of all the other methods.

Key-words: sparse LU-factorization, reordering techniques, hypergraph partitioning, nested dissection

[*] INRIA Saclay - Ile de France, Laboratoire de Recherche en Informatique Universite Paris-Sud 11, France (`laura.grigori@inria.fr`).

[†] Scalable Algorithms Dept., Sandia National Laboratories, NM 87185-1318, USA, Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000. This work was supported by the US DOE Office of Science through the CSCAPES SciDAC institute (`egboman@sandia.gov`).

[‡] Universite de Yaounde I, Computer Science Department, B.P 812 Yaounde - Cameroun, the work of this author was performed during his Master at INRIA Saclay through the INRIA Internship program (`sidonfack@gmail.com`).

[§] CISE Dept., University of Florida, supported by the National Science Foundation (0620286) (`davis@cise.ufl.edu`).

Dissection Emboîtée Non-Symmetrique Basée sur des Hypergraphes pour la Factorisation LU Creuse

Résumé : Dans ce papier nous présentons HUND, un algorithme de renumérotation d'inconnues pour la factorization LU creuse, utilisant une approche de dissection emboîtées et des hypergraphes. HUND a quelques propriétés importantes. Il utilise une approche globale de la matrice d'entrée. Il prends en compte la non-symétrie de la matrice en utilisant un hypergraphe pour représenter sa structure. Il peut être utilisé dans un algorithme parallèle pour efectuer l'élimination de Gauss avec pivotage partiel. Ceci est possible car les permutations de lignes effectuées due au pivotage partiel ne détruisent pas les séparateurs de colonnes identifiés par l'approche dissection emboîtée. Les résultats expérimentaux obtenus sur 27 matrices de taille moyenne et grande et avec une structure très non-symétrique compare HUND avec quatre autre heuristiques de renumérotation très utilisées. Les résultats montrent que HUND est un algorithme de renumérotation robuste, car il donne ou bien le meilleur résultat, ou très prôche du meilleur résultat.

Mots-clés : factorisation LU creuse, renumérotation des inconnues, partitionnement de hypergraphes, dissection emboîtée

1   Introduction

Solving a linear system of equations $Ax = b$ is an operation that lies at the heart of many scientific applications. We focus on sparse, general systems in this paper. Gaussian elimination can be used to accurately solve these systems, and consists in decomposing the matrix A into the product of L and U, where L is a lower triangular matrix and U is an upper triangular matrix. One of the characteristics of Gaussian elimination is the notion of a fill element, which denotes a zero element of the original matrix that becomes nonzero in the factors L and U due to the operations associated with the Gaussian elimination. Hence one of the important preprocessing steps preceding the numerical computation of the factors L and U consists in reordering the equations and variables such that the number of fill elements is reduced.

Although this problem is NP-complete [33], in practice there are several efficient fill reducing heuristics. They can be grouped into two classes. The first class uses local greedy heuristics to reduce the number of fill elements at each step of Gaussian elimination. One of the representative heuristics is the minimum degree algorithm. This algorithm uses the graph associated with a symmetric matrix, and chooses at each step to eliminate the row corresponding to the vertex of minimum degree. Several variants, such as multiple minimum degree [28] (MMD) and approximate minimum degree [1] (AMD), improve the minimum degree algorithm, in terms of time and/or memory usage.

The second class is based on global heuristics and uses graph partitioning to restrict the fill to only specific blocks of the permuted matrix. Nested dissection [15] is the main technique used in the graph partitioning approach. This algorithm uses the graph of a symmetric matrix and employs a top-down divide-and-conquer paradigm. The graph partitioning approach has the advantage of reordering the matrix into a form suitable for parallel execution. State-of-the-art nested dissection algorithms use multilevel partitioning [20, 25]. A widely used routine is METIS_NODEND from the METIS [24] graph partitioning package.

It has been observed in practice that minimum degree is better at reducing the fill for smaller problems, while nested dissection works better for larger problems. This observation has lead to the development of hybrid heuristics that consist in applying several steps of nested dissection, followed by the usage of a variant of the minimum degree algorithm on local blocks [21].

For unsymmetric matrices, the above algorithms use the graph associated with the symmetrized matrix $A+A^T$ or $A^T A$. One additional algorithm, the column approximate minimum degree [9] (COLAMD), implements the approximate minimum degree algorithm on $A^T A$ without explicitly forming the structure of $A^T A$. The approach of symmetrizing the input matrix works well in practice when the matrix is almost symmetric. However, when the matrix is very unsymmetric, the information related to the asymmetry of the matrix is not exploited.

There are few approaches in the literature that aim at developing fill-reducing algorithms targeting unsymmetric matrices. For local heuristics, this is due to the fact that the techniques for improving the runtime of minimum degree are difficult to extend to unsymmetric matrices. In fact the minimum degree algorithm is related to the Markowitz algorithm [30], which was developed earlier for unsymmetric matrices. The Markowitz algorithm defines the degree of a vertex (called the Markowitz count) as the product of the number of nonzeros in the

row and the number of nonzeros in the column corresponding to this vertex. However, this algorithm is asymptotically slower than the minimum degree algorithm. A recent local greedy heuristic for unsymmetric matrices is presented in [2]. To obtain reasonable runtime, the authors use local symmetrization techniques and the degree of a vertex is given by an approximate Markowitz count.

In this paper we present one of the first fill-reducing ordering algorithms that fully exploits the asymmetry of the matrix and that is also suitable for parallel execution. It relies on a variation of nested dissection, but it takes into account the asymmetry of the input matrix by using a hypergraph to represent its structure. The algorithm first computes a hyperedge separator of the entire hypergraph that divides it into two disconnected parts. The matrix is reordered such that the columns corresponding to the hyperedge separator are ordered after those in the disconnected parts. The nested dissection is then recursively applied to the hypergraph of the two disconnected parts, respectively. The recursion can be stopped at any depth.

An important property of our approach is that the structure of the partitioned matrix is insensitive to row permutations. In other words, the row permutations induced by pivoting during Gaussian elimination do not destroy the column separators. Hence the fill is reduced because it can occur only in the column separators and in the disconnected parts of the matrix. But also this property is particularly important for a parallel execution, since the communication pattern, which depends on the column separators, can be computed prior to the numerical factorization. In addition, the partitioning algorithm can be used in combination with other important techniques in sparse Gaussian elimination. This includes permuting large entries on the diagonal [12], a technique improving stability in solvers implementing Gaussian elimination with static pivoting [26].

We note that a partitioning algorithm that takes into account the asymmetry of the input matrix was also considered by Duff and Scott in [13, 14]. There are several important differences with our work. The authors focus on the parallel execution of LU factorization, and their goal is to permute the matrix to a so called singly-bordered block diagonal form. In this form the matrix has several diagonal blocks (which can be rectangular), and the connections between the different blocks are assembled in the columns ordered at the end of the matrix. The advantage of this form is that the diagonal blocks can be factorized independently, though special care must be taken since the blocks are often non-square. The authors do not analyze this approach for fill-reducing ordering. The core part of our method can be viewed as a recursive application of the Duff and Scott strategy, but for ordering, where non-square blocks are less problematic. We also gain similar advantages with respect to parallel execution.

The remainder of the paper is organized as follows. In Section 2 we give several basic graph theory definitions and we describe in detail the nested dissection process. In Section 3 we present our hypergraph based unsymmetric nested dissection algorithm and its different steps. In Section 4 we describe a possible variation of the algorithm that aims at decreasing the size of the separators. In Section 5 we present experimental results that study the effectiveness of the new algorithm, in terms of fill, on a set of highly unsymmetric matrices. We also compare the results with other fill-reducing ordering algorithms. Finally, Section 6 concludes the paper.

2 Background: Nested Dissection and Hypergraphs

2.1 Nested Dissection

Nested dissection [15, 27] is a fill-reducing ordering method based on the divide-and-conquer principle. The standard method only applies to symmetric matrices; here we show a nonsymmetric variation.

The sparsity structure of a structurally symmetric matrix is often represented as an undirected graph. The nested dissection method is based on finding a small vertex separator, S, that partitions the graph into two disconnected subgraphs. If we order the rows and columns corresponding to the separator vertices S last, the permuted matrix PAP^T has the form

$$\begin{bmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{13}^T & A_{23}^T & A_{33} \end{bmatrix} \tag{1}$$

where the diagonal blocks are square and symmetric. Now the diagonal blocks A_{11} and A_{22} can be factored independently and will not cause any fill in the zero blocks. We propose a similar approach in the nonsymmetric case, based on a column separator. Suppose we can permute A into the form

$$\begin{bmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & 0 & A_{33} \end{bmatrix} \tag{2}$$

where none of the blocks are necessarily square. (This is known as singly bordered block form.) Then we can perform Gaussian elimination and there will be no fill in the zero blocks. Furthermore, this property holds even if we allow partial pivoting and row interchanges. Note that if any of the diagonal blocks are square, then A is reducible and the linear systems decouple.

The question is how to obtain singly-bordered block structure with a small column separator. There are two common approaches: a direct approach [3, 23], and indirect methods that first find doubly-bordered block diagonal form [14]. We choose the direct method, and use hypergraph partitioning.

2.2 Hypergraph Partitioning and Ordering

Since graph models are limited to (structurally) symmetric matrices, either a bipartite graph or hypergraph must be used in the unsymmetric case. We prefer the hypergraph model. A hypergraph $H(V, E)$ contains a set of vertices V and a set of hyperedges E (also known as *nets*), where each hyperedge is a subset of V. We will use the column-net hypergraph model of a sparse matrix [4] where each row corresponds to a vertex and each column is a hyperedge. Hyperedge e_j contains the vertices given by the nonzero entries in column j. An example of a matrix A is given in Equation 3 and its hypergraph is shown in Figure 1.

$$A = \begin{pmatrix} x & & & & & & & x \\ & x & & & x & x & & \\ & & x & & x & & & x \\ & x & & x & x & & & x \\ x & & x & & x & & & \\ & & & & x & & x & x \\ & & & x & & & & x \\ x & & x & & & & x & x \end{pmatrix} \tag{3}$$
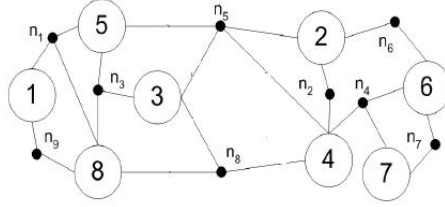
Figure 1: Hypergraph of matrix A in Equation 3. The large circles are vertices, and the small black dots represent hyperedges (nets).

Suppose we partition the vertices (rows) into two sets, R_1 and R_2. This induces a partition of the hyperedges (columns) into three sets: C_1, C_2, and C_3. Let C_1 be the set of hyperedges (columns) where all the vertices (rows) are in R_1. Similarly, let C_2 be the set of hyperedges (columns) where all the vertices (rows) are in R_2. Let C_3 be the set of hyperedges (columns) that are "cut", that is, they have some vertices in R_1 and some in R_2. Now let P be a permutation such that all of R_1 is ordered before R_2, and let Q be a similar column permutation. Then

$$PAQ = \left[\begin{array}{ccc} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \end{array} \right] \tag{4}$$

It may happen that some rows in A_{11} or A_{22} are empty (all zero). In this case, permute such rows to the bottom and we get

$$\bar{P}AQ = \left[\begin{array}{ccc} \bar{A}_{11} & 0 & \bar{A}_{13} \\ 0 & \bar{A}_{22} & \bar{A}_{23} \\ 0 & 0 & \bar{A}_{33} \end{array} \right] \tag{5}$$

Figure 2 displays the result of the first step of unsymmetric nested dissection applied on the hypergraph in Figure 1. The matrix obtained after permuting matrix A in Equation 3 is presented in Equation 6.
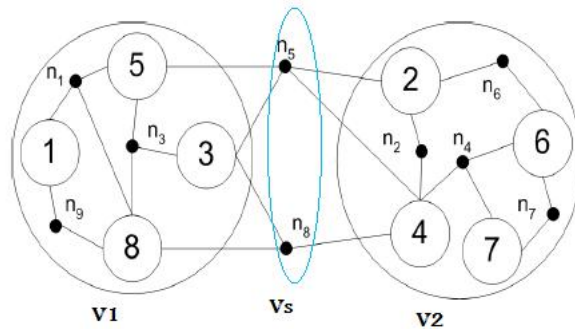

Figure 2: The result of the first step of unsymmetric nested dissection applied on the hypergraph in Figure 1, the hyperedge separator is $E_s = \{n_5, n_8\}$. Note that the separator only contains hyperedges and no vertices.

$$PAQ = \begin{pmatrix} x & & x & & & & & \\ & x & & & & & x & x \\ x & x & & & & & & x \\ x & x & x & & & & & x \\ & & & x & & x & & x \\ & & & x & x & & & x \\ & & & & x & x & x & \\ & & & & x & & x & \end{pmatrix} \tag{6}$$

Hypergraph partitioning has been well studied [4]. The objective is to partition the vertices into two parts such that there are approximately equally many vertices in each part, and the number of cut hyperedges is minimized. Although it is an NP-hard problem, fast multilevel algorithms give good solutions in practice. Good software is readily available, like PaToH, hMetis, and Zoltan. The k-way partitioning problem ($k > 2$) is usually solved by recursive bisection. We note that the MONET algorithm [23] is precisely hypergraph partitioning applied to sparse matrices.

One disadvantage of hypergraph partitioning is that it can take relatively long time. Fortunately, methods based on graph partitioning and vertex separators are faster and can also produce the desired singly bordered block form [14] but we do not explore this option here.

3 An Unsymmetric Nested Dissection Algorithm

We present an algorithm with three stages. First, we apply hypergraph-based nested dissection to limit the fill. Second, we perform row interchanges based on numerical values to reduce pivoting. Third, we apply a local reordering on local blocks to again reduce fill.

3.1 Hypergraph Recursive Ordering

Recall our goal is to find permutations P_1 and Q_1 such that $(P_1 A Q_1)(Q_1^T x) = P_1 b$ is easier to solve than the original system $Ax = b$. Our idea is to apply the block decomposition (4) recursively. This is quite different from most recursive approaches for solving linear systems, because our blocks are usually not square. Our approach works because we only produce orderings recursively, and do not factor or solve recursively (using the matrix blocks).

Figure 3.1 (left) shows the sparsity structure of $P_1 A Q_1$ after two levels of bisection. We continue the recursive bisection until each block is smaller than a chosen threshold. As in symmetric nested dissection, we expect it is beneficial to switch to a local ordering algorithm on small blocks but in principle one could continue the recursion until each block has only one row or column. We sketch the recursive ordering heuristic in Algorithm 1. In this variant, the recursion stops at a constant block size, t_{min}.

3.2 Stabilization

The ordering procedure above only takes the structure into account, and not the numerical values. To stabilize the factorization and minimize pivoting, we wish to permute large entries to the diagonal. A standard approach is to model this as matching in the bipartite graph [12], and we can use the HSL [22] routine MC64. We use the matching permutation to permute the rows as shown in Figure 3.1 (right). Observe that after row permutation, the diagonal blocks are now square. The remaining rows in the originally rectangular blocks have been "pushed down". All the permutations applied on the matrix after this step should be symmetric.
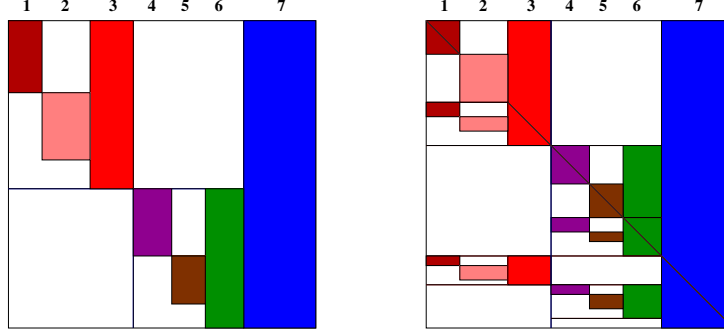
Figure 3: The matrix after hypergraph ordering (left) and after row permutation from matching (right).

Algorithm 1 Hypergraph Unsymmetric Nested Dissection

1: Function $[p, q] = \text{HUND}(A)$
2: $[m, n] = \text{size}(A)$
3: **if** $\min(m, n) \leq t_{min}$ **then**
4:     p= 1:m
5:     q= 1:n
6: **else**
7:     Create column-net hypergraph H for A
8:     Partition H into two parts using hypergraph partitioning
9:     Let p and q be the row and column permutations, respectively, to permute A into the block structure in Eq. 4
10:     $[m_1, n_1] = \text{size}(A_{11})$
11:     $[p_1, q_1] = \text{HUND}(A_{11})$
12:     $[p_2, q_2] = \text{HUND}(A_{22})$
13:     $p = p(p_1, p_2 + m_1)$
14:     $q = q(q_1, q_2 + n_1)$
15: **end if**

This permutation step for obtaining a strong diagonal is helpful for dynamic (partial) pivoting methods, since the number of row swaps is significantly reduced, thereby speeding up the factorization process [12]. It is essential for static pivoting methods [26], because it decreases the probability of encountering small pivots during the factorization.

3.3   Local Reordering

The goal of the third preprocessing step is to use local strategies to further decrease the fill in the blocks of the permuted matrix. Algorithms as CAMD [6] (constrained AMD) or CCOLAMD [6] (constrained COLAMD) can be used for this step. These algorithms are based on COLAMD, respectively AMD, and have the property of preserving the partitioning obtained by the unsymmetric nested dissection algorithm. This is because in a constrained ordering method, each node belongs to one of up to n constraint sets. In our case, a constraint set corresponds to a separator or a partition. After the ordering it is ensured that all the nodes in set zero are ordered first, followed by all the nodes in set one, and so on.

A preprocessing step useful for the efficiency of direct methods consists of reordering the matrix according to a postorder traversal of its elimination tree. This reordering tends to group together columns with the same non zero structure, so they can be treated as a dense matrix during the numeric factorization. This allows for the use of dense matrix kernels during numerical factorization, improves the memory hierarchy usage, and hence leads to a more efficient numeric factorization.

In order to preserve the structure obtained in the previous steps, we compute the elimination tree corresponding to the diagonal blocks of the input matrix. Note that in practice, postordering a matrix preserves its structure but can change the fill in the factors L and U. We remark that the local reordering should be applied symmetrically, so that the diagonal is preserved.

3.4 Algorithm Summary

In summary, LU factorization with partial pivoting based on unsymmetric nested dissection contains several distinct steps in the solution process:

1. Reorder the equations and variables by using the HUND heuristic that chooses permutation matrices P_1 and Q_1 so that the number of fill-in elements in the factors L and U of $P_1 A Q_1$ is reduced.

2. Choose a permutation matrix P_2 so that the matrix $P_2 P_1 A Q_1$ has large entries on the diagonal. The above permutation helps ensure the accuracy of the computed solution. In our tests this is achieved using the HSL routine MC64 [12].

3. Find a permutation matrix P_3 using a local heuristic and a postorder of the elimination tree associated with the diagonal blocks such that the fill-in is further reduced in the matrix $P_3 P_2 P_1 A Q_1 P_3^T$. In our tests this is achieved using constrained COLAMD and postordering based on the row merge tree [17] of the diagonal blocks.

4. Compute the numerical values of L and U.

The execution of the above algorithm on a real matrix (FD18) is displayed in Figure 4. The structure of the original matrix is presented at top left. The structure obtained after the unsymmetric nested dissection HUND is presented at top right. The structure obtained after permuting to place large entries on the diagonal using MC64 is displayed at bottom left. And finally the structure obtained after the local ordering is displayed at bottom right.

4 Variations

We have presented an ordering method that uses both row and column permutations to reorder a matrix to reduce fill. We used a column separator approach based on the column-net hypergraph model, where rows are vertices. Another option is to use the row-net hypergraph model, where columns are vertices. The method will work as before, except now we find row separators instead of column separators. One step of hypergraph bisection gives the matrix block structure

$$PAQ = \begin{pmatrix} A_{11} & 0 & 0 \\ 0 & A_{22} & 0 \\ A_{31} & A_{32} & A_{33} \end{pmatrix}. \tag{7}$$

The row separator approach is advantageous when the row separator is smaller than the column separator. However, row permutations can now destroy the sparsity structure. This variation is thus not suitable for partial pivoting with row interchanges (though partial pivoting with column interchanges would be fine). For static pivoting, we can again use a permutation based on matching (MC64), but now we should permute columns not rows.
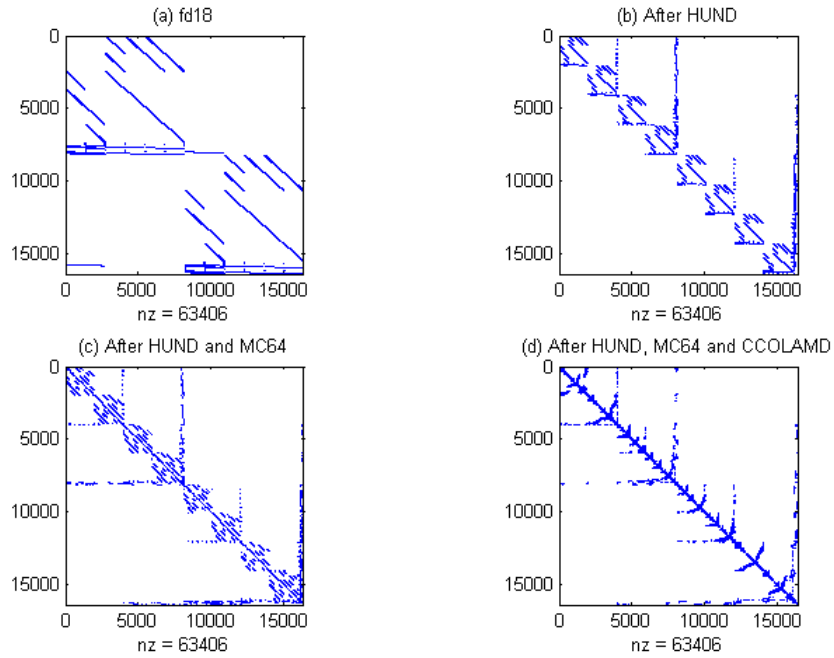
Figure 4: Example of application of preprocessing steps on a real matrix FD18. Displayed are (a) the structure of the original matrix FD18, (b) the structure obtained after HUND, (c) after MC64, and (d) after CCOLAMD
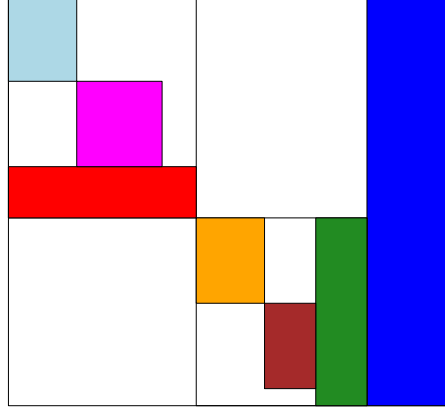
Figure 5: Example of Mondriaan variation. The top level separator is a column separator (blue), while one of the subproblems has a row separator (red) while the other has a column separator (green).

4.1   Row-or-column (Mondriaan) Approach

Since the best variation (row or column) depends on the matrix structure, an intriguing idea is to combine these two methods. The idea is to try both partitioning methods for every bisection, and pick the best. This gives a recursive decomposition that uses a combination of row and column separators. This is illustrated in Figure 4.1. We call this row-or-column hybrid method *Mondriaan* ordering, since it is similar to the Mondriaan method for sparse matrix partitioning [32], which also uses recursive hypergraph bisection with varying directions.

Obtaining a strong diagonal is a bit more difficult with the Mondriaan method. As usual, we compute a matching in the bipartite graph, but it is not obvious how to apply this as a permutation. A pure row or column permutation of the entire matrix will ruin the sparsity structure. Instead, parts of the matrix should be permuted by columns and other parts by rows. We omit the details here since it is difficult to describe.

We have not implemented the Mondriaan hybrid ordering, and leave it as future work to evaluate the potential improvement over the column-based method.

5   Experimental Results

In this section we present experimental results for HUND algorithm applied to real world matrices. The goal of the experiments is two-fold. First, we want to compare the performance of the new ordering algorithm with other widely used ordering algorithms as MMD, AMD, COLAMD and METIS (nested dissection). Second, we want to study the quality of the partitioning, in terms of size of the separators. As stated in the introduction, our goal is to reorder the matrix into a form that is suitable for parallel computation, while reducing or at least maintaining comparable the number of fill-in elements in the factors L and U to other state-of-art reordering algorithms. We show here that this goal is indeed achieved.

We use a set of highly unsymmetric matrices that represent a variety of application domains. We present in Table 1 their characteristics which include the matrix order, the number of nonzeros in the input matrix A, the numerical symmetry and the application domain. The matrices are grouped depending on their number of nonzeros,

#	Matrix	Order n	$nnz(A)$	Sym.	Application Domain
1	LNS_3937	3937	25407	0.14%	Linearized n.-s. compressible
2	FD18	16428	63406	0.00%	Crack problem
3	POLI_LARGE	15575	33074	0.05%	Chemical process simulation
4	BAYER04	20545	159082	0.02%	Chemical process simulation
5	SWANG1	3169	20841	0.00%	Semiconductor device sim
6	MARK3JAC020	9129	56175	1.00%	Economic model
7	LHR04	4101	82682	0.00%	Light hydrocarbon recovery
8	RAEFSKY6	3402	137845	0.00%	Incompressible flow
9	ZHAO2	33861	166453	0.00%	Electromagnetism
10	MULT_DCOP_03	25187	193216	1.00%	Circuit simulation
11	SHERMANACB	18510	145149	3.00%	Circuit simulation
12	JAN99JAC120SC	41374	260202	0.00%	Economic model
13	BAYER01	57735	277774	0.00%	Chemical process simulation
14	SINC12	7500	294986	0.00%	Single material crack problem
15	MARK3JAC140SC	64089	399735	1.00%	Economic model
16	ONETONE1	36057	341088	4.00%	Circuit simulation
17	AF23560	23560	484256	0.00%	Airfoil eigenvalue calculation
18	SINC15	11532	568526	0.00%	Single material crack problem
19	E40R0100	17281	553562	0.20%	Fluid dynamics
20	ZD_JAC2_DB	22835	676439	0.00%	Chemical process simulation
21	LHR34C	35152	764014	0.00%	Light hydrocarbon recovery
22	SINC18	16428	973826	0.00%	Single material crack problem
23	TWOTONE	120750	1224224	11.00%	circuit simulation
24	LHR71C	70304	1528092	0.00%	Light hydrocarbon recovery
25	TORSO2	115967	1033473	0.00%	Bioengineering
26	AV41092	41092	1683902	0.00%	Unstructured finite element
27	BBMAT	38744	1771722	0.06%	Computational fluid dynamics

Table 1: Benchmark matrices.

with no particular order within a group. For example, the first eleven matrices have less than $2 \cdot 10^5$ nonzeros, the following three matrices have less than $3 \cdot 10^5$ nonzeros, and so on. The matrices are available from University of Florida Sparse Matrix collection [7].

5.1   HUND versus other Reordering Algorithms: Results with SuperLU

We compare the ordering produced by HUND with four widely used fill-reducing ordering algorithms, that is MMD (applied on the structure of $A + A^T$ or on the structure of AA^T), AMD, COLAMD, and METIS nested dissection (applied on the structure of $A + A^T$ or on the structure of AA^T). The quality of each algorithm can be evaluated using several criteria, as the number of nonzero entries (nnz) in the factors L and U, the number of floating point operations performed during the numerical factorization, and the factorization time. We restrict our attention to the first criterion, the number of nonzeros in the factors L and U, since floating point operations are very fast on current computers while memory is often the bottleneck. (We also computed the number of operations and the results were quite similar to the nonzero counts.)

In our first set of tests we use LU factorization with partial pivoting implemented in the SuperLU solver [10]. SuperLU uses partial pivoting with threshold and chooses

in priority the diagonal element. In our experiments we use a threshold of 1, that is at each step of factorization the element of maximum magnitude in the current column of L is used as pivot. To evaluate HUND, the different preprocessing steps presented in section 3.4 are performed before the LU factorization with partial pivoting. That is, first the matrix is reordered using HUND heuristic. Second, the MC64 routine [12] is called to move large entries onto the diagonal. Third, the matrix is reordered using constrained COLAMD algorithm, as presented in [6], and based on a postorder traversal of the row merge tree [17] of the diagonal blocks. After these three preprocessing steps, the LU factorization with partial pivoting of SuperLU is called.

For the other reordering algorithms we use only two preprocessing steps. The matrix is scaled and permuted using MC64 in order to place large entries on the diagonal, and then a fill-reducing ordering is applied.

The HUND reordering heuristic presented in Algorithm 1 starts with the hypergraph of the input matrix and partitions it recursively into two parts. The recursion can be stopped either when a predefined number of parts is reached, or when the size of a part is smaller than a predefined threshold. In our tests we use PaToH [5] (with a fixed seed of 42 for the random number generator) to partition a hypergraph in two parts at each iteration of Algorithm 1. To study the performance of HUND we vary the number of parts in which the matrix is partitioned. When HUND partitions into a fixed number of parts, we present the results obtained for the number of parts (denoted as *kparts*) equal to 16 and 128. When HUND partitions until the size of each part is smaller than a given threshold, we present results for values of threshold (denoted as *tmin*) equal to 1 and 100.

Figure 6 compares HUND to the best result obtained for each matrix by one of the other reordering algorithms tested. It displays the ratio of the smallest number of nonzeros in the factors L and U obtained by one of the other reordering algorithms relative to the number of nonzeros in the factors L and U obtained by four versions of HUND (*kparts* = 16, *kparts* = 128, *tmin* = 1, *tmin* = 100). When this ratio is bigger than 1, HUND is the best reordering strategy. Figure 6 also shows the best reordering algorithm among MMD, COLAMD, AMD and METIS in term of fill-in. For completeness, we report in Table 3 of the Appendix, the results obtained for all the ordering strategies tested. The represented values are $nnz(L + U - I)/nnz(A)$. The cases represented in the table by "-" mean that SuperLU failed due to too much fill-in generated, and hence a memory requirement that exceeded the limits of our computer.

We observe that for half of the matrices in our test set, one variant of HUND induced the least fill-in compared to the other state-of-art reordering algorithms. For 3 other matrices, each of AMD, MMD and COLAMD produced the best results, while METIS produced the best result for 2 matrices. For 15 matrices, COLAMD produces results comparable to the best results.

For most of the matrices, the four variants of HUND produce comparable results.

As displayed in Table 3, the fill-in has a large value between 30 and 50 for the matrices MARK3JAC020, ZHAO2, SINC12, MARK3JAC140SC, SINC15, and SINC18 (numbers $6, 9, 14, 15, 18$ and 22). However for these matrices HUND produced the best, or very close to the best, results. The other reordering strategies lead generally to a larger number of fill-in elements. COLAMD leads to a fill-in factor between 42 and 116, and METIS ($A^T A$) leads to a fill-in factor between 32 and 66. The algorithms (MMD($A + A^T$) and AMD) fail for half of these matrices. Note that matrices SINC12, SINC15, and SINC18 come from the same application domain (single-material crack problem), as well as matrices MARK3JAC020 and MARK3JAC140SC (economic model).

In Figure 7 we restrict our attention to two variants of HUND (*kparts* = 128 and *tmin* = 1) and compare them with one of the best algorithms that uses a local strategy (COLAMD) and one of the best algorithms that uses a global approach (METIS applied to the structure of $A^T A$). The figure displays for each reordering algorithm the fill-in,
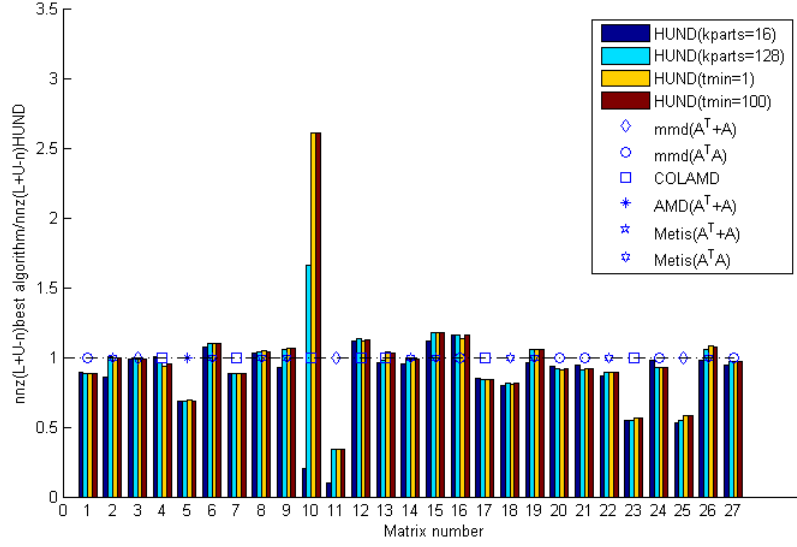
Figure 6: Ratio of number of nonzeros in the factors L and U produced by best state-of-art algorithm relative to four variants of HUND algorithm
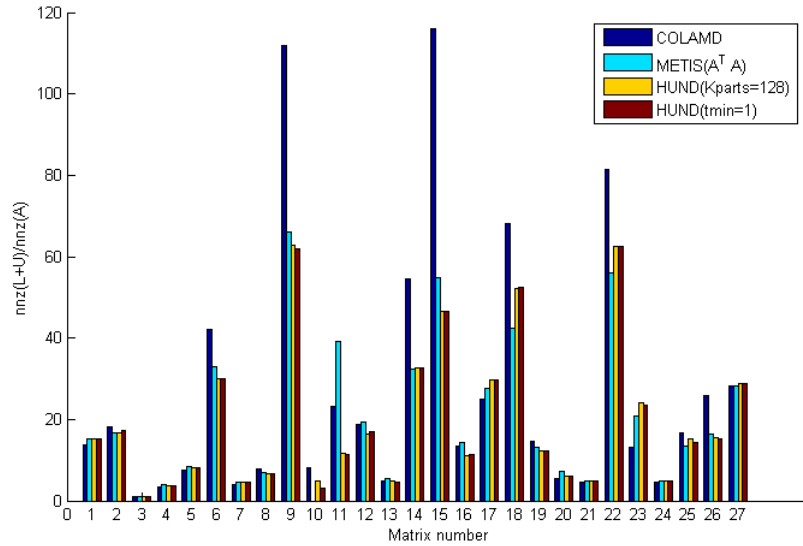

Figure 7: Comparison of two variants of HUND with COLAMD and METIS in terms of fill in the factors L and U, computed as $nnz(L + U - I)/nnz(A)$
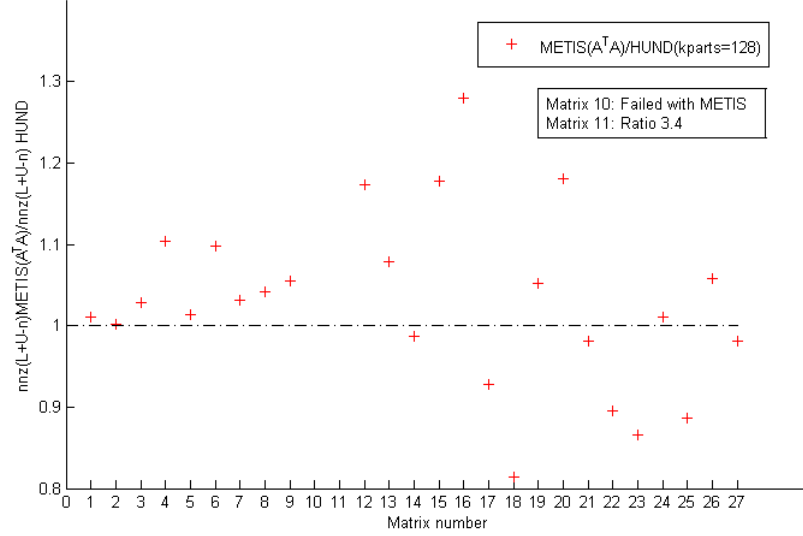
Figure 8: Ratio of the number of nonzeros in the factors of L and U obtained by METIS relative to HUND.

that is the ratio of the number of nonzeros of L and U to the number of nonzeros of A. There are several cases for which HUND significantly outperforms COLAMD.

Figure 8 presents more in detail the fill in the factors L and U obtained by the two global strategies in our tests, HUND (with *kparts* = 128) and METIS (applied on the structure of $A^T A$, which was better than $A + A^T$). We see that for most (about two thirds) of the matrices in our test set HUND outperforms METIS. The best result is obtained for matrix SHERMANACB, for which HUND leads to 3 times less fill than METIS.

5.2 Results with UMFPACK

UMFPACK is a right-looking multifrontal method which factorizes a sparse matrix using a sequence of frontal matrices. A frontal matrix is a small dense matrix F that holds $k > 1$ pivot rows and columns in their entirety. The lower right portion of the frontal matrix is a contribution block C (or Schur complement) that remains after the leading part is factorized into the corresponding k rows of U and columns of L:

$$ F = \left[\begin{array}{cc} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array} \right] = \left[\begin{array}{cc} L_{11} & 0 \\ L_{21} & I \end{array} \right] \times \left[\begin{array}{cc} U_{11} & U_{12} \\ 0 & I \end{array} \right] + \left[\begin{array}{cc} 0 & 0 \\ 0 & C_{22} \end{array} \right] $$

The ordering strategy in UMFPACK combines a fill-reducing symbolic preordering with adjustments made during numeric factorization.

As the first step of fill-reducing ordering, all pivots with zero Markowitz cost (referred to as *singletons*) are removed from the matrix. Suppose a column j exists with one nonzero entry a_{ij}. Then row i is removed from A and becomes the first row of U; column j is also removed and becomes the first column of L. This process repeats until all remaining columns have two or more nonzero entries. Next, a similar process is used for rows with one nonzero entry. These entries would be the leading and trailing

1-by-1 blocks of a block upper triangular form (Dulmage-Mendelsohn decomposition) if UMFPACK were to compute such a form. This singleton removal process takes $O(|A|)$ time, where $|A|$ is the number of entries in A; a complete permutation to block triangular form can take much more time. Most matrices seen in practice that are reducible to block triangular form have many such leading and trailing singleton blocks, and finding these singletons is often sufficient for obtaining the benefits of the more-costly block triangular form.

If both the rows and columns lower or upper triangular n-by-n matrix are arbitrarily permuted, the singleton removal process will find n column singletons and will permute A into upper triangular form. No numerical factorization is then needed. One matrix in the test collection (RAEFSKY6) falls into this category.

In the current version of UMFPACK (v5.2), singleton removal is always performed. For these experiments (and in a future release of UMFPACK) we have added a parameter that allows the removal of singletons to be disabled. By default, singletons are removed. We added this option for two reasons:

1. Removing singletons can restrict the fill-reducing ordering applied to the matrix. If removing of singletons causes an unsymmetric permutation to the matrix, then the symmetric and 2-by-2 strategies are disabled (even if explicitly requested by the user). For most matrices, such unsymmetric permutations are clear indicators that the unsymmetric strategy should be used to obtain the best fill-in. For a very few matrices this can be a poor choice of ordering heuristics, however. More pertinent to this paper, however, it also confounds our experimental results, since requesting the symmetric ordering (with AMD or METIS) would otherwise lead to COLAMD being used instead.

2. Removing singletons does not diminish the numerical accuracy of the LU factorization, but it can result in a matrix L that is not well-conditioned. With normal partial pivoting, L is always well-conditioned and any ill-conditioning of A is contained in U. Some applications (linear programming solvers, for example [31]) require a well-conditioned L.

After singletons are removed, the columns (and perhaps the rows) of the remain matrix are permuted to reduce fill-in in the LU factorization, using one of the four following strategies:

1. The **auto** strategy is the default: UMFPACK examines the nonzero pattern of the matrix A and the entries on the diagonal of A, and selects one of the other three strategies automatically. This usually leads to the best choice, but the user can select one of the other strategies manually.

2. The **unsymmetric** strategy is best suited for matrices with an unsymmetric nonzero pattern. A column ordering Q is chosen that minimizes the fill-in in the Cholesky factorization of $Q^T A^T A Q$, or identically the R factor in the QR factorization of AQ. Normally, a slightly modified version of COLAMD is used [9]; this algorithm is the same as COLAMD except that additional information about each frontal matrix is collected for the subsequent numerical factorization. This unsymmetric strategy is best suited for use with the HUND ordering. All results that we present with HUND use this strategy. With the unsymmetric strategy, UMFPACK can use COLAMD or HUND (applied to A) or either AMD or METIS (applied to the pattern of $A^T A$).

3. The **symmetric** strategy is selected for matrices with a nonzero-free diagonal (or mostly so) symmetric (or nearly symmetric) nonzero pattern. A symmetric ordering P is found that minimizes the fill-in in the Cholesky factorization of the matrix $P(A + A^T)P^T$. With the symmetric strategy, UMFPACK can use AMD or METIS (both applied to the pattern of $A + A^T$).

4. The **2-by-2** strategy is a pre-processing step followed by the symmetric strategy. In the pre-processing step, rows are exchanged to increase the number of nonzeros on the diagonal. None of our results in this paper use this strategy.

The next step is the numerical factorization, which can revise the ordering computed by the symbolic preanalysis.

In the symbolic preanalysis, the size of each frontal matrix F is bounded by the frontal matrix that would arise in a sparse multifrontal QR factorization. Since this can be much larger what is needed by an LU factorization, columns within each frontal matrix (the columns of A_{1*}) are reordered during numerical factorization to further reduce fill-in. This column reordering is only performed for the unsymmetric strategy; it is not performed by SuperLU.

Numerical threshold partial pivoting is used to select pivots within the A_{11} and A_{21} part of F. If the symmetric or 2-by-2 strategy is used, a strong preference is given to the diagonal entry. SuperLU also has these options.

Since UMFPACK is a right-looking method, it can consider the sparsity of a candidate pivot row when deciding whether or not to select it. This is a key advantage over left-looking methods such as Gilbert-Peierl's LU [16], SuperLU [10], and the implementation of Gilbert-Peierl's left-looking sparse LU in CSparse [8]. Left-looking methods cannot consider the sparsity of candidate pivot rows, since the matrix to the right of the pivot column has not yet been updated when the pivot row is selected.

Removing singletons prior to the fill-reducing ordering and factorization can have a dramatic impact on fill-in. This is another reason why UMFPACK can have a lower fill-in than SuperLU, but it is not intrinsic to the algorithm used by UMFPACK. That is, singleton removal is independent of the method used to factorize the remaining matrix, and any LU factorization method (including left-looking methods such as SuperLU) could use it as well.

There are thus four primary differences between UMFPACK and SuperLU which affect the results presented in this paper. The first three are detail of the implementation and not intrinsic to the methods used in UMFPACK and SuperLU.

1. UMFPACK removes singletons prior to factorization; SuperLU does not.

2. UMFPACK can selects ordering strategy automatically (unsymmetric, symmetric, or 2-by-2); SuperLU does not.

3. UMFPACK revises its column orderings within each frontal matrix to reduce fill-in; SuperLU does not revise the column orderings with each supercolumn.

4. UMFPACK can select a sparse pivot row; SuperLU cannot.

In our results, we disable the automatic strategy selection. Instead, we use the unsymmetric strategy for COLAMD, METIS (applied to $A^T A$) and HUND. We use the symmetric strategy with singletons disabled for AMD and METIS (applied to $A + A^T$). Complete results are shown in Table 4. Figure 9 displays a performance profile of just four of the unsymmetric orderings (COLAMD, HUND with $kparts = 16$ and $t = 100$, and METIS applied to $A^T A$). Overall, HUND provides a robust ordering with a performance profile superior to both COLAMD and METIS. We can notice that for 70% of the matrices, the performance of HUND ($kparts = 16$) is within 10% of the best performance. A similar profile for the same four methods is given for the SuperLU results in Figure 10.

5.3 Quality of the Partitioning

In this section we study the separators obtained during HUND's unsymmetric nested dissection, when $kparts = 128$. These separators are important because they tend to have an impact on the fill in the factors L and U as well as on the suitability of the reordered matrix for parallel execution. Fill in the factors can occur only in the
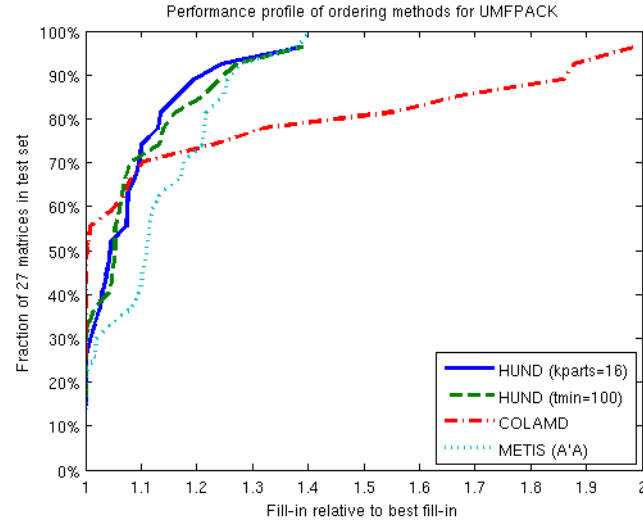
Figure 9: Performance profile of ordering methods with UMFPACK. Closer to 1, better the performance is.
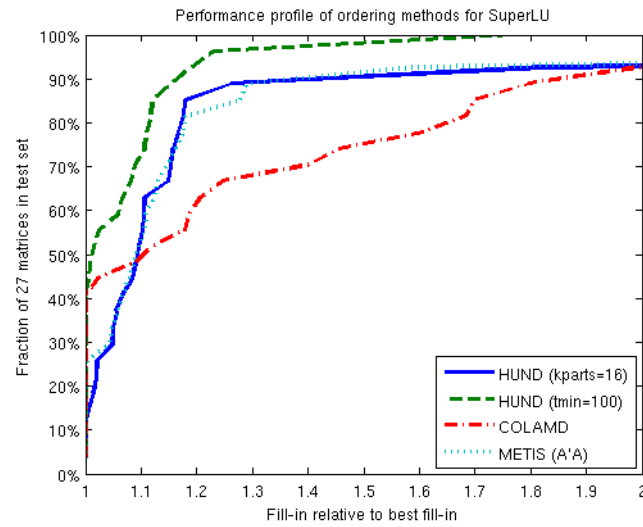

Figure 10: Performance profile of ordering methods with SuperLU. Closer to 1, better the performance is.

separators, hence smaller the separator size, fewer nonzeros should be obtained in the factors. In a parallel execution, the communication will incur during the factorization of the separators. Hence the communication will be reduced for separators of a smaller size.

#	Column Counts			Nonzero counts		
	Lvl 1	Lvl 2	Lvl 3	Lvl 1	Lvl 2	Lvl 3
1	3.33 / 3.33	3.33 / 3.33	2.05 / 2.39	3.82 / 3.82	3.82 / 3.82	2.39 / 2.77
2	0.85 / 0.85	0.91 / 0.91	0.62 / 0.66	1.18 / 1.18	1.06 / 1.07	0.64 / 0.69
3	0.65 / 0.65	0.18 / 0.28	0.13 / 0.37	1.09 / 1.09	0.27 / 0.42	0.27 / 0.82
4	0.40 / 0.40	0.24 / 0.25	0.23 / 0.31	1.03 / 1.03	0.68 / 0.81	0.70 / 0.97
5	2.97 / 2.97	1.47 / 1.51	1.32 / 1.45	3.13 / 3.13	1.54 / 1.59	1.39 / 1.53
6	4.44 / 4.44	5.13 / 5.14	1.18 / 1.30	7.12 / 7.12	12.14 / 12.16	1.80 / 2.03
7	2.66 / 2.66	1.89 / 2.27	1.38 / 1.51	4.14 / 4.14	2.88 / 3.22	1.95 / 2.54
8	13.43 / 13.43	8.14 / 8.76	3.29 / 4.61	16.36 / 16.36	10.69 / 10.86	4.13 / 5.93
9	1.83 / 1.83	1.37 / 1.40	0.90 / 0.93	1.83 / 1.83	1.45 / 1.48	0.90 / 0.92
10	43.71 / 43.71	11.51 / 23.02	2.90 / 11.59	63.23 / 63.23	8.73 / 17.46	2.12 / 8.49
11	29.39 / 29.39	10.00 / 19.99	3.13 / 12.52	45.78 / 45.78	10.07 / 20.13	3.08 / 12.32
12	1.07 / 1.07	1.07 / 1.07	0.84 / 0.97	7.94 / 7.94	7.84 / 7.92	4.13 / 5.53
13	0.14 / 0.14	0.08 / 0.11	0.06 / 0.09	0.26 / 0.26	0.14 / 0.15	0.11 / 0.17
14	15.33 / 15.33	15.69 / 15.87	2.48 / 4.41	20.47 / 20.47	18.37 / 18.40	3.01 / 5.32
15	0.63 / 0.63	0.65 / 0.65	0.64 / 0.65	1.00 / 1.00	1.02 / 1.03	1.00 / 1.03
16	2.27 / 2.27	0.91 / 0.95	0.51 / 0.73	16.12 / 16.12	4.59 / 4.77	1.86 / 4.18
17	2.58 / 2.58	2.58 / 2.58	2.29 / 2.41	2.59 / 2.59	2.59 / 2.59	2.33 / 2.44
18	31.22 / 31.22	7.80 / 7.80	2.52 / 3.10	36.85 / 36.85	10.15 / 10.17	2.92 / 3.63
19	3.61 / 3.61	1.83 / 2.05	1.70 / 1.96	4.14 / 4.14	2.09 / 2.35	1.94 / 2.25
20	0.00 / 0.00	0.60 / 1.20	0.50 / 1.02	0.00 / 0.00	1.56 / 3.11	1.33 / 2.71
21	0.18 / 0.18	0.42 / 0.46	0.34 / 0.47	0.16 / 0.16	0.64 / 0.71	0.47 / 0.61
22	31.56 / 31.56	7.88 / 7.88	2.43 / 2.68	37.35 / 37.35	10.12 / 10.12	2.82 / 3.12
23	1.71 / 1.71	0.38 / 0.74	0.25 / 0.95	3.71 / 3.71	1.32 / 2.60	0.14 / 0.45
24	0.15 / 0.15	0.11 / 0.13	0.20 / 0.23	0.21 / 0.21	0.11 / 0.15	0.29 / 0.35
25	0.45 / 0.45	0.17 / 0.22	0.20 / 0.26	0.45 / 0.45	0.17 / 0.22	0.20 / 0.26
26	4.23 / 4.23	1.66 / 2.65	0.65 / 1.23	29.43 / 29.43	12.34 / 23.91	2.74 / 6.81
27	2.75 / 2.75	2.40 / 2.40	2.37 / 2.40	1.28 / 1.28	2.94 / 2.97	2.63 / 2.77

Table 2: Percentage of columns and of nonzeros in the separators corresponding to the first three levels of unsymmetric nested dissection HUND with *kparts* = 128. The values are displayed as AVG/MAX, and compute *separatorsize*/n * 100, where n is the order of the matrix.

Table 2 present the number of columns and the number of nonzeros in the separators obtained at each step of the unsymmetric nested dissection. In these tables, level i denotes the step i of unsymmetric nested dissection. For each level we determine the maximum and the average separator size. The values displayed are *separatorsize*/n * 100. We display the results only for the first three levels, since the results for the other levels were in general less than 1%. Note that a value of 0.0 denotes a small value rounded to zero.

For several matrices, the number of columns and the number of nonzeros in the separators of the first and the second level are very large. For example, for matrix MULT_DCOP_03 (number 10), 43.4% of the columns are in the first level separator, and an average of 11.5% of the columns are in the second level separator. Matrices

SHERMANACB, SINC12, SINC15, and SINC18 (numbers 11, 14, 18, 22) have more than 15% of the columns in the first level separator. As already previously observed and as reported in Table 3, for matrices in SINC family, HUND leads to a high amount of fill in the factors L and U, between 32 and 62. This observation shows that the size of the separator has an important impact on the quality of the reordering, that is the number of nonzeros in the factors L and U.

However this is not always true. For example the matrices ZHAO2 and MARK3JAC140SC (numbers 9, 15) have separators of small size. But the fill in the factors L and U is high, 61 for ZHAO2 and 46 for MARK3JAC140SC.

6   Conclusions

We have presented a new ordering algorithm (HUND) for unsymmetric sparse matrix factorization, based on hypergraph partitioning and unsymmetric nested dissection. To enhance performance, we proposed a hybrid method that combines the nested dissection with local reordering. Our method allows partial pivoting, without destroying sparsity. We have tested the method using SuperLU and UMFPACK, two well-known partial pivoting LU codes. Empirical experiments show that our method is highly competitive with existing ordering methods. In particular, it is robust in the sense that it in most cases (23 out of 27 in our study) it performs close to the best of all the other existing methods (often within 10%). Thus, it is a good choice as an all-purpose ordering method.

The HUND method was designed for parallel computing, though we only evaluated it in serial here. The recursive top-down design allows coarse-grain parallelism, as opposed to local search methods like AMD and COLAMD. For symmetric systems, nested dissection ordering is considered superior for large systems and it is reasonable to expect the same holds for unsymmetric systems. The most expensive part of HUND is hypergraph partitioning, which can be done efficiently in parallel using the Zoltan toolkit [11]. The matching for strong diagonal can also be performed in parallel [29], though no parallel MC64 is yet available. Local reordering can be done locally in serial. Thus, our approach is well suited for fully parallel solvers that aim at being time and memory scalable [18, 26].

There are several directions for future work. First, we only used the default settings for the hypergraph partitioner. By default, Patoh tries to achieve a balance of 3%. This is quite strict, and perhaps a looser tolerance would work better in HUND. We can likely reduce fill and operation count by allowing more imbalance in the partitioning (and thus the elimination tree). A second future optimization is the "Mondriaan" version with varying directions outlined in section 4. A third direction is to study hybridization with other local (greedy) ordering methods, in particular, the recent unsymmetric method by Amestoy, Li, and Ng [2].

References

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17:886–905, 1996.

[2] P. R. Amestoy, X. S. Li, and E. G. Ng. Diagonal Markowitz scheme with local symmetrization. *SIAM J. Matrix Anal. Appl.*, 29(1):228–244, 2007.

[3] C. Aykanat, A. Pınar, and U. V. Çatalyürek. Permuting sparse rectangular matrices into block-diagonal form. *SIAM J. Sci. Comp.*, 26(6):1860–1879, 2004.

[4] U. V. CatalyÃ¼rek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transaction on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[5] U. V. CatalyÃ¼rek and C. Aykanat. Patoh: Partitioning tool for hypergraphs. User's guide, 1999.

[6] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 8xx: CHOLMOD, sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Softw.*, 200x. (to appear).

[7] T. Davis. University of Florida Sparse Matrix Collection. NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 1997. http://www.cise.ufl.edu/research/sparse/matrices.

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

[9] T. A. Davis, J. R. Gilbert, S. Larimore, and E. Ng. Algorithm 836: Colamd, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Software*, 30(3):377–380, 2004.

[10] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *SIAM J. Mat. Anal. Appl.*, 20(3):720–755, 1999.

[11] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

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

[13] I. S. Duff and J. A. Scott. A parallel direct solver for large sparse highly unsymmetric linear systems. *ACM Trans. Math. Software*, 30(2):95–117, 2004.

[14] I. S. Duff and J. A. Scott. Stabilized bordered block diagonal forms for parallel sparse solvers. *Parallel Computing*, 31:275–289, 2005.

[15] A. George. Nested dissection of a regular finite-element mesh. *SIAM J. Numerical Analysis*, 10:345–363, 1973.

[16] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Stat. Comput.*, 9:862–874, 1988.

[17] L. Grigori, M. Cosnard, and E. Ng. On the Row Merge Tree for Sparse LU Factorization with Partial Pivoting. *BIT Numerical Mathematics Journal*, 47(1):45–76, 2007.

[18] L. Grigori, J. Demmel, and X. Li. Parallel Symbolic Factorization for Sparse LU Factorization with Static Pivoting. *SIAM J. on Sc. Comp.*, 29(3):1289–1314, 2007.

[19] B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM J. Sci. Stat. Comput.*, 16(2):452–469, 1995.

[20] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing '95*. ACM, December 1995.

[21] B. Hendrickson and E. Rothberg. Improving the runtime and quality of nested dissection ordering. *SIAM J. Sci. Comp.*, 20(2):468–489, 1997.

[22] HSL. A collection of Fortran codes for large scale scientific computation, 2004. http://www.cse.clrc.ac.uk/nag/hsl/.

[23] Y. Hu, K. Maguire, and R. Blake. A multilevel unsymmetric matrix ordering algorithm for parallel process simulation. *Computers and Chemical Engineering*, 23:1631–1647, January 2000.

[24] G. Karypis and V. Kumar. METIS: A software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing orderings of sparse matrices - verstion 4.0, 1998. See http://www-users.cs.umn.edu/karypis/metis.

[25] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. on Scientific Computing*, 20:359–392, 1999.

[26] X. S. Li and J. W. Demmel. SuperLU_DIST: A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric linear systems. *ACM Trans. Math. Software*, 29(2), 2003.

[27] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.

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

#	Mmd $(A^T + A)$	Mmd $(A^T A)$	Colamd	Amd	Metis $(A^T + A)$	Metis $(A^T A)$	Hund			
							k=16	k=128	t=1	t=100
1	46.6	**13.5**	13.6	46.7	22.8	15.3	15.0	15.2	15.2	15.2
2	337.9	19.2	18.3	112.3	47.1	**16.8**	19.6	**16.8**	17.2	16.9
3	**1.0**	1.1	1.1	**1.0**	**1.0**	1.1	1.1	1.1	1.1	1.1
4	18.3	3.5	**3.4**	10.4	10.1	3.9	**3.4**	3.6	3.7	3.6
5	5.7	7.7	7.5	**5.6**	6.0	8.3	8.3	8.4	8.2	8.3
6	105.7	42.8	42.0	86.8	60.9	32.9	30.6	**30.0**	**30.0**	**30.0**
7	22.4	4.0	**3.9**	9.4	6.7	4.6	4.5	4.5	4.5	4.5
8	28.8	7.6	7.9	23.5	15.5	7.0	6.8	**6.7**	**6.7**	**6.7**
9	-	94.7	111.9	-	225.2	66.0	71.5	62.7	**61.9**	62.3
10	19.8	93.4	8.1	218.0	83.9	-	39.9	4.9	**3.2**	**3.2**
11	**3.9**	37.0	23.1	14.5	40.2	39.0	40.8	11.6	11.5	11.6
12	93.3	23.0	18.7	85.9	76.8	19.4	16.8	**16.5**	16.8	16.7
13	10.1	5.3	4.8	5.2	17.2	5.3	5.1	4.9	**4.7**	**4.7**
14	68.6	47.2	54.4	55.1	48.2	**32.3**	33.9	32.8	32.7	32.8
15	-	144.8	116.0	-	102.8	54.7	49.1	**46.5**	46.6	**46.5**
16	17.2	12.9	13.4	18.1	27.8	14.2	**11.1**	**11.1**	11.4	11.2
17	80.5	26.1	**24.9**	33.0	28.3	27.6	29.3	29.8	29.6	29.6
18	92.5	61.0	68.1	72.1	66.5	42.4	53.5	**52.2**	52.4	**52.2**
19	-	13.1	14.7	-	28.5	13.0	13.6	12.4	**12.3**	12.4
20	14.7	**5.5**	**5.5**	15.2	8.5	7.1	6.0	6.1	6.1	6.1
21	27.6	**4.5**	**4.5**	13.2	9.6	4.8	4.8	5.0	4.9	4.9
22	-	76.3	81.4	-	85.7	**55.9**	64.7	62.5	62.6	62.6
23	29.5	15.8	**13.1**	45.8	-	20.8	23.9	24.1	23.5	23.2
24	36.0	**4.5**	4.6	13.1	9.9	5.0	4.7	4.9	4.9	4.9
25	**8.3**	14.0	16.7	9.9	8.9	13.4	15.8	15.2	14.3	14.5
26	-	23.2	25.8	-	41.5	16.4	16.8	15.5	**15.1**	15.2
27	-	**27.8**	28.1	-	61.2	28.2	29.5	28.8	28.8	28.8

Table 3: Fill in the factors L and U, computed as $nnz(L + U - I)/nnz(A)$ obtained by different fill-reducing strategies with SuperLU

[29] F. Manne and R. Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *Proc. Seventh Int. Conf. on Parallel Processing and Applied Mathematics (PPAM 2007)*, 2007.

[30] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management*, 3:255–269, 1957.

[31] M. A. Saunders. personal communication.

[32] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[33] M. Yannakakis. Computing the minimum fill-in is np-complete. *SIAM J. Alg. Disc. Meth.*, 2(x):77–79, 1981.

Appendix

In Table 3 we present the detailed experimental data that was used to create Figures 6, 7 and 8 in section 5, using SuperLU.

Table 4 presents the results using UMFPACK, for Figure 9.

#	Colamd	Amd	Metis $(A^T + A)$	Metis $(A^T A)$	Hund			
					k=16	k=128	t=1	t=100
1	**11.5**	40.5	23.8	14.5	14.3	14.1	14.2	14.3
2	14.5	134.4	125.0	14.0	14.2	13.8	13.9	**13.8**
3	1.0	1.0	1.0	1.0	**1.0**	1.0	1.0	1.0
4	**2.5**	4.0	11.7	2.8	2.7	2.9	2.8	2.9
5	6.9	**5.7**	6.0	7.7	7.2	7.3	7.2	7.3
6	32.9	45.0	36.9	29.3	25.0	24.8	**24.8**	24.9
7	**3.0**	13.0	9.5	3.8	3.6	3.6	3.6	3.6
8	**1.0**	2.0	2.1	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
9	78.7	47.2	**28.4**	55.7	57.6	51.0	50.3	50.7
10	**1.9**	2.3	2.8	2.3	1.9	2.0	1.9	2.0
11	4.4	4.1	4.9	4.1	4.1	4.1	**4.0**	4.1
12	8.8	**7.6**	9.5	10.7	9.2	9.2	9.4	9.3
13	**3.0**	4.7	13.0	3.6	3.3	3.5	3.4	3.4
14	38.3	36.5	35.4	20.6	22.7	21.9	**19.9**	22.3
15	79.6	63.9	42.0	44.7	41.9	39.7	**39.2**	40.1
16	10.2	-	37.8	11.6	**9.5**	9.6	10.2	9.6
17	22.0	**16.9**	17.5	24.9	25.6	25.6	25.6	25.5
18	49.9	-	-	**29.9**	41.4	39.7	40.0	41.5
19	6.8	14.4	11.6	8.5	**6.2**	7.8	8.3	7.8
20	**4.6**	9.8	6.1	5.6	5.0	4.6	4.7	4.7
21	3.8	26.8	20.9	4.2	**3.8**	4.0	4.0	4.0
22	-	-	-	**38.9**	-	-	-	-
23	**5.9**	-	14.4	8.3	6.1	6.1	6.1	6.3
24	3.8	-	-	4.2	**3.8**	4.0	4.0	4.0
25	14.3	9.4	**8.9**	11.6	13.1	12.5	11.5	12.1
26	20.2	-	-	11.2	11.8	11.0	**10.6**	10.7
27	**23.7**	-	25.8	25.6	25.4	24.6	24.8	24.8

Table 4: Fill in the factors L and U, computed as $nnz(L+U-I)/nnz(A)$, using UMFPACK