NESTED-DISSECTION ORDERINGS FOR SPARSE LU WITH PARTIAL PIVOTING

IGOR BRAINMAN AND SIVAN TOLEDO

Abstract. We describe the implementation and performance of a novel fill-minimization ordering technique for sparse LU factorization with partial pivoting. The technique was proposed by Gilbert and Schreiber in 1980 but never implemented and tested. Like other techniques for ordering sparse matrices for LU with partial pivoting, our new method preorders the columns of the matrix (the row permutation is chosen by the pivoting sequence during the numerical factorization). Also like other methods, the column permutation Q that we select is a permutation that minimizes the fill in the Cholesky factor of $Q^T A^T A Q$. Unlike existing column-ordering techniques, which all rely on minimum-degree heuristics, our new method is based on a nested-dissection ordering of $A^T A$. Our algorithm, however, never computes a representation of $A^T A$, which can be expensive. We only work with a representation of A itself. Our experiments demonstrate that the method is efficient and that it can reduce fill significantly relative to the best existing methods. The method reduces the LU running time on some very large matrices (tens of millions of nonzeros in the factors) by more than a factor of 2.

1. Introduction

Reordering the columns of sparse nonsymmetric matrices can significantly reduce fill in sparse LU factorizations with partial pivoting. Reducing fill in a factorization reduces the amount of memory required to store the factors, the amount of work in the factorization, and the amount of work in subsequent triangular solves. Symmetric positive-definite matrices, which can be factored without pivoting, are normally reordered to reduce fill by applying the same permutation to both the rows and columns of the matrix. Applying the same permutation to the rows and columns preserves the symmetry of the matrix. When partial pivoting is required for maintaining numerical stability, however, prepermuting the rows is meaningless, since the rows

This research was supported by Israel Science Foundation founded by the Israel Academy of Sciences and Humanities (grant number 572/00 and grant number 9060/99) and by the University Research Fund of Tel-Aviv University.

are exchanged again during the factorization. Therefore, we normally preorder the columns and let numerical consideration dictate the row ordering. Since columns are reordered before the row permutation is known, we need to order the columns such that fill is minimized no matter how rows are exchanged. (Some nonsymmetric factorization codes that employ pivoting, such as UMFPACK/MA38 [4, 5], determine the column permutation during the numerical factorization; such codes do not preorder columns so the technique in this paper does not apply to them.)

A result by George and Ng [8] suggests one effective way to preorder the columns to reduce fill. They have shown that the fill of the LU factors of PA is essentially contained in the fill of the Cholesky factor of A^TA for every row permutation P. (P is a permutation matrix that permutes the rows of A and represents the actions of partial pivoting.) Gilbert [10] later showed that this upper bound on the fill of the LU factors is not too loose, in the sense that for a large class of matrices, for every fill element in the Cholesky factor of A^TA there is a pivoting sequence P that causes the element to fill in the LU factors of A. Thus, nonsymmetric direct sparse solvers often preorder the columns of A using a permutation Q that minimizes fill in the Cholesky factor of Q^TA^TAQ.

The main challenge in column-ordering algorithms is to find a fill-minimizing permutation without computing A^TA or even its nonzero structure. While computing the nonzero structure of A^TA allows us to use existing symmetric ordering algorithms and codes, it may be grossly inefficient. For example, when an n-by-n matrix A has nonzeros only in the first row and along the main diagonal, computing A^TA takes $\Omega(n^2)$ work, but factoring it takes only $O(n)$ work. Consider an n-by-n matrix in which all the nonzeros are in the first row and along the main diagonal, such as ($\times$'s represent nonzeros)

$$A = \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ & \times & & & & \\ & & \times & & & \\ & & & \times & & \\ & & & & \times & \\ & & & & & \times \end{bmatrix}.$$

The matrix A^TA is full, so computing its structure requires at least $\Theta(n^2)$ work. But since A^TA is full, all its orderings are equivalent in terms of fill. Thus, we perform $\Omega(n^2)$ work and get no useful information. If we factor this matrix without reordering its columns, no

pivoting takes place and no fill is produced, so the factorization requires only $\Theta(n)$ work. To summarize, computing A^TA may require significantly more memory and work than the partial-pivoting numerical factorization requires.

This challenge has been met for the class of reordering algorithms based on the minimum-degree heuristic. Modern implementations of minimum-degree heuristics use a *clique-cover* to represent the graph G_A of the matrix[1] A (see [7]). A clique cover represents the edges of the graph (the nonzeros in the matrix) as a union of cliques, or complete subgraphs. The clique-cover representation allows us to simulate the elimination process with a data structure that only shrinks and never grows. There are two ways to initialize the clique-cover representation of G_{A^TA} directly from the structure of A. Both ways create a data structure whose size is proportional to the number of nonzeros in A, not the number of nonzeros in A^TA. From then on, the data structure only shrinks, so it remains small even if A^TA is relatively dense. In other words, finding a minimum-degree column ordering for A requires about the same amount of work and memory as finding a symmetric ordering for $A^T + A$, the symmetric completion of A.

Nested-dissection ordering methods were proposed in the early 1970's and have been known since then to be theoretically superior to minimum-degree methods for important classes of sparse symmetric definite matrices. Only in the last few years, however, have nested-dissection methods been shown experimentally to be more effective than minimum-degree methods. Today's state-of-the-art methods use fast multilevel algorithms for finding separators and fuse nested-dissection and minimum-degree to reduce fill below the level that either method alone produces.

In 1980 Gilbert and Schreiber proposed a method for ordering G_{A^TA} using nested-dissection heuristics, without ever forming A^TA [9, 11]. Their method uses *wide separators*, a term that they coined. They have never implemented or tested their proposed method.

The main contribution of this paper is an implementation and an experimental evaluation of the wide-separator ordering method, along with a new presentation of the theory of wide separators. Our code dissects G_{A^TA} without forming it. The code then uses existing techniques for minimum-degree column ordering to reduce fill in LU with partial pivoting to below that of any existing technique.

Modern symmetric ordering methods generally work as follows:

[1]The graph $G_A = (V, E)$ of an n-by-n matrix A has a vertex set $V = \{1, 2, \ldots, n\}$ and an edge set $E = \{(i, j)|a_{ij} \neq 0\}$. We ignore numerical cancellations in this paper.

1. The methods find a small vertex separator that separates the graph G into two subgraphs with roughly the same size.
2. Each subgraph is dissected recursively, until each subgraph is fairly small (typically several hundred vertices).
3. The separators are used to impose a coarse ordering. The vertices in the top-level separator are ordered last, the vertices in the second-to-top level come before them, and so on. The vertices in the small subgraphs that are not dissected any further appear first in the ordering. The ordering within each separator and the ordering within each subgraph has not yet been determined.
4. A minimum-degree algorithm computes the final ordering, subject to the coarse ordering constraints.

While there are many variants, most codes use this overall framework.

Our methods apply the same framework to the graph of A^TA, but without computing it. We find separators in A^TA by finding *wide separators* in $A^T + A$. We find a wide separator by finding a conventional vertex separator and widening it by adding to it all the vertices that are adjacent to the separator in one of the subgraphs. Such a wide separator corresponds to a vertex separator in A^TA. Just like symmetric methods, our methods recursively dissect the graph, but using wide separators. When the remaining subgraphs are sufficiently small, we compute the final ordering using a constrained column-minimum-degree algorithm. We use existing techniques to produce a minimum-degree ordering of A^TA without computing G_{A^TA} (either the row-clique method or the augmented-matrix method).

The (conventional) vertex-separator code that we use is part of a library called SPOOLS [1]. Our code can use SPOOLES's minimum-degree code, as well as a version of COLAMD [12] that we modified to respect the coarse ordering.

Experimental results show that our method can reduce the work in the LU factorization by up to a factor of 3 compared to state-of-the-art column-ordering codes. The running times of our method are higher than the running-times of strict minimum-degree codes, such as COLAMD [12], but they are low enough to easily justify using the new method. On many matrices, including large ones, our method significantly reduces the work compared to all the existing column ordering methods. On some matrices, however, constraining the ordering using wide-separators increases fill rather than reduce it.

The rest of the paper is organized as follows. Section 2 presents the theory of wide separators and algorithms for finding them. Our experimental results are presented in Section 3. We discuss our conclusions from this research in Section 4.

2. Wide Separators: Theory and Algorithms

Our column-ordering methods find separators in G_{A^TA} by finding a so-called *wide separator* in G_{A^T+A}. We work with the graph of $A^T + A$ and not with G_A for two reasons. First, this simplifies the definitions and proofs. Second, to the best of our knowledge all existing vertex-separator codes work with undirected graphs, so there is no point in developing the theory for the directed graph G_A.

A vertex subset $S \subseteq V$ of an undirected graph $G = (V, E)$ is a *separator* if the removal of S and its incident edges breaks the graph into two components $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, such that any path between $i \in V_1$ and $j \in V_2$ passes through at least one vertex in S. A vertex set is a *wide separator* if every path between $i \in V_1$ and $j \in V_2$ passes through a sequence of two vertices in S (one after the other along the path).

Our first task is to show that every wide separator in G_{A^T+A} is a separator in G_{A^TA}.

Theorem 2.1. *A wide separator in G_{A^T+A} is a separator in G_{A^TA}.*

Proof. Let S be a wide separator in G_{A^T+A}. Suppose for contradiction that S is not a separator in G_{A^TA}: there exists a path in G_{A^TA} between $i \in V_1$ and $j \in V_2$ that does not pass though a vertex s in S. There must be a pair of vertices i' and j' along the path such that $i' \in V_1$ and $j' \in V_2$. Thus, i' and j' are neighbors in G_{A^TA}, so the (i', j') element in A^TA is nonzero. Since $(A^TA)_{i',j'} = \sum_k A_{k,i'}A_{k,j'} \neq 0$, there must be some k such that $A_{k,i'} \neq 0$ and $A_{k,j'} \neq 0$. Hence, there is a path $i' \leftrightarrow k \leftrightarrow j'$ in G_{A^T+A} between i' and j' that passes through only vertex in S, a contradiction. □

The converse is not always true. There are matrices with separators in G_{A^TA} that do not correspond to wide separators in $A^T + A$. Consider

$$A = \begin{bmatrix} & \times \\ \times & \end{bmatrix}$$

(the $\times$'s represent nonzeros). The empty set is a separator in the graph of

$$A^TA = \begin{bmatrix} & \times \\ \times & \end{bmatrix} \begin{bmatrix} & \times \\ \times & \end{bmatrix} = \begin{bmatrix} \times & \\ & \times \end{bmatrix}$$

but it is not a wide separator in the graph of $A^T + A$ (it is not even a separator). The converse of the Theorem 2.1 is true, however, when there are no zeros on the main diagonal of A:

Theorem 2.2. *If there are no zeros on the diagonal of A, then a separator in G_{A^TA} is a wide separator in G_{A^T+A}.*

Proof. Let S be a separator in G_{A^TA}. Suppose for contradiction that S is not a wide separator in G_{A^T+A}. There exists a path in G_{A^T+A} between some $i \in V_1$ and $j \in V_2$ that does not pass though a sequence of two vertices in S. This can happen in two ways. Either there are some $i' \in V_1$ and $j' \in V_2$ that are adjacent in G_{A^T+A}(that is, S is not a separator at all in G_{A^T+A}), or there are some $i' \in V_1$ and $j' \in V_2$ that are separated in G_{A^T+A} but not widely: there is a path $i' \leftrightarrow s \leftrightarrow j'$ in G_{A^T+A}. In both cases the edge (i', j') will be in G_{A^TA} (in the former case due to the paths $i' \leftrightarrow i' \leftrightarrow j'$ and $i' \leftrightarrow j' \leftrightarrow j'$), a contradiction. □

Given a code that finds conventional separators in an undirected graph, finding wide separators is easy. The separator and its neighbors in either G_1 or G_2 form a wide separator:

Lemma 2.3. *Let S be a separator in an undirected graph G. The sets $S_1 = S \cup \{i | i \in V_1, (i, j) \in E \text{ for some } j \in S\}$ and $S_2 = S \cup \{i | i \in V_2, (i, j) \in E \text{ for some } j \in S\}$ are wide separators in G.*

The proof is trivial. The sizes of S_1 and S_2 are bounded by $d|S|$, where d is the maximum degree of vertices in S. Given S, it is easy to enumerate S_1 and S_2 in time $O(d|S|)$. This running time is typically insignificant compared to the time it takes to find S.

Which one of the two candidate wide separators should we choose? A wide separator that is small and that dissects the graph evenly reduces fill in the Cholesky factor of A^TA, and hence in the LU factors of A. The two criteria are usually contradictory. Over the years it has been determined the the best strategy is to choose a separator that is as small as possible, as long as the ratio of the number of vertices in G_1 and G_2 does not exceed 2 or so.

The following method is, therefore, a reasonable way to find a wide separator: Select the smallest of S_1 and S_2, unless the smaller wide separator unbalances the separated subgraphs (so that one is more than twice as large as the other) but the larger does not. Our code, however, is currently more naive and always choose the smaller wide separator.

3. Experimental Results

This section summarizes our experimental results. We begin by describing our code, our collection of test matrices, and the computer that was used to carry out the experiments. We then describe and

analyze the results of our experiments. The analyses focus on the effectiveness of various ordering methods and on their performance. By effectiveness we mean the number of nonzeros in the factors, the number of floating-point operations (flops) required to compute them, and the factorization time. By performance we mean the cost, mostly in terms of time, of the ordering algorithm itself.

3.1. **Experimental Setup.** The experiments that this section describe test the effectiveness and performance of several column-ordering codes. We have tested our new codes, which implement nested-dissection-based orderings, as well as two existing ordering codes.

Our codes build a hierarchy of wide separators and use the separators to constrain a minimum-degree algorithm. We obtain the wide separators by widening separators in G_{A^T+A} that SPOOLES [1] finds. SPOOLES is a library of sparse ordering and factorization codes written by Cleve Ashcraft and others. Our codes then invoke a column-minimum-degree code to produce the final ordering. One minimum-degree code that we use is SPOOLES's multi-stage-minimum-degree (MSMD) algorithm, which we run on the augmented matrix

$$\widetilde{A} = \begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} .$$

We constrain the minimum-degree code to eliminate the first n rows/columns first. This elimination constructs a clique-cover representation of G_{A^TA} on the remaining uneliminated vertices, which MSMD eliminate next under the wide-separator constraints.

The other minimum-degree code that we used is a version of COLAMD [12] that we modified to respect the constraints imposed by the separators.

The existing minimum-degree codes that we have tested include COLAMD, SPOOLES's MSMD (operating on the augmented matrix with no separator constraints). In an earlier experiment, reported in [2], we also tested COLMMD, a column minimum-degree code, originally written by Joseph W.-H. Liu and distributed with SuperLU. It was not shown to be consistently superior to the other two codes so we dropped it from the experiment reported here.

We use the following acronyms to refer to the ordering methods: MSMD refers to SPOOLES's minimum-degree code operating on the augmented matrix without constraints, WSMSMD refers to the same minimum-degree code but constrained to respect wide separators, and similarly for COLAMD and WSCOLAMD.

In the experiments reported here, we always reduce the input matrices to *block triangular form* (see [14]) and factor only the diagonal blocks in the reduced form. Many of the matrices in our test suite have numerous tiny diagonal blocks (most of them 1-by-1); we report the performance of factoring all the diagonal blocks with dimension at least 250.

We factor the reordered matrix using SuperLU [6, 13] version 2.0, a state-of-the-art sparse partial-pivoting LU code. SuperLU uses the Basic Linear Algebra Subroutines (BLAS). we used ATLAS[2], a high-performance implementation of the BLAS.

We conducted the experiments on a 600MHz dual Pentium III computer with 2 GBytes of main memory running Linux. The machine was configured without swap space so no paging occurred during the experiments. This machine has two processors, but our code only uses one processor. The compiler that we used is GCC version egcs-2.91.66. We used the recommended optimization level for each package: `-O` for SPOOLES and `-O3` for SuperLU.

3.2. **Matrices.** We tested the ordering methods on a set of nonsymmetric sparse matrices from Tim Davis's sparse matrix collection[3]. We used all the nonsymmetric matrices in Davis's collection that are not too small (factorization time with the best ordering method at least 1/1000 of a second). Two of the matrices in Davis's collection were too large to factor on our machine (`appu` and `pre2`) and SPOOLES broke down on two more (`av41092` and `twotone`; we are unsure whether the breakdown is due to a bug in our code or due to a problem in SPOOLES).

The matrices are listed in Tables 1 and 2. We split the matrices into small ones and large ones based on the number of flops (floating-point operations) in the factorization. We refer to matrices whose factorization with the best ordering requires more than 100 Mflops (millions of flops) as *large*, the rest are referred to as small. We always sort matrices by this factorization-flops metric. The tables show the matrix's name, dimension (N), number of nonzeros (NNZ), number of blocks in the block triangular form, number of big blocks (dimension at least 250) in the block triangular form, and the best flop count in millions.

We also run experiments on matrices whose graphs are regular 2- and 3-dimensional meshes and whose values are random numbers in the range $[0, 1]$.

[2] `www.netlib.org/atlas`

[3] `www.cise.ufl.edu/~davis/sparse/`

TABLE 1. General information for the small matrices.

NO	NAME	N	NNZ	BTF BLOCKS	BIG BTF BLOCKS	BEST MFLOPS
1	raefsky6	3402	137845	3402	0	0
2	raefsky5	6316	168658	6316	0	0
3	poli_large	15575	33074	15466	0	0
4	bwm2000	2000	7996	1	1	0
5	epb0	1794	7764	1	1	0
6	cavity04	317	7327	82	0	0
7	lhr01	1477	18592	298	1	2
8	rdist2	3198	56934	199	1	4
9	bayer02	13935	63679	2151	1	7
10	bayer10	13436	94926	1541	1	8
11	rdist3a	2398	61896	99	1	8
12	rdist1	4134	94408	199	1	8
13	orani678	2529	90158	700	1	17
14	lhr04c	4101	82682	439	1	17
15	lhr04	4101	82682	439	1	18
16	bayer04	20545	159082	6378	1	18
17	ex9	3363	99471	1	1	23
18	lhr07c	7337	156508	672	2	27
19	lhr07	7337	156508	672	2	27
20	ex31	3909	115357	1	1	28
21	rw5151	5151	20199	6	1	30
22	bayer01	57735	277774	8861	1	30
23	ex28	2603	77781	1	1	35
24	lhr11	10964	233741	1192	3	35
25	lhr11c	10964	233741	1192	3	35
26	lhr10c	10672	232633	908	3	36
27	lhr10	10672	232633	908	3	36
28	ex19	12005	259879	305	3	36
29	memplus	17758	126150	1	1	39
30	lhr14	14270	307858	1556	5	42
31	lhr14c	14270	307858	1556	5	42
32	lhr17	17576	381975	1798	6	55
33	lhr17c	17576	381975	1798	6	56
34	ex8	3096	106344	1	1	68
35	ex35	19716	228208	173	4	73
36	cavity26	4562	138187	322	1	91
37	cavity24	4562	138187	322	1	91
38	onetone2	36057	227628	3843	1	92
39	cavity25	4562	138187	322	1	92
40	cavity23	4562	138187	322	1	92
41	cavity22	4562	138187	322	1	92
42	cavity21	4562	138187	322	1	94
43	cavity20	4562	138187	322	1	94
44	cavity19	4562	138187	322	1	95
45	cavity18	4562	138187	322	1	98

TABLE 2. General information for the large matrices.

NO	NAME	N	NNZ	# OF BLOCKS	# OF BIG BLOCKS	BEST MFLOPS
46	cavity17	4562	138187	322	1	101
47	epb1	14734	95053	1	1	123
48	utm5940	5940	83842	147	1	126
49	lhr34	35152	764014	3533	10	128
50	lhr71	70304	1528092	7066	20	259
51	lhr71c	70304	1528092	7066	20	262
52	shyy161	76480	329762	25761	1	479
53	epb2	25228	175027	1	1	517
54	goodwin	7320	324784	2	1	524
55	epb3	84617	463625	1	1	809
56	raefsky2	3242	294276	1	1	921
57	raefsky1	3242	294276	1	1	921
58	graham1	9035	335504	478	1	989
59	garon2	13535	390607	1	1	1061
60	ex40	7740	458012	1	1	1075
61	rim	22560	1014951	2	1	1877
62	onetone1	36057	341088	3843	1	2371
63	olafu	16146	1015156	1	1	2584
64	venkat01	62424	1717792	1	1	4299
65	venkat50	62424	1717792	1	1	4299
66	venkat25	62424	1717792	1	1	4299
67	rma10	46835	2374001	1	1	4386
68	af23560	23560	484256	1	1	4515
69	raefsky3	21200	1488768	1	1	5243
70	raefsky4	19779	1328611	1	1	7800
71	ex11	16614	1096948	1	1	11194
72	psmigr_2	3140	540022	1	1	13412
73	psmigr_3	3140	543162	1	1	14649
74	psmigr_1	3140	543162	1	1	14776
75	wang3	26064	177168	1	1	15515
76	wang4	26068	177196	1	1	24484
77	bbmat	38744	1771722	1	1	44553
78	li	22695	1350309	2	2	84241

3.3. **Results and Analysis.** Table 3 and Figures 3.1, 3.2, and 3.3 summarize the results of our experiments. These results supersede the preliminary results that we reported in [2, 3].

Table 3 shows that wide-separator (WS) orderings are both effective and efficient. On the largest 2D and 3D meshes, WS orderings lead to the fastest factorization times and to the fastest overall solution time (including ordering time). Beyond performance, WS orderings enable us to solve problem that we could simply not solve with minimum-degree orderings with this amount of main memory (2GB).

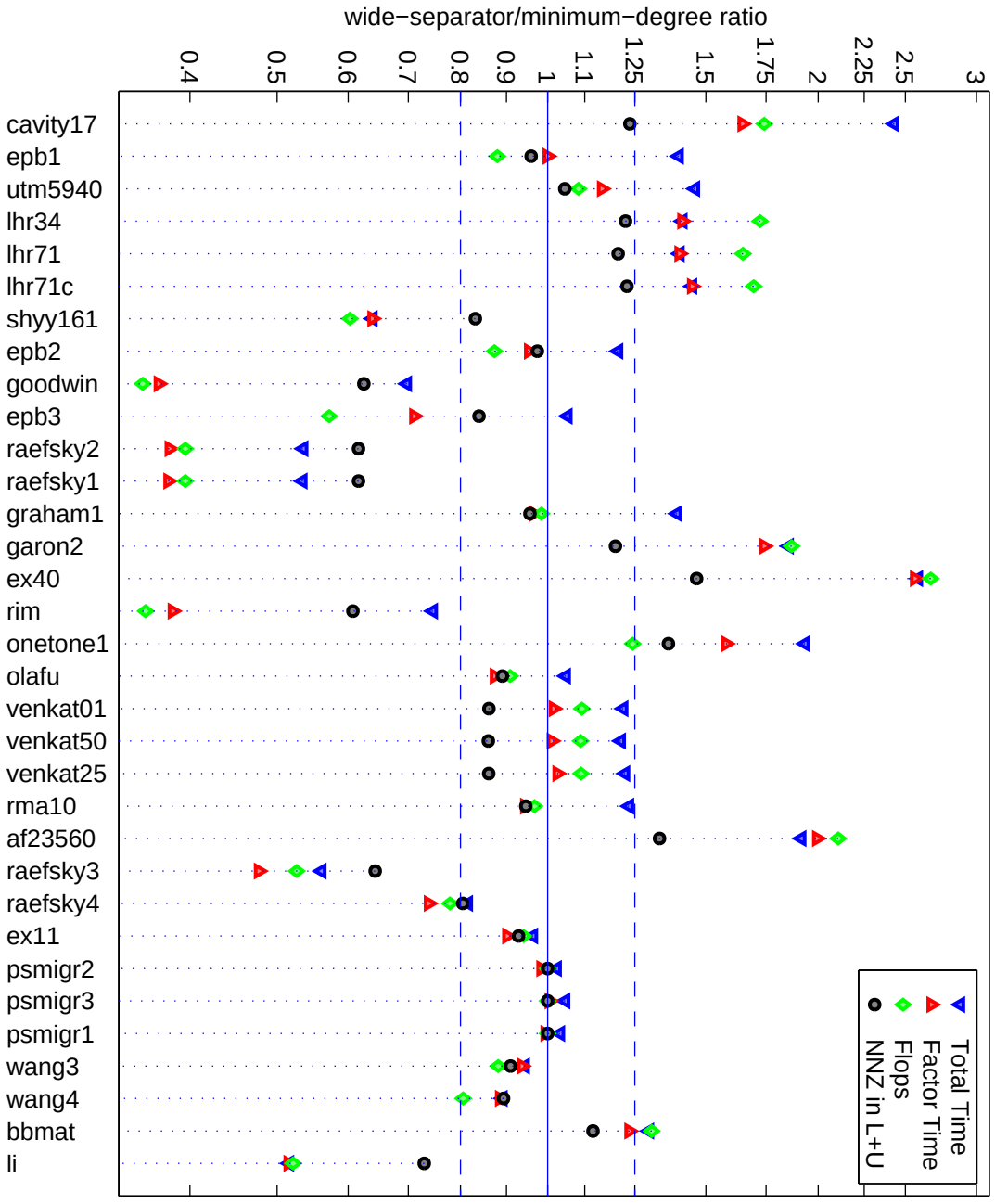

FIGURE 3.1. The ratios of WSCOLAMD's performance to that of COLAMD. Performance is reported in terms of flops, number of nonzeros in L and U, factorization times, and total times (including ordering). Data points below 1 indicate that WSCOLAMD is better than COLAMD. Matrices are sorted by best factorization flops. The y-axis is logarithmic.

TABLE 3. A comparison of wide-separator and minimum-degree orderings on regular 2- and 3-dimensional meshes. All the matrix entries are random. The first three columns show the dimensions of the meshes, the next two the best factorization time and the ordering method that lead to the best time. The last four columns show the combined ordering and factorization times in seconds.

			Best	Best	Times In Seconds			
N_x	N_y	N_z	Time	Method	WSCOLAMD	COLAMD	WSCOLMSMD	COLMSMD
500	500		113	WSCOLAMD	150	202	150	—
750	750		496	WSCOLAMD	601	—	684	—
30	30	30	352	COLAMD	399	352	1210	404
40	40	40	786	WSCOLAMD	792	2340	958	—

Wide-separator orderings are not effective on small matrices. Of the 45 small matrices in our test suite, WS orderings reduce flop counts significantly (by more than 25%) over COLAMD on only 2 matrices (`ex8` and `ex9`). We note, however, that even though wide separators do not reduce work in the factorization small matrices, they rarely increase work by a factor of 2 or more. Since wide-separator orderings do not appear to be effective on small matrices, the rest of this section refers only to large matrices.

Figures 3.1, 3.2, and 3.3 summarize the results with large matrices from a test-matrix collection. A comparison of the best WS method to the best non-WS method, shown in Figure 3.3, shows that WS orderings are effective. Ws and non-WS orderings produced similar flop counts (within 25%) on 14 of the 33 matrices. Ws orderings reduced flop counts by more than 25% on 12 matrices including 5 of the 10 largest. On the other hand, ws orderings increased flop counts on only 7 matrices, none of them in the top 10. The results with COLMSMD and WSCOLMSMD, shown in Figure 3.2, are even better: the overall numbers are the same, but WS orderings reduce work significantly on 7 matrices in the top 10. The results with COLAMD and WSCOLAMD are a bit less favorable to WS orderings: they reduce work on 9 matrices but increase work on 8 (discounting relative differences of less than 25%).

Nonzero counts in the LU factors and factorization times are generally correlated with flop counts; smaller flop counts usually imply fewer nonzeros in the LU factors and shorter factorization times.

The improvement due to wide separators is often large. On the largest matrix in our test suite, `li`, wide separators reduce flop counts

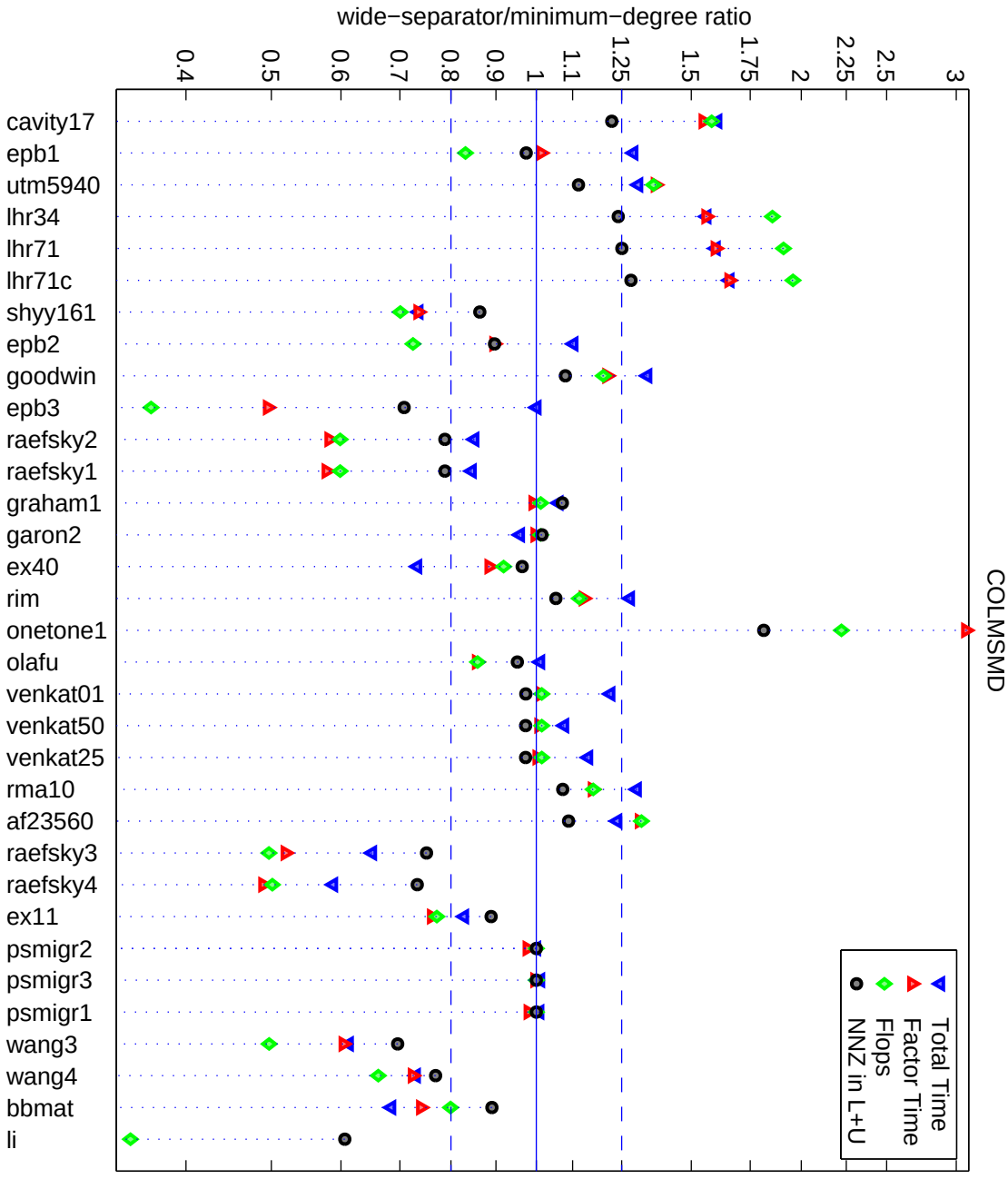

FIGURE 3.2. WSCOLMSMD to COLMSMD ratios.

and factorization time by about a factor of 2. The reduction in terms of flop counts compared to non-WS methods is also highly significant on `wang3`, `raefsky1/2/3`, `rim`, and especially `epb3`.

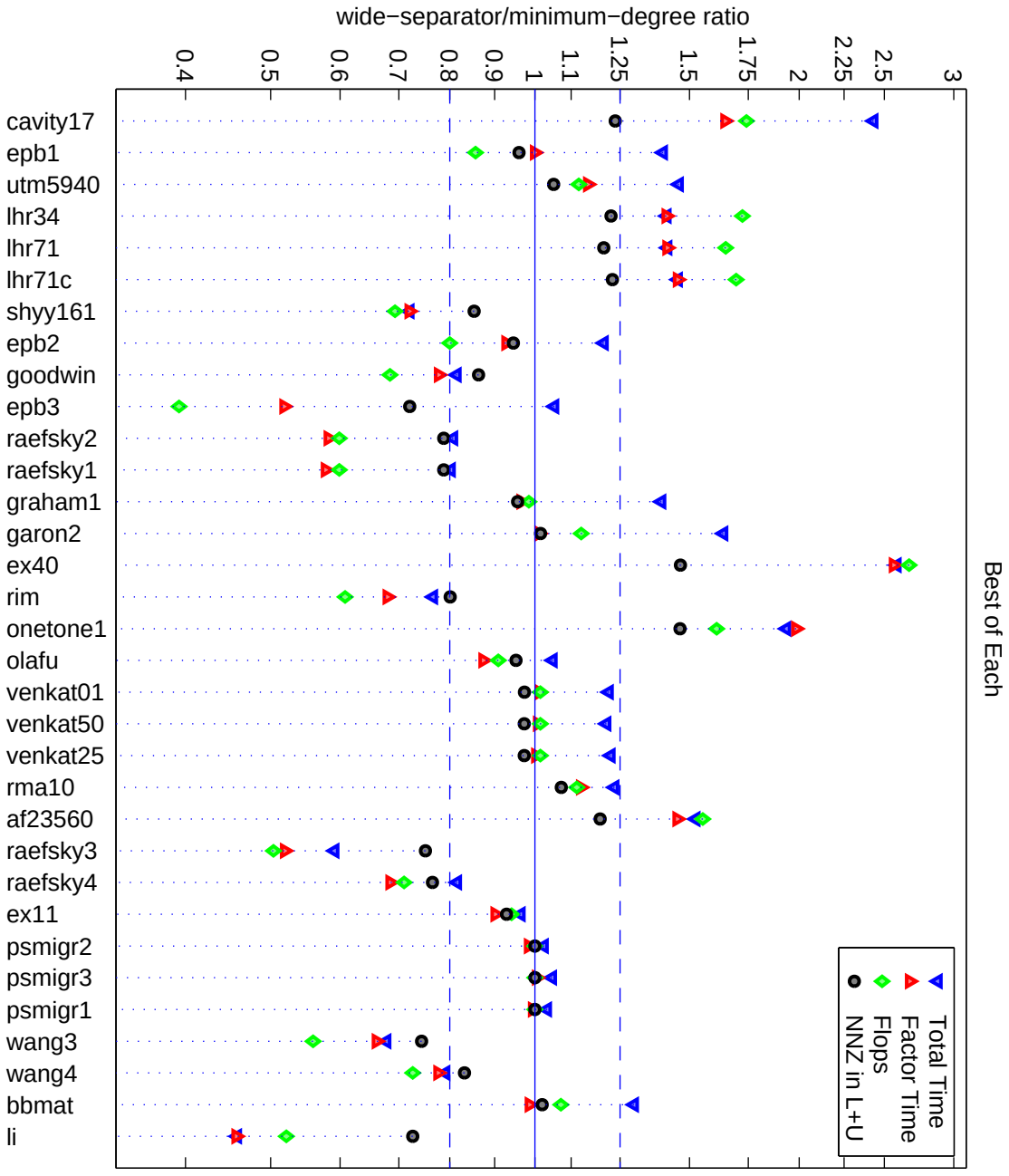

FIGURE 3.3. Best WS to best non-WS methods.

When WS orderings do poorly compared to non-WS methods, however, they sometimes do significantly poorer. On `ex40`, for example, using wide separators slows down the factorization by a factor of about

2.5. In earlier experiments [2], in which we did not reduce the matrices to block triangular form, we have found that on some of matrices, especially the `lhr` and `bayer` matrices, the slowdowns are even more dramatic. The experiments reported here show that reduction to block triangular form resolves these problems.

Wide-separator orderings are somewhat more expensive to compute than strict minimum-degree orderings. Figure 3.3 shows that when the ordering times are taken into account, wide-separator orderings speed up the total solution time by more than 25% in only 6 out of the 33 matrices (but including 4 in the top 10). But there are no cases where WS orderings significantly reduce the factorization time but significantly increase the total times. Hence, ordering is a significant cost in WS-based factorization, but not a dominant one. We also note that even when a wide-separator ordering reduces the factorization time but not total time, it typically also reduces the size of the factors, which is often highly important (since it saves memory, reduces the occurrence of paging, and speeds up subsequent triangular solves).

4. Conclusions And Discussion

Our main conclusion from this research is that hybrid wide-separator minimum-degree column orderings are effective. WS orderings are clearly superior to minimum-degree orderings alone on large 2D and 3D meshes that require pivoting. On matrices obtained from a matrix collection, WS orderings often substantially reduce the amount of time and storage required to factor a sparse matrix with partial pivoting, compared to column-minimum-degree orderings. They are more expensive to compute than minimum-degree orderings but the expense is often more than paid off by reductions in time and storage during the factorization stage.

Wide-separator orderings, like other column orderings based on fill in the factors of A^TA, are robust but pessimistic. They are robust in the sense that they reduce worst-case fill. Optimistic column orderings that attempt to reduce the fill in the factors of $A^T + A$ tend to reduce fill better than pessimistic orderings when little or no pivoting occurs, but can lead to catastrophic fill when pivoting does occur. Further discussion of pessimistic versus optimistic orderings is beyond the scope of this paper.

The combined results of this paper and of an earlier paper [2] show that first permuting the matrix to block triangular form reduces the wide-separator ordering times and improves the quality of the ordering on some matrices.

This work can be extended in several directions. First, improving the performance of the ordering phase itself would be significant. This can be done by tuning the parameters of the ordering code (stopping the recursive bisection on fairly large subgraphs) or by improving the wide-separator algorithm itself. Second, one can try to improve the orderings by trying to derive smaller wide-separators from a given conventional separator. Third, one can interleave the ordering and factorization in a way that widens separators only when necessary. That is, we would find a conventional separator S in G, recursively order G_1 and factor the columns corresponding to G_1. Once this phase is completed, we can widen the separator by adding to S the neighbors of vertices that were used as pivots. We now recursively order and factor the (shrunken) G_2.

Acknowledgement. Thanks to John Gilbert for telling us about wide-separator orderings. Thanks to John Gilbert and Bruce Hendrickson for helpful comments on an early draft of the paper. Thanks to Cleve Ashcraft for his encouragement, for numerous discussions concerning this research, and for his prompt response to our questions concerning SPOOLES.

References

[1] Cleve Ashcraft and Roger Grimes. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San-Antonio, Texas, 1999. 10 pages on CD-ROM.

[2] Igor Brainman and Sivan Toledo. Nested-dissection orderings for sparse LU with partial pivoting. In *Proceedings of the 2nd Conference on Numerical Analysis and Applications*, Rousse, Bulgaria, June 2000. 8 pages, to appear in a Springer LNCS volume.

[3] Igor Brainman and Sivan Toledo. Nested-dissection orderings for sparse LU with partial pivoting. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, Virginia, March 2001. 10 pages on CDROM.

[4] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 19:140–158, 1997.

[5] T. A. Davis and I. S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Transactions on Mathematical Software*, 25:1–19, 1999.

[6] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20:720–755, 1999.

[7] A. George and J. W. H. Liu. The evolution of the minimum-degree ordering algorithm. *SIAM Review*, 31:1–19, 1989.

[8] Alan George and Esmond Ng. On the complexity of sparse QR and LU factorization on finite-element matrices. *SIAM Journal on Scientific and Statistical Computation*, 9:849–861, 1988.

[9] John R. Gilbert. *Graph Separator Theorems and Sparse Gaussian Elimination*. PhD thesis, Stanford University, 1980.

[10] John R. Gilbert. Predicting structure in sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, 15:62–79, 1994.

[11] John R. Gilbert and Robert Schreiber. Nested dissection with partial pivoting. In *Sparse Matrix Symposium 1982: Program and Abstracts*, page 61, Fairfield Glade, Tennessee, October 1982.

[12] S. I. Larimore. An approximate minimum degree column ordering algorithm. Master's thesis, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, Florida, 1998. Also available as CISE Tech Report TR-98-016 at ftp://ftp.cise.ufl.edu/cis/tech-reports/tr98/tr98-016.ps.

[13] Xiaoye S. Li. *Sparse Gaussian Elimination on High Performance Computers*. PhD thesis, Department of Computer Science, UC Berkeley, 1996.

[14] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16(4):303–324, December 1990.

School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel

E-mail address: stoledo@tau.ac.il

URL: http://www.tau.ac.il/~stoledo