Ordering Sparse Matrices and Transforming Front Trees

Cleve Ashcraft, Boeing Shared Services Group*

January 25, 1999

1 Introduction

If the ultimate goal is to solve linear systems of the form $AX = B$, one must compute an $A = LDU$, $A = U^T DU$ or $A = U^H DU$ factorization, depending on whether the matrix A is nonsymmetric, symmetric or Hermitian. D is a diagonal or block diagonal matrix, L is unit lower triangular, and U is unit upper triangular. A is sparse, but the sparsity structure of L and U will likely be much larger than that of A, i.e., they will suffer fill-in. It is crucial to find a permutation matrix such that the factors of PAP^T have as moderate fill-in as can be reasonably expected.

To illustrate, consider a 27-point finite difference operator defined on an $n \times n \times n$ grid. The number of rows and columns in A is n^3, as is the number of nonzero entries in A. Using the natural ordering, the numbers of entries in L and U are $O(n^5)$, and it takes $O(n^7)$ operations to compute the factorization. The banded and profile orderings [11] have the same complexity.

Using the nested dissection ordering, [10], the factor storage is reduced to $O(n^4)$ and factor operations to $O(n^6)$. In practice, the minimum degree ordering has this same low-fill nature, although topological counterexamples exist [7]. A unit cube is the worst case comparison between banded and profile orderings and the minimum degree and nested dissection orderings. But, there is still a lot to be gained by using a good permutation when solving most sparse linear systems, and the relative gain becomes larger as the problem size increases.

This short paper is a gentle introduction to the ordering methods — the background as well as the specific function calls. But finding a good ordering is not enough. The "choreography" of the factorization and solves, i.e., what data structures and computations exist, and in a parallel environment, which thread or processor does what and when, is as crucial. The structure of the factor matrices, as well as the structure of the computations is controlled by a "front tree". This object is constructed directly by the **SPOOLES** ordering software, or can be created from the graph of the matrix and an outside permutation. Various transformations on the front tree can make a large difference in performance. Some knowledge of the linear system, (e.g., does it come from a 2-D or 3-D problem? is it small or large?), coupled with some knowledge of how to tailor a front tree, can be important to getting the best performance from the library.

Section 2 introduces some background on sparse matrix orderings and describes the **SPOOLES** ordering software. Section 3 presents the front tree object that controls the factorization, and its various transformations to improve performance.

*P. O. Box 24346, Mail Stop 7L-21, Seattle, Washington 98124. This research was supported in part by the DARPA Contract DABT63-95-C-0122 and the DoD High Performance Computing Modernization Program Common HPC Software Support Initiative.

2 Sparse matrix orderings

The past few years have seen a resurgence of interest and accompanying improvement in algorithms and software to order sparse matrices. The minimum degree algorithm, specifically the multiple external minimum degree algorithm [19], was the preferred algorithm of choice for the better part of a decade. Alternative minimum priority codes have recently pushed multiple minimum degree aside, including approximate minimum degree [1] and approximate deficiency [21], [25]. They offer improved quality or improved run time, and on occasion, both.

Nested dissection for regular grids [10] is within a factor of optimal with respect to factor entries and operation counts. One of the earliest attempts, automatic nested dissection [11] used a simple profile algorithm to find separators. It rarely performed as well as minimum degree. Better heuristics to find separators were brought in from the electrical device simulation area [18] and while these algorithms produced better orderings, the run times kept them from practical application. Nested dissection came on its own with two developments. The first was the application of spectral analysis of graphs to find separators [22]. The eigenvector associated with the smallest nonzero eigenvalue of the Laplacian of a graph generates a spectrum of separators. While the ordering times for spectral nested dissection were still on the order of ten or more times the cost of a minimum degree ordering, the ordering quality sometimes made the cost worthwhile.

The key that made nested dissection a competitive and practical alternative to minimum degree was the introduction of multilevel ideas — to find a separator on a graph, first find a separator on a coarse graph and project back to the original. Early implementations include [6] and [8]. Multilevel algorithms are very popular in current software including **CHACO** [14], [13], **METIS** [16], [17], **BEND** [15], **WGGP** [12] and **PCO** [23].

SPOOLES also includes a hybrid ordering approach called multi-section [3], [5], [4] and [24]. For some types of graphs, nested dissection does much better than minimum degree, for others much worse. Multisection is an ordering that uses both nested dissection and minimum degree to create an ordering that is almost always as good or better than the better of nested dissection or minimum degree and rarely much worse.

2.1 The `Graph` object

Our goal is to find a permutation matrix P such that the factorization of PAP^T has low-fill. This is a symbolic step, we do not need to know the numerical entries in A, but we do need to know the structure of A. More specifically, since when computing LDU, U^TDU or U^HDU factorizations we consider A to have symmetric structure. We need to know the structure of $A + A^T$, and so we need to construct the graph of $A + A^T$. The way that **SPOOLES** deals with the graph of $A + A^T$ is via a `Graph` object. There are several ways to construct a `Graph` object, some are high level, some are low level.

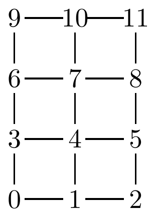
Inside each `Graph` object is an `IVL` object. `IVL` stands for `Integer Vector List`, and stores the adjacency lists for the vertices. For example, consider a 3×4 grid with a nine point operator. The adjacency lists for this graph are stored in the `IVL` object, displayed in Figure 1. (The listing comes from the `IVL_writeForHumanEye()` method.)

One can construct the `IVL` object directly. There are methods to set the number of lists, to set the size of a list, to copy entries in a list into the object. It resizes itself as necessary. However, if one already has the matrix entries of A stored in an `InpMtx` object (which is the way that **SPOOLES** deals with sparse matrices), there is an easier way. One can create an `IVL` object from the `InpMtx` object, as follows.

```
InpMtx   *A ;
IVL      *adjIVL ;

adjIVL = InpMtx_fullAdjacency(A) ;
```

Figure 1: A 3×4 9-point grid with its adjacency structure




```
IVL : integer vector list object :
type 1, chunked storage
12 lists, 12 maximum lists, 70 tsize, 4240 total bytes
1 chunks, 70 active entries, 1024 allocated, 6.84 % used
   0 : 0 1 3 4
   1 : 0 1 2 3 4 5
   2 : 1 2 4 5
   3 : 0 1 3 4 6 7
   4 : 0 1 2 3 4 5 6 7 8
   5 : 1 2 4 5 7 8
   6 : 3 4 6 7 9 10
   7 : 3 4 5 6 7 8 9 10 11
   8 : 4 5 7 8 10 11
   9 : 6 7 9 10
  10 : 6 7 8 9 10 11
  11 : 7 8 10 11
```

During a block shifted Lanczos eigenanalysis, one needs the graph of $A + \sigma B$ for a pair of matrices. There is a method to construct the `IVL` object for this case.

```
InpMtx   *A, *B ;
IVL      *adjIVL ;

adjIVL = InpMtx_fullAdjacency2(A, B) ;
```

Recall, we actually construct the adjacency structure of $A + A^T$ (or $A + A^T + B + B^T$), because the graph object is undirected, and so needs a symmetric structure.

Once we have an `IVL` object, we can construct a `Graph` object as follows.

```
Graph   *graph ;
IVL     *adjIVL ;
int     nedges, neqns ;

nedges = IVL_tsize(adjIVL) ;
graph = Graph_new() ;
Graph_init2(graph, 0, neqns, 0, nedges, neqns, nedges, adjIVL, NULL, NULL) ;
```

This is an initializer for the `Graph` object, one that takes as input a complete `IVL` adjacency object. The `0` and `NULL` fields are not applicable here. (The `Graph` object is sophisticated — it can have weighted or unweighted vertices, weighted or unweighted edges, or both, and it can have boundary vertices. Neither is relevant now.)

2.2 Constructing an ordering

Once we have a `Graph` object, we can construct an ordering. There are four choices:

- minimum degree, (actually multiple external minimum degree, from [19]),

- generalized nested dissection,
- multisection, and
- the better of generalized nested dissection and multisection.

Minimum degree takes the least amount of CPU time. Generalized nested dissection and multisection both require the a partition of the graph, which can be much more expensive to compute than a minimum degree ordering. By and large, for larger graphs nested dissection generates better orderings than minimum degree, and the difference in quality increases as the graph size increases. Multisection is an ordering which almost all the time does about as well as the better of nested dissection and minimum degree. The user should know their problem and choose the ordering. Here are some rules of thumb.

- If the matrix size is small to moderate in size, say up to 10,000 rows and columns, use minimum degree. The extra ordering time for nested dissection or multisection may not make up for any decrease in factor or solve time.
- If the matrix size comes from a partial differential equation that has several degrees of freedom at a grid point, use multisection or nested dissection, no matter the size.
- If the target is a parallel factorization, use nested dissection.
- For 2-D problems, minimum degree is usually good enough, for 3-D problems, use nested dissection or multisection.
- To be safe, use the better of the nested dissection and multisection methods. The additional ordering time is not much more than using either of them alone.

Before we discuss the different ordering methods found in `SPOOLES`, what is the output of the ordering?

One normally thinks of a permutation matrix P as represented by a permutation vector, a map from old vertices to new vertices, or vice versa. That is sufficient if one is just concerned with permuting a matrix, but there is much more information needed for the factor and solves. The `SPOOLES` ordering methods construct and return an `ETree` object that holds the *front tree*. We will discuss this object in the next section. Let us now look at the four different wrapper methods for the orderings.

```
ETree   *etree ;
Graph   *graph ;
int     maxdomainsize, maxsize, maxzeros, msglvl, seed ;
FILE    *msgFile ;

etree = orderViaMMD(graph, seed, msglvl, msgFile) ;
etree = orderViaND(graph, maxdomainsize, seed, msglvl, msgFile) ;
etree = orderViaMS(graph, maxdomainsize, seed, msglvl, msgFile) ;
etree = orderViaBestOfNDandMS(graph, maxdomainsize, maxzeros,
                              maxsize, seed, msglvl, msgFile) ;
```

Now let us describe the different parameters.

- The `msglvl` and `msgFile` parameters are used to control output. When `msglvl = 0`, there is no output. When `msglvl > 0`, output goes to the `msgFile` file. The **SPOOLES** library is a research code, we have left a great deal of monitoring and debug code in the software. Large values of `msglvl` may result in large message files. To see the statistics generated during the ordering, use `msglvl = 1`.

- The `seed` parameter is used as a random number seed. (There are many places in the graph partitioning and minimum degree algorithms where randomness plays a part. Using a random number seed ensures repeatability.)
- `maxdomainsize` is used for the nested dissection and multisection orderings. This parameter is used during the graph partition. Any subgraph that is larger than `maxdomainsize` is split. We recommend using a value of `neqns/16` or `neqns/32`. Note: `maxdomainsize` must be greater than zero.
- `maxzeros` and `maxsize` are used to transform the front tree. In effect, we have placed the ordering functionality as well as the transformation of the front tree into this method. So let's wait until the next section to learn about the `maxzeros` and `maxsize` parameters.

There is also the capability to create a front tree from a graph using any permutation vectors, e.g., a permutation that came from another graph partitioning or ordering library.

```
ETree   *etree ;
Graph   *graph ;
int     newToOld[], oldToNew[] ;

etree = ETree_new() ;
ETree_initFromGraphWithPerms(etree, graph, newToOld, oldToNew) ;
```

The output from this method is a *vertex* elimination tree, there has been no merging of rows and columns into fronts. But we are getting ahead of ourselves.

2.3 Results

Let us look at the four different orderings and compare the ordering time, the number of factor entries, and number of operations in the factorization. (The latter two are for a real, symmetric matrix.)

The `R2D10000` matrix is a randomly triangulated grid on the unit square. There are 10000 grid points, 100 points are equally spaced along each boundary. The remainder of the vertices are placed in the interior using quasi-random points, and the Delauney triangulation is computed.

	minimum degree			nested dissection		
seed	CPU	# entries	# ops	CPU	# entries	# ops
10101	1.4	212811	11600517	4.8	213235	11092015
10102	1.5	211928	11654848	4.6	216555	11665187
10103	1.5	222119	13492499	4.9	217141	11606103
10104	1.5	214151	11849181	5.0	217486	11896366
10105	1.5	216176	12063326	4.8	216404	11638612
	multisection			better of ND and MS		
seed	CPU	# entries	# ops	CPU	# entries	# ops
10101	4.6	207927	10407553	6.2	208724	10612824
10102	4.6	210364	10651916	6.2	211089	10722231
10103	4.6	215795	11760095	6.4	217141	11606103
10104	4.6	210989	10842091	6.1	212828	11168728
10105	4.8	209201	10335761	6.1	210468	10582750

For the nested dissection and multisection orderings, we used `maxdomainsize = 100`. We see that there is really little difference in ordering quality, while the minimum degree ordering takes much less time than the other orderings.

Let us now look at a random triangulation of a unit cube. This matrix has 13824 rows and columns. Each face of the cube has a 22×22 regular grid of points. The remainder of the vertices are placed in the interior using quasi-random points, and the Delauney triangulation is computed.

	minimum degree			nested dissection		
seed	CPU	# entries	# ops	CPU	# entries	# ops
10101	9.2	5783892	6119141542	27.8	3410222	1921402246
10102	8.8	5651678	5959584620	31.4	3470063	1998795621
10103	8.8	6002897	6724035555	25.8	3456887	1986837981
10104	8.6	5888698	6652391434	29.6	3459432	1977133474
10105	8.5	5749469	6074040475	30.1	3567956	2223250836
	multisection			better of ND and MS		
seed	CPU	# entries	# ops	CPU	# entries	# ops
10101	26.9	3984032	2807531148	34.3	3410222	1921402246
10102	29.7	4209860	3266381908	37.0	3470063	1998795621
10103	23.5	4044399	2963782415	31.7	3456887	1986837981
10104	25.9	4239568	3325299298	34.8	3459432	1977133474
10105	27.2	4039078	2945539836	35.9	3567956	2223250836

Again there is about a factor of three in CPU time comparing minimum degree against the others, but unlike R2D10000, the minimum degree requires far fewer operations than the others. Note how the multisection ordering requires about 50% more operations than nested dissection. The situation can be reversed in other cases. That is the reason that we recommend using the wrapper that generates the better of the nested dissection and multisection orderings.

3 Front Trees

To illustrate the different types of front trees, and their transformations we do for the sake of efficiency, we will use an an example the matrix R2D100, a matrix generated by first randomly triangulating the unit square with 100 grid points. The resulting matrix has 100 rows and columns. We ordered the matrix using a generalized nested dissection algorithm from the **SPOOLES** library. On the left in Figure 2 is the triangulation. On the right we have labeled the grid points with their place in the nested dissection ordering. Note that vertices 90 through 99 form a separator of the graph. Vertices 0 through 47 are found on the right of the separator, vertices 48 through 89 are found on the left

3.1 Vertex elimination trees

Recall that the four ordering methods from Section 2 return an `ETree` object. There is another way to construct a tree using the `Graph` object and the permutation vectors. The following code fragment shows how to do this.

```
ETree   *vetree ;
int     *newToOld, *oldToNew ;
Graph   *graph ;

vetree = ETree_new() ;
ETree_initFromGraphWithPerms(vetree, graph, newToOld, oldToNew) ;
```

The `vetree` object in the code fragment above is a *vertex elimination tree* [20], [26], where each front contains one vertex.

Figure 2: R2D100: randomly triangulated, 100 grid points

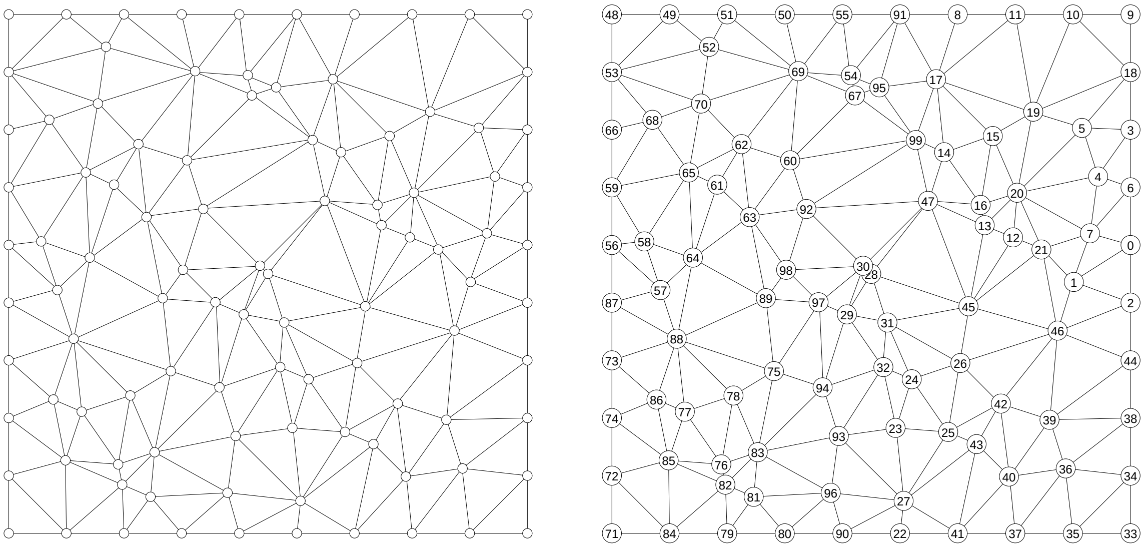


Figure 3 contains the vertex elimination tree for this ordering. The vertex elimination tree is a representation of the partial order by which the vertices in the graph may be eliminated.[1] The dependencies of the rows and columns form a tree structure. The leaves of the tree (our trees hang upside down with the leaves at the bottom and the root at the top) represent vertices which can be eliminated first. The parents of those leaf nodes can be eliminated next, and so on, until finally the vertices represented by the root of the tree will be eliminated last.

The elimination tree illustrates the dependence of the vertices. The basic rule is that a vertex *depends* only on its descendents and will *affect* only its ancestors. It should be clear that the tree allows us to identify independent, parallel computation. For example, the computation of the factor entries in the subtree rooted at vertex 47 is completely independent of the subtree rooted at vertex 89, so we could identify one process to compute the left subtree and another to compute the right subtree.

3.2 Fundamental supernode trees

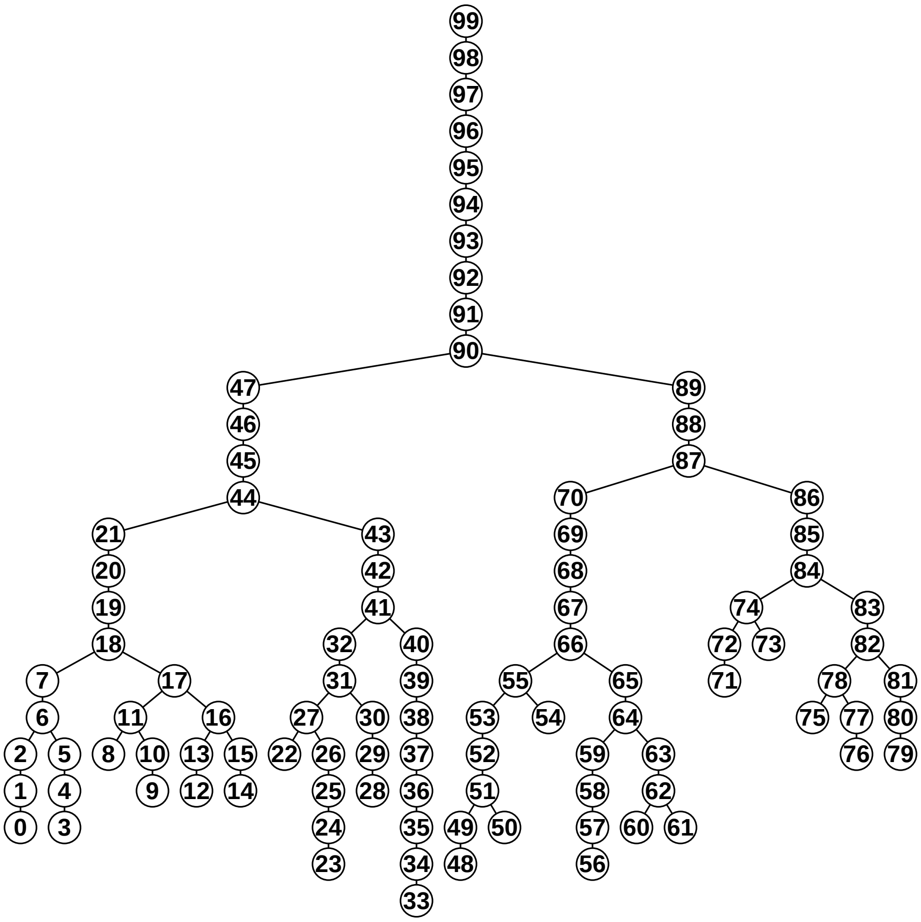
While the vertex elimination tree is useful to communicate the data dependencies, it is not a good granularity on which to base a factorization or solve, in serial or in parallel. It is important to group vertices together in some meaningful way to create larger data structures that will be more efficient with respect to storage and computation. Any grouping of vertices imposes a block structure on the matrix. The *fundamental supernode tree* [2] has these property: any node in the tree is

- either a leaf,
- or has two or more children,
- or its nonzero structure is not contained in that of its one child.

The top tree in Figure 4 shows the vertex elimination tree with the "front" number of each vertex superimposed on the vertex. The bottom tree is the fundamental supernode tree. Figure 5 shows the block partition

[1] Vertex j is the parent of i if j is the first vertex greater than i such that $L_{j,i} \neq 0$.

Figure 3: Vertex elimination tree for R2D100, 100 rows and columns



superimposed on the structure of the factor L. Note this one important property: within any block column and below the diagonal block, a row is either zero or dense.

The code fragment to convert a tree into a fundamental supernode tree is given below.

```
ETree   *fsetree, *vetree ;
int     maxzeros ;
IV      *nzerosIV ;

nzerosIV = IV_new() ;
IV_init(nzerosIV, vetree->nfront, NULL) ;
IV_fill(nzerosIV, 0) ;
maxzeros = 0 ;
fsetree = ETree_mergeFrontsOne(vetree, maxzeros, nzerosIV) ;
```

The `ETree_mergeFrontsOne()` method constructs a new `ETree` object from the `vetree` object. When a node J has a single child I, it looks to see whether merging I and J together will add more than a given number of zeroes into the block columns of I and J. (The nonzero rows of the block of I and J together is the union of the nonzero rows of blocks I and J separately, and all nonzero rows are stored as dense rows.) To create a fundamental supernode tree, the number of zeros allowed into a block column is zero, i.e., the nonzero structure of the fundamental supernode tree contains no zeros. The `nzerosIV` object contains a running count of the number of zero entries present in the factor storage. It will be used in later calls to other transformation methods.

3.3 Amalgamated or relaxed supernode trees

A factorization based on the fundamental supernode tree requires no more operations than one based on the vertex elimination tree. There are many small supernodes at the lower levels of the tree. By *amalgamating* small but connected sets of supernodes together into larger supernodes we can reduce the overhead of the processing all of the small supernodes at the expense of adding entries to the factors and operations to compute the factorization. This amalgamation of supernodes generally leads to an overall increase in efficiency [2], [9]. We call the result the *amalgamated* or *relaxed* supernode tree.

The top tree in Figure 6 shows the vertex elimination tree with the "front" number of each vertex superimposed on the vertex. The bottom tree is the amalgamated supernode tree. Figure 7 shows the block partition superimposed on the structure of the factor L.

The code fragment to create this amalgamated tree is found below.

```
ETree   *ametree ;

maxzeros = 20 ;
ametree = ETree_mergeFrontsAll(fsetree, maxzeros, nzerosIV) ;
```

This method will merge a node with *all* of its children if it will not result in more than `maxzeros` zeros inside the new block. On input, `nzerosIV` object keeps count of the number of zeroes already in the blocks of `fsetree`, and on return it will contain the number of zeros in the blocks of `ametree`.

3.4 Splitting large fronts

There is one final step to constructing the tree that governs the factorization and solve. Large matrices will generate large supernodes at the topmost levels of the tree. For example, a $k \times k \times k$ grid with a 27 point finite difference operator, when ordered by nested dissection, has a root supernode with k^2 rows and columns.

Figure 4: Top: vertex elimination tree with the vertices mapped to the fundamental supernode that contains them. Bottom: fundamental supernode tree.

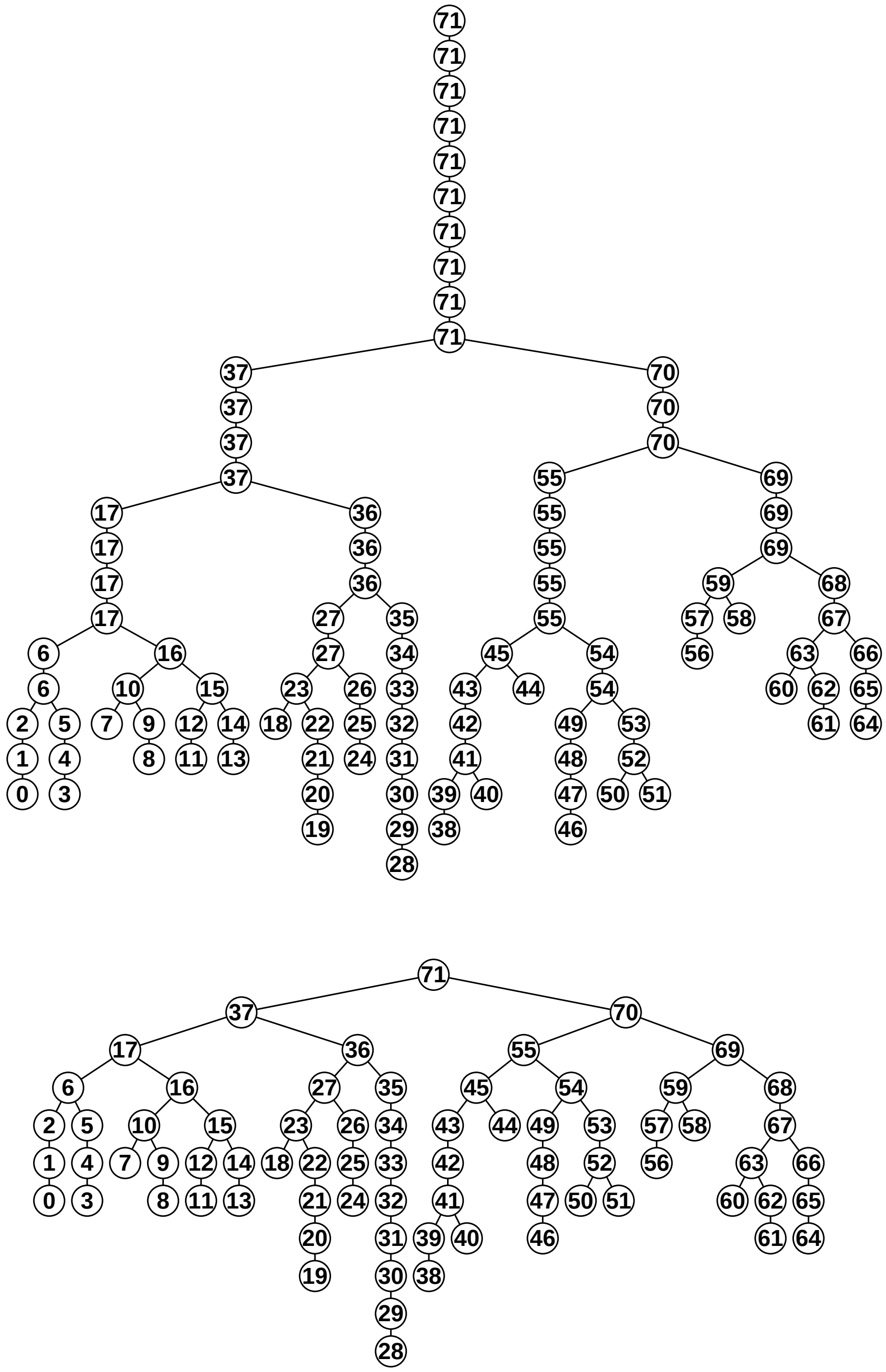


Figure 5: Block structure of L with the fundamental supernode partition.

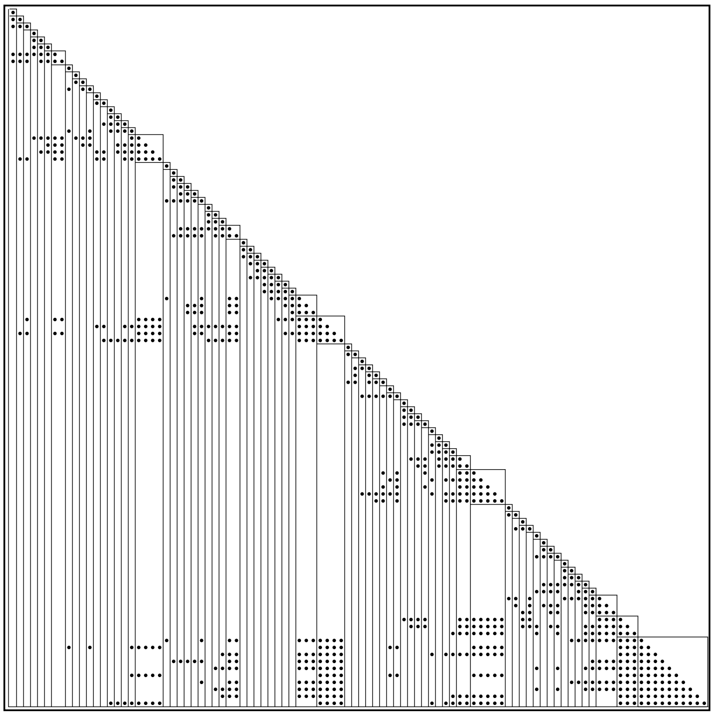


Figure 6: Top: fundamental supernode tree with the supernodes mapped to the amalgamated supernode that contains them. Bottom: amalgamated supernode tree.

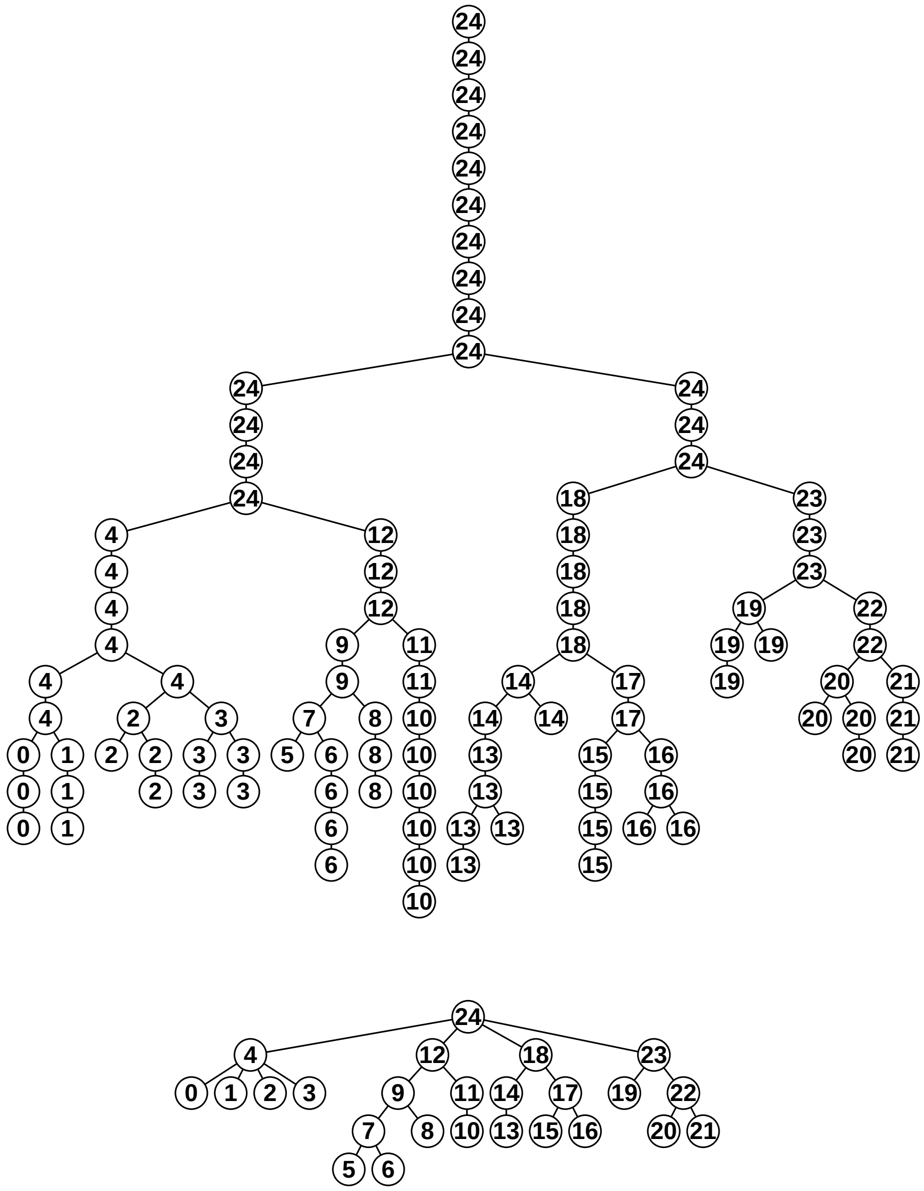
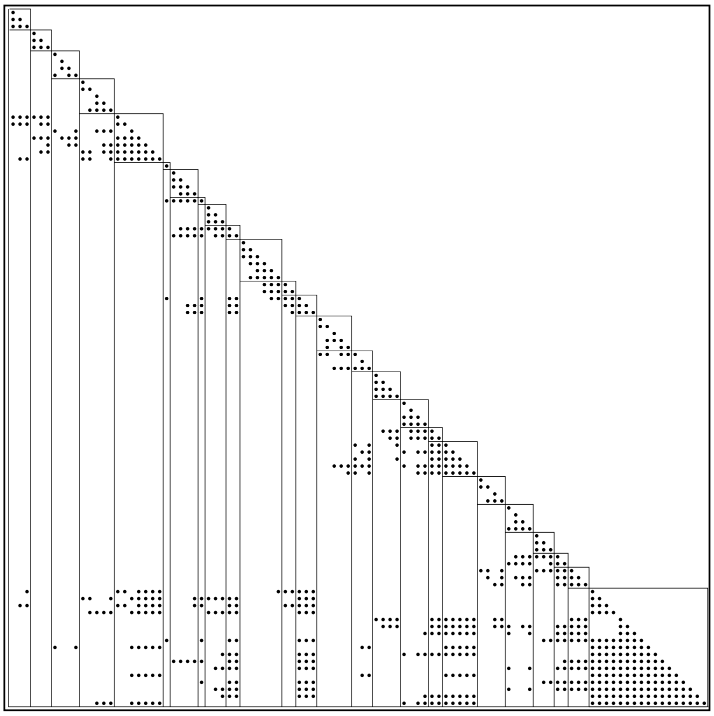


Figure 7: Block structure of L with the amalgamated supernode partition.



The data structure for a top level supernode can be very large, too large to fit into memory. In a parallel environment, we follow the convention that each node in the tree is handled by one process. Having a very large node at the top levels of the tree will severely decrease the parallelism available to the computations.

The solution to both problems, large data structures and limited parallelism, is to split large supernodes into pieces. We can specify a maximum size for the nodes in the tree, and split the large supernode into pieces no larger than this maximum size. This will keep the data structures to a manageable size and increase the available parallelism. We call the resulting tree the *front* tree because it represents the final computational unit for the factorization, the frontal matrix.

The amalgamated supernode tree has been transformed so that except for the leaf nodes, which are not changed, no node in the tree has more than four vertices. The top tree in Figure 8 shows the vertex elimination tree with the "front" number of each vertex superimposed on the vertex. The bottom tree is the amalgamated and split supernode tree. Figure 9 shows the block partition superimposed on the structure of the factor L. Splitting large nodes into smaller nodes will not increase the factor storage or operation counts, in fact, as we shall soon see, it is possible to decrease them slightly when compared to the amalgamated tree before splitting.

The code fragment to split the large fronts is found below.

```
ETree   *spetree ;
int     maxsize, seed ;

maxsize = 4 ;
spetree = ETree_splitFronts(ametree, NULL, maxsize, seed) ;
```

This method creates and returns an `ETree` object where each front has `maxsize` or fewer internal rows and columns, except for the fronts that are leaves in the tree. Here we imposed the condition that no non-leaf front has more than four vertices. The second parameter in the calling sequence is non-`NULL` if the graph has nonunit vertex weights. The last parameter is a seed for a random number generator. When we identify a front with more than `maxsize` internal rows and columns, there are many ways to split the front into smaller fronts. We try to keep the sizes of the fronts roughly equal, but which vertices to play into which fronts is not specified. We shuffle the vertices using a random number generator and assign vertices to smaller fronts in a block manner.

3.5 Results

This front tree is now the defining structure for the numerical factorization and solve steps. The structure of the front tree defines the order of the computations that will be carried out in the factorization and the solve. The composition of the front tree can have a profound effect on storage and performance of the factorization and solves.

Our `R2D100` matrix was small enough to illustrate the steps in the transformation of the front tree, but is not large enough to realistically display how the front tree influences the differences in storage and speed of the computations. Now we look at the `R3D13824` matrix. Table 3.5 contains some statistics for a sequence of front trees. The original front tree came from our nested dissection ordering.

There are 13824 rows and columns in the matrix, and 6001 fronts in the nested dissection tree. While there is an average of two rows and columns per front, most of the fronts are singleton fronts at the lower levels of the tree. The top level front has 750 internal rows and columns.

- In the first step we create an fundamental supernode tree with a call to `ETree_mergeFrontsOne()` with `maxzeros = 0`. We see that the number of fronts decreases by one and the number of entries does not change.

Figure 8: Left: tree after the large supernodes have been split. Right: tree with nodes mapped back to their amalgamated supernode.

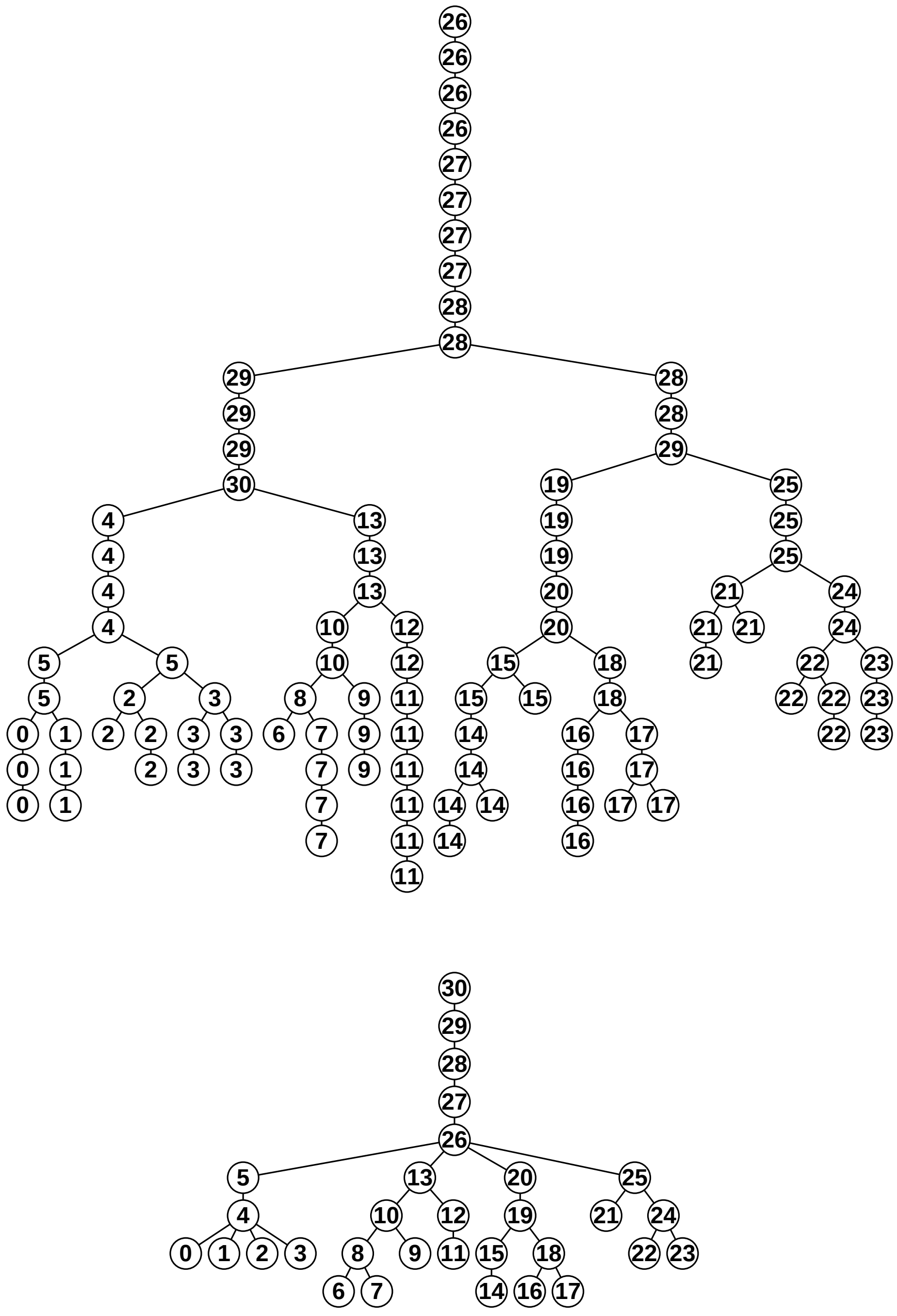


Figure 9: Block structure of L with the amalgamated and split supernode partition.

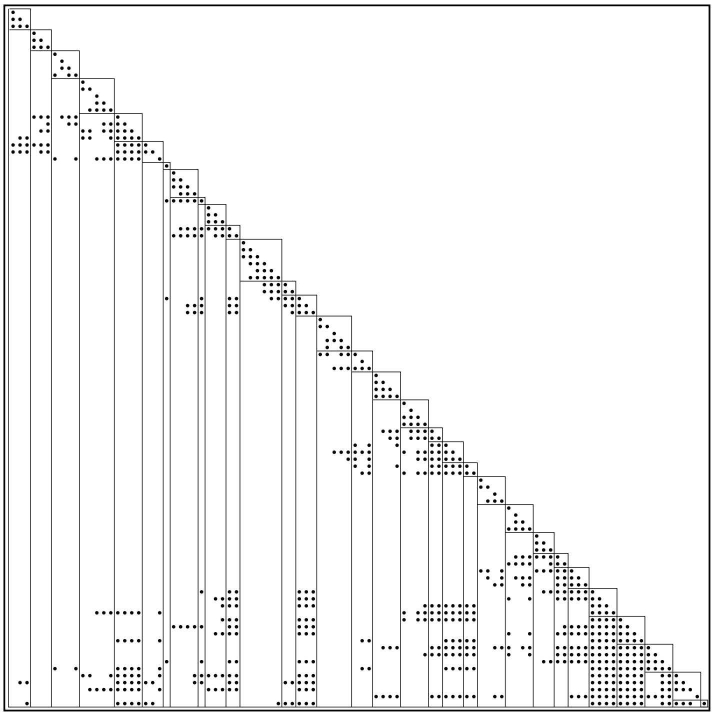


Table 1: R3D13824: front tree transformations

	CPU	# fronts	# indices	# entries	# operations
original		6001	326858	3459359	1981403337
fs tree	0.040	6000	326103	3459359	1981403337
merge one	0.032	3477	158834	3497139	2000297117
merge all	0.020	748	95306	3690546	2021347776
merge any	0.012	597	85366	3753241	2035158539
split	0.043	643	115139	3753241	2035158539
final	0.423	643	115128	3752694	2034396840

- The second step is also a call to `ETree_mergeFrontsOne()`, this time with `maxzeros = 1000`. Here we merge fronts with only one child with that child, in other words, only chains of nodes can merge together. Note how the number of fronts is decreased by almost one half, and the number of factor entries and operations increase by 1%.

- The third step is a call to `ETree_mergeFrontsAll()` with `maxzeros = 1000`, where we try to merge a node with all of its children if possible. The number of fronts decreases again by a factor of five, while the number of factor entries and operations increases by 7% and 2%, respectively, when compared with the original factor matrices.

- The fourth step is a call to `ETree_mergeFrontsAny()` with `maxzeros = 1000`, where we try to merge a front with any subset of its children. The number of fronts decreases further, and the factor entries and operations increase by 8% and 3%, respectively.

- In the fifth step is a call to `ETree_splitFronts()` with `maxsize = 64`, where we try split the large fronts into smaller fronts. Note that the number of factor entries and operations do not seem to increase, while the number of fronts increases by about 8%. In reality, a large front that is split into smaller fronts may have a non-dense block column structure, a one of its smaller fronts may have rows in its block column of L that are zero, whereas that same row in the larger front is nonzero.

Merging fronts and splitting fronts can have a large effect on the computational performance of a factor and solve. Table 3.5 contains some results for solving linear systems of equations for the `R3D13824` matrix using the five different front trees.

Table 2: R3D13824: factor and solve timings for five different front trees.

	factor				solve		total
	init	CPU	mflops	postprocess	CPU	mflops	CPU
original	4.0	131.7	15.0	5.0	7.3	7.6	148.0
fs tree	3.3	130.4	15.2	5.4	7.8	7.1	146.9
merge one	3.1	119.9	16.7	2.7	4.6	12.1	130.3
merge all	3.0	120.7	16.7	1.4	3.6	16.2	128.7
merge any	3.0	121.6	16.7	1.4	3.5	16.9	129.5
split	3.0	84.9	24.0	1.9	3.5	17.1	93.3

The first thing to notice is that factorization performance improves slightly as small fronts are merged together. The large improvement comes when the fronts are split. The explanation of this behavior is that all *inter-front* computation is done using BLAS3 kernels for the operation $Y := Y - L * D * U$, where L and U are dense matrices, D is diagonal or block diagonal with 1×1 and 2×2 pivots, and Y is dense. The *intra-front* computations, done entirely within the block columns of L and block rows of U, are done using BLAS1 kernels. This is necessary when pivoting for stability. Had we chosen to write BLAS3 kernels for the intra-front computations when pivoting is not enabled, the factorization timings for the first five front trees would have been higher. But splitting fronts into smaller fronts is necessary for parallel computations, so it made sense to make it the recommended route for serial computations as well. There would be very little difference in speed had the intra-front computations been done with BLAS3 kernels compared with using the final front tree, for the intra-front computations are a small fraction of the total number of operations.

The solve time improves dramatically when small fronts are merged together into larger fronts. Our solves are submatrix algorithms, where the fundamental kernel is an operation $Y_J := B_J - L_{J,I} X_I$ and $X_J := Y_J - U_{I,J} Y_J$, and is designed to be a BLAS2 kernel (when X and Y have a single column) or BLAS3 kernel (when X and Y are matrices). When fronts are small, particularly with one internal row and column, the submatrices that take part are very small. The overhead for the computations takes far more time than the computations themselves.

This multistep process of merging, merging again, etc, and finally splitting the front trees is tedious. There are simple methods that do the process in one step.

```
ETree   *etree, *etree2, *etree3 ;
int     maxfrontsize, maxzeros, seed ;

etree2 = ETree_transform(etree, NULL, maxzeros, maxfrontsize, seed) ;
etree3 = ETree_transform2(etree, NULL, maxzeros, maxfrontsize, seed) ;
```

Inside The `ETree_transform()` method is a sequence of four transformations:

- Merge small fronts into larger fronts using the `ETree_mergeFrontsOne()` method.
- Then merge small fronts into larger fronts using the `ETree_mergeFrontsAll()` method.
- Then merge small fronts into larger fronts using the `ETree_mergeFrontsAny()` method.
- Then merge a large front into a chain of smaller fronts using the `ETree_splitFronts()` method.

The `ETree_transform2()` method differs from the `ETree_transform()` method in that it omits the setp with `ETree_mergeFrontsAny()`. Either method will be suitable in most cases.

However, there are some times one method is to be preferred over the other. If we look again at the vertex elimination tree in Figure 3, we see the top level separator with nodes $\{90, \cdots, 99\}$, and the two second level separators with nodes $\{45, \cdots, 47\}$ and $\{87, \cdots, 89\}$. If one looks at their block columns in Figure 5 we see that either of the two second level separators could be merged with the top level separator without introducing any zero entries into the factor. Using the `ETree_mergeFrontsAny()` method could merge the top level separator with one of its two children, and produce an imbalanced tree, not as well suited for parallel computation had the two separators not been merged.

In a parallel environment, it is much more efficient to not merge the top level separator with either of its second level separators. The transformation methods in **SPOOLES 1.0** created front trees that were not as efficient for parallel processing, precisely because of the use of the "merge-with-any" step. This led us to write three separate merging methods to replace the single method from the 1.0 release, and thus give us the ability to avoid the trees unsuitable for parallel computation.

The values of `maxzeros` and `maxsize` will have a fair amount of influence on the efficiency of the factor and solves. This is illustrated in Table 3.5 for the `R3D13824` matrix and a number of different combinations of `maxzeros` and `maxsize`.

As the matrix size grows, the number of zero entries we can allow into a front can also grow. We recommend using `maxzeros` somewhere between `0.01*neqns` and `0.1*neqns`. The exact value isn't crucial, what is important is to have the smaller subtrees at the lower levels of the tree merged together. The `maxsize` parameter specifies the "panel" size of the large fronts, and so influences the granularity of the BLAS3 computations in the factorization and solve. If `maxsize` is too large, then too much of the computations in the factorization is done inside a front, which uses a slow kernel. If `maxsize` is too small, then the fronts are too small to get much computational efficiency. We recommend using a value between `32` and `96`. Luckily, the factor and solve times are fairly flat within this range. A value of `64` is what we customarily use.

References

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

Table 3: R3D13824: the influence of `maxzeros` and `maxsize`.

		factor				solve		total
`maxzeros`	`maxsize`	init	CPU	mflops	postprocess	CPU	mflops	CPU
0	∞	3.3	129.8	15.3	5.3	7.8	7.1	146.2
10	∞	3.5	129.2	15.3	3.3	5.3	10.5	141.3
100	∞	3.0	119.3	16.7	2.0	3.9	14.4	128.2
1000	∞	3.0	121.8	16.7	1.4	3.5	17.0	129.7
10000	∞	3.5	138.1	16.8	1.5	4.0	17.8	147.1
1000	32	3.3	89.8	22.7	2.6	4.1	14.7	99.8
1000	48	3.1	85.8	23.7	2.1	3.6	16.5	94.6
1000	64	3.1	85.2	23.9	1.9	3.5	17.1	93.7
1000	80	3.0	85.9	23.7	1.8	3.4	17.4	94.1
1000	96	3.0	86.1	23.6	1.8	3.4	17.6	94.3
0	32	3.3	100.3	19.8	6.6	7.9	7.0	118.1
0	48	3.2	97.0	20.4	6.1	7.6	7.3	113.9
0	64	3.2	95.8	20.7	5.4	6.9	8.0	111.3
0	80	3.2	96.7	20.5	5.5	7.1	7.8	112.5
0	96	3.2	97.2	20.4	5.2	6.7	8.3	112.3

[2] C. Ashcraft and R. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Software*, 15:291–309, 1989.

[3] C. Ashcraft and J. W. H. Liu. Using domain decomposition to find graph bisectors. *BIT*, 37, 1997.

[4] C. Ashcraft and J. W. H. Liu. Robust ordering of sparse matrices using multisection. *SIAM J. Matrix. Anal.*, 19:816–832, 1998.

[5] C. Ashcraft and J. W. H. Liu. Applications of the Dulmage-Mendelsohn decomposition and network flow to graph bisection improvement. *SIAM J. Matrix. Anal.*, 19:325–354, 1999.

[6] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 711–718, 1993.

[7] P. Berman and G. Schnitger. On the performance of the minimum degree ordering for Gaussian elimination. *SIAM J. Matrix Analysis and Applic.*, 11:83–88, 1990.

[8] T. Bui and C. Jones. A heuristic for reducing fill-in in sparse matrix factorization. In *Proceedings of Sixth SIAM Conference on Parallel Processing*, pages 445–452, 1993.

[9] I. Duff and J. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Software*, 6:302–325, 1983.

[10] J. A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

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

[12] A. Gupta. WGPP: Watson Graph Partitioning and sparse matrix ordering Package. Technical Report Users Manual, IBM T.J. Watson Research Center, New York, 1996.

[13] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. Technical Report SAND92-1460, Sandia National Laboratories, Albuquerque, NM, 1992.

[14] B. Hendrickson and R. Leland. The Chaco user's guide. Technical Report SAND93-2339, Sandia National Laboratories, Albuquerque, NM, 1993.

[15] B. Hendrickson and E. Rothberg. Improving the runtime and quality of nested dissection ordering. *SIAM J. Sci. Comput.*, 20:468–489, 1998.

[16] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, Minnesota, 1995.

[17] G. Karypis and V. Kumar. Metis 4.0: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, 1998. Available on the WWW at URL *http://www.cs.umn.edu/˜metis*.

[18] C. E. Leiserson and J. G. Lewis. Orderings for parallel sparse symmetric factorization. In *Parallel Processing for Scientific Computing*, pages 27–31, 1989.

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

[20] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Analysis and Applic.*, 11:134–172, 1990.

[21] E. Ng and P. Raghavan. Minimum deficiency ordering. In *Second SIAM Conference on Sparse Matrices*, 1996. Conference presentation.

[22] A. Pothen, H. Simon, and K.P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Analysis and Applic.*, 11:430–452, 1990.

[23] P. Raghavan. Parallel ordering using edge contraction. Technical Report CS-95-293, Dept. of Computer Science, The University of Tennessee, Knoxville, Tennessee, 1995.

[24] E. Rothberg. Robust ordering of sparse matrices: a minimum degree, nested dissection hybrid. unpublished, 1995.

[25] E. Rothberg and S. C. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM J. Matrix Anal.*, 19:682–695, 1998.

[26] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Trans. Math. Soft.*, pages 256–276, 1982.