Electronic Transactions on Numerical Analysis.
Volume 21, pp. 81-106, 2005.

ISSN 1068-9613.

ETNA
Kent State University
etna@mcs.kent.edu

LOCALITY OF REFERENCE IN SPARSE CHOLESKY FACTORIZATION METHODS*

ELAD ROZIN† AND SIVAN TOLEDO†

Dedicated to Alan George on the occasion of his 60th birthday

Abstract. This paper analyzes the cache efficiency of two high-performance sparse Cholesky factorization algorithms: the multifrontal algorithm and the left-looking algorithm. These two are essentially the only two algorithms that are used in current codes; generalizations of these algorithms are used in general-symmetric and general-unsymmetric sparse triangular factorization codes. Our theoretical analysis shows that while both algorithms sometimes enjoy a high level of data reuse in the cache, they are incomparable: there are matrices on which one is cache efficient and the other is not, and vice versa. The theoretical analysis is backed up by detailed experimental evidence, which shows that our theoretical analyses do predict cache-miss rates and performance in practice, even though the theory uses a fairly simple cache model. We also show, experimentally, that on matrices arising from finite-element structural analysis, the left-looking algorithm consistently outperforms the multifrontal algorithm. Direct cache-miss measurements indicate that the difference in performance is largely due to differences in the number of level-2 cache misses that the two algorithms generate. Finally, we also show that there are matrices where the multifrontal algorithm may require significantly more memory than the left-looking algorithm. On the other hand, the left-looking algorithm never uses more memory than the multifrontal one.

Key words. Cholesky factorization, sparse cholesky, multifrontal methods, cache-efficiency, locality of reference

AMS subject classifications. 15A23, 65F05, 65F50, 65Y10, 65Y20

1. Introduction. In the late 1980's and early 1990's it became clear that exploiting cache memories has become crucial for achieving high performance in sparse matrix factorizations, as in other algorithms. Independently, Rothberg and Gupta [23] and Ng and Peyton [22] showed that supernodes, groups of consecutive columns with identical nonzero structure, are key to exploiting cache memories in sparse Cholesky-factorizations. Neither paper, however, used a formal model to analyze the memory-system behavior of the factorization algorithms. Without such a model, different algorithms can only be compared experimentally, and it is not possible to predict how different algorithms behave under untested extreme conditions.

In this paper we fill this gap in the understanding of the cache behavior of sparse factorization algorithms. We present a formal model for analyzing cache misses, and show that the so-called *multifrontal* algorithms are better able to exploit caches for some input matrices while for some other input matrices the so-called *left-looking* algorithms are better. That is, they are incomparable in terms of cache efficiency. Experiments on specially constructed matrices validate our theoretical analyses. The experiments show, through running-time measurements and processor-event measurements, that some matrices cause many more cache misses in the left-looking algorithm than in the multifrontal one, while others cause many more cache misses in the multifrontal algorithm than in the left-looking one. On these classes of matrices, more cache misses are correlated with increased running time.

We also show that the total memory consumption of multifrontal algorithms may be significantly higher than that of left-looking algorithms, which sometimes causes problems in the lower levels of the memory hierarchy, such as paged virtual memory.

*Received August 4, 2004. Accepted for publication July 11, 2005. Recommended by J. Gilbert. This research was supported in part by an IBM Faculty Partnership Award, by grants 572/00 and 848/04 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities), and by grant 2002261 from the United-States-Israel Binational Science Foundation.

†School of Computer Science, Tel-Aviv Univesity, Tel-Aviv 69978, Israel (`stoledo@tau.ac.il`).

ETNA
Kent State University
etna@mcs.kent.edu

In experiments on typical matrices obtained from a sparse matrix collection, the left-looking algorithm usually outperforms the multifrontal one. On these matrices, the left-looking algorithm incurs more level-1 cache misses but fewer level-2 misses. However, a detailed analysis of several processor events, not only cache misses, suggests that the superior performance of the left-looking algorithm is mainly due to a reduced instruction count, not to differences in cache-miss rates. Since the asymptotic instruction-count behavior of the two algorithms is similar, and since the exact instruction counts depend to a large extent on hand and compiler optimizations of the inner loops, the superiority of the left-looking algorithm on these real-world matrices is probably implementation-dependent.

The papers of Rothberg and Gupta and Ng and Peyton examined three classes of factorizations. Rothberg and Gupta used a so-called *right-looking* factorization scheme. Ng and Peyton compared *left-looking* and *multifrontal* schemes. It is now generally accepted that left- and right-looking schemes are very similar, but that left-looking schemes are somewhat more efficient. Therefore, there does not seem to be a reason to prefer a right-looking factorization over a left-looking one (Rothberg and Gupta explain that they chose a right-looking over a multifrontal one to save memory, but they do not make any claims concerning left-looking factorizations). In this paper, we provide additional experimental evidence that substantiates Ng and Peyton's empirical conclusion that left-looking algorithms are more efficient than multifrontal ones. But we also show that left-looking algorithms may almost completely fail to exploit cache, even when the matrix is partitioned into wide supernodes. The multifrontal algorithm may also fail to exploit the cache, but it never fails when supernodes are wide. This observation is new.

The main conclusions from this research are that in practice, left-looking methods are slightly faster and require less memory, but in theory, both the left-looking and the multifrontal approaches have defects. A left-looking factorization, even a supernodal one, may suffer from poor cache utilization even when the matrix has wide supernodes; a multifrontal factorization may also suffer from poor cache reuse, and it may require a large working storage, which may force it to utilize lower, slower levels of the memory hierarchy.

The paper is organized as follows. Section 2 describes the basic structure of sparse Cholesky factorization and the so-called left-looking and multifrontal algorithms. Section 3 analyzes the two algorithms using simplified cache models. In Sections 4 and 5 we show that in terms of cache-miss rates, each algorithm is inferior to the other on some classes of matrices. Section 6 deals with the memory usage of the multifrontal algorithm. In Section 7 we describe our experimental results, on both synthetic matrices and real-world matrices. We present our conclusions in Section 8.

2. Sparse Cholesky Factorization. This section describes the basic structure of the sparse Cholesky factorization; for additional information, see the monographs of Duff, Erisman, and Reid [7] and of George and Liu [10], as well as the papers cited below. The factorization computes the factor L of a sparse symmetric positive-definite matrix $A = LL^T$. All state-of-the-art sparse Cholesky algorithms, including ours, explicitly represent only few nonzero elements in L, or none at all. Many algorithms do represent explicitly a small number of the zero elements when doing so is likely to reduce indexing overhead. The factor L is typically sparse, but not as much as A. One normally pre-permutes the rows and columns of A symmetrically to reduce fill (elements that are zero in A but nonzero in L).

A combinatorial structure called the *elimination tree of* A [26] plays a key role in virtually all state-of-the-art Cholesky factorization methods, including ours. The elimination tree (etree) is a rooted forest (a tree unless A has a nontrivial block-diagonal decomposition) with n vertices, where n is the dimension of A. The parent $p(j)$ of vertex j in the etree is defined to be $p(j) = \min_{i>j}\{L_{ij} \neq 0\}$. The elimination tree compactly represents dependencies

ETNA
Kent State University
etna@mcs.kent.edu

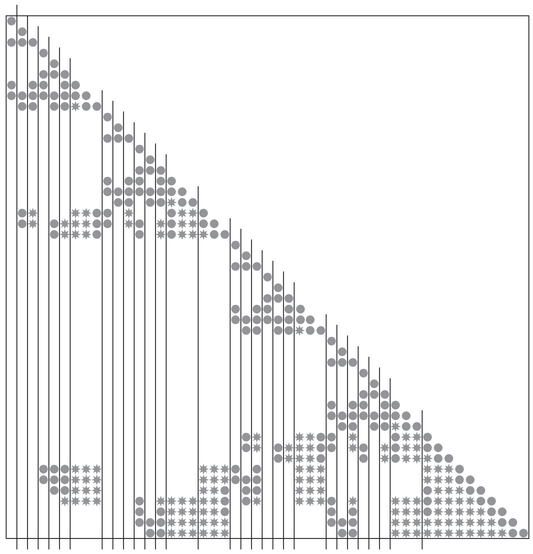

FIG. 2.1. *A fundamental supernodal decomposition of the factor of a matrix corresponding to a 7-by-7 grid problem, ordered with nested dissection. The circles correspond to elements that are nonzero in the coefficient matrix and the stars represent fill elements.*

in the factorization process and has several uses in sparse factorization methods [18]. The elimination tree can be computed from A in time that is essentially linear in the number of nonzeros in A.

Virtually all the state-of-the-art sparse Cholesky algorithms use a *supernodal decomposition* of the factor L, illustrated in Figure 2.1 [8, 22, 23]. The factor is decomposed into dense diagonal blocks and into the corresponding subdiagonal blocks, such that rows in the subdiagonal rows are either entirely zero or almost completely dense. In a *strict* supernodal decomposition, subdiagonal rows must be either zero or dense; the subdiagonal blocks in a strict decomposition can be packed into completely dense two-dimensional arrays. In an *amalgamated* [8] (sometime called *relaxed* [5]) decomposition, a small fraction of zeros is allowed in the nonzero rows of a subdiagonal block. Relaxed decompositions generally have larger blocks than strict ones, but the blocks contain some explicit zeros. Since larger blocks reduce indexing overheads and provide more opportunities for data reuse in caches, some amount of relaxation typically improves performance. Given A and its etree, a linear time algorithm can compute a useful strict supernodal decomposition of L called the *fundamental* supernodal decomposition [20]. This decomposition typically has fairly large supernodes and is widely used in practice. Finding a relaxed supernodal decomposition with larger supernodes is somewhat more expensive, but is usually still inexpensive relative to the numerical factorization itself [5, 8].

The supernodal decomposition is represented by a *supernodal elimination tree* or an *assembly tree*. In the supernodal etree, a tree vertex represent each supernode. The vertices are labeled 0 to $\sigma - 1$ using a postorder traversal, where σ is the number of supernodes. We associate with supernode j the ordered set Λ_j of columns in the supernode, and the unordered set Ξ_j of row indices in the subdiagonal block. The ordering of indices in Λ_j is some ordering consistent with a postorder traversal of the non-supernodal etree of A. We define $\lambda_j = |\Lambda_j|$ and $\xi_j = |\Xi_j|$. For example, the sets of supernode 29, the next-to-rightmost supernode in Figure 2.1, are $\Lambda_{29} = (37, 38, 39)$ and $\Xi_{29} = \{40, 41, 42, 46, 47, 48, 49\}$.

ETNA
Kent State University
etna@mcs.kent.edu


```
left-looking-factor(A,S)
  for each root j ∈ S
    call recursive-ll-factor(j,A)
  end for
end

recursive-ll-factor(j,A)
  for each child k of j
    call recursive-ll-factor(k,A)
  end for
  set V_{Λj∪Ξj,Λj} ← A_{Λj∪Ξj,Λj}
  for each child k of j
    call recursive-ll-update(j,V,k,L,S)
  end for
  factor V_{Λj,Λj} = L_{Λj,Λj} L^T_{Λj,Λj}
  solve L_{Ξj,Λj} L^T_{Λj,Λj} = V_{Ξj,Λj} for L_{Ξj,Λj}
  append L_{Λj∪Ξj,Λj} to L
end

recursive-ll-update(j,V,k,L,S)
  if Ξk ∩ Λj = ∅ return
  for each child k' of k
    call recursive-ll-update(j,V,k',L,S)
  end for
  V_{(Λj∪Ξj)∩Ξk,Λj∩Ξk} ←
      V_{(Λj∪Ξj)∩Ξk,Λj∩Ξk} − L_{(Λj∪Ξj)∩Ξk,Λk} L^T_{Λj∩Ξk,Λ'k}
end
```

FIG. 2.2. *Supernodal left-looking sparse Cholesky. These three subroutines compute the Cholesky factor L given a matrix A and its supernodal etree S.*

State-of-the-art sparse factorization codes fall into two categories: left-looking [22, 23] and multifrontal [8, 19]. The next paragraph explains how left-looking and multifrontal methods, including ours, work.

Given a matrix A and its supernodal etree S, the code shown in Figure 2.2 computes the Cholesky factor of L using a left-looking method. The code factors supernodes in postorder. To factor supernode j, the code copies the jth supernode block of A into working storage V. The code then updates V using blocks of L in the subtree rooted at j. Once all the updates have been applied, the code factors V to form the jth supernode block of L. The factorization of V is performed in two steps: we first factor its dense diagonal block and then solve multiple linear systems with the same dense triangular coefficient matrix $L^T_{\Lambda_j,\Lambda_j}$. Two sparsity issues arise in the subroutine `recursive-ll-update`: testing the condition $\Xi_k \cap \Lambda_j = \emptyset$ and updating V. The condition $\Xi_k \cap \Lambda_j = \emptyset$ can be tested in time $\Theta(\xi_k)$ by looking up the elements of Ξ_k in a bitmap representation of Λ_j, but the set of supernodes that update supernode j can also be computed directly from S and A [18]. The update to V is a sparse-sparse update, which is typically implemented by representing $V_{(\Lambda_j \cup \Xi_j)\cap\Xi_k,\Lambda_j\cap\Xi_k}$ in an n-by-ζ_{jk} array, where $\zeta_{jk} \geq |\Lambda_j \cap \Xi_k|$, and by using Ξ_k to index into this array. That is, V is unpacked into an array with full columns. The unpacking can be done inside the jk update, in which case $\zeta_{jk} = |\Lambda_j \cap \Xi_k|$, or once before we begin updating supernode j, in which

```
multifrontal-factor(A,S)
  for each root j ∈ S
    call recursive-mf-factor(j,A)
  end for
end

recursive-mf-factor(j,A)
  set F^(j)_{Λj∪Ξj,Λj∪Ξj} ← 0
  add F^(j)_{Λj∪Ξj,Λj} ← F^(j)_{Λj∪Ξj,Λj} + A_{Λj∪Ξj,Λj}
  for each child k of j
    U^(k)_{Ξk,Ξk} ←recursive-mf-factor(k,A)
    extend-add F^(j)_{Λj∪Ξj,Λj∪Ξj} ← F^(j)_{Λj∪Ξj,Λj∪Ξj} − U^(k)_{Ξk,Ξk}
    discard U^(k)_{Ξk,Ξk}
  end for
  factor F^(j)_{Λj,Λj} = L_{Λj,Λj} L^T_{Λj,Λj}
  solve L_{Ξj,Λj} L^T_{Λj,Λj} = F^(j)_{Ξj,Λj} for L_{Ξj,Λj}
  append L_{Λj∪Ξj,Λj} to L
  update U^(j)_{Ξj,Ξj} ← F^(j)_{Ξj,Ξj} − L_{Ξj,Λj} L^T_{Ξj,Λj}
  return U^(j)_{Ξj,Ξj}
end
```

FIG. 2.3. *Supernodal multifrontal sparse Cholesky. These subroutines compute the Cholesky factor L given a matrix A and its supernodal etree S.*

case we use $\zeta_{jk} = \lambda_j$. To allow finding the intersection $(\Lambda_j \cup \Xi_j) \cap \Xi_k$ quickly, the sets Ξ_k are kept sorted in etree postorder. This allows us to exploit the identity $(\Lambda_j \cup \Xi_j) \cap \Xi_k = \Xi_k \cap \{k' \colon k' \text{ is a descendant of } k\}$.

Figure 2.3 describes how multifrontal methods work. These methods factor supernodes in etree postorder. To factor supernode j, the algorithm constructs a symmetric $(\lambda_j + \xi_j)$-by-$(\lambda_j + \xi_j)$ full matrix $F^{(j)}$ called a *frontal matrix*. This matrix represents the submatrix of L with rows and columns in $\Lambda_j \cup \Xi_j$. We first add the jth supernode of A to the first λ_j columns of $F^{(j)}$. We then factor all the children of j. The factorization of child k returns a ξ_k-by-ξ_k full matrix $U^{(k)}$ called an *update matrix*. The update matrix contains all the updates to the remaining equations from the elimination of columns in supernode k and its descendants. These updates are then subtracted from $F^{(j)}$ in a process called *extend-add*. The extend-add operation subtracts one compressed sparse matrix from another containing a superset of the indices; the condition $\Xi_k \subseteq \Lambda_j \cup \Xi_j$ always holds. Once all the children have been factored and their updates incorporated into the frontal matrix, the first λ_j columns of F_j are factored to form columns Λ_j of L. A rank-λ_j update to the last ξ_j columns of $F^{(j)}$ forms $U^{(j)}$, which the subroutine returns. If j is a root, then $U^{(j)}$ is empty ($\xi_j = 0$). The frontal matrices in a sequential multifrontal factorization are allocated on the calling stack, or on the heap using pointers on the calling stack. The frontal matrices that are allocated at any given time reside on one path from a root of the etree to some of its descendants. By delaying the allocation of the frontal matrix until after we factor the first child of j and by cleverly restructuring the etree, one can reduce the maximum size of the frontal-matrices stack [16].

3. Theoretical Cache-Miss Analysis of the Algorithms. This section presents a theoretical analysis of cache misses in left-looking and multifrontal sparse Cholesky factorization algorithms. We focus on *capacity misses*, which are caused by the relatively small size of the cache. We ignore *compulsory* or cold-start misses, which read the input data into the cache, since their number is independent of the algorithm that is used. Any sparse Cholesky algorithm must read $\Theta(|A|)$ words into the cache and must write $\Theta(|L|)$ words back to memory. We also ignore *conflict* misses, which are caused by the mapping of memory addresses to cache locations. Conflict misses are relatively unimportant in highly associative caches, and there are simple techniques to reduce them even in low-associativity and direct-mapped caches [27].

In a sparse Cholesky factorization algorithm, data reuse within the cache can occur due to two different reasons. Suppose that during the processing of supernode j, the algorithm references a datum that is already in the cache, so no cache miss occurs. When did that datum arrive in the cache? If the datum was brought into the cache when the algorithm processed another supernode and was not evicted since, we say that this is an *inter-supernode* data reuse. If the datum was brought into the cache earlier in the factorization of supernode j, this is an *intra-supernode* data reuse. Inter-supernode data reuse is important near the leaves of the elimination tree, where supernodes tend to be smaller, or when the cache is very large. For example, in out-of-core factorization, many supernodes often fit simultaneously within main memory, so there is a significant amount of inter-supernode data reuse.

Analyzing both inter- and intra-supernode cache misses simultaneously is hard, because sparse-matrix factorizations are irregular computations. The elimination tree, which guides the computation, is often irregular, and supernode sizes and aspect ratios vary widely. Because of the irregularity, it is hard to derive closed-form expressions for bounds or estimates on cache misses, the kind of expressions that one can derive for structured computations such as dense-matrix computations [13, 21, 29], sorting [1, 15], and FFTs [1]. However, inter-supernode cache misses usually have only a minor influence on cache misses in the data caches close to the processor, which are small relative to the size of supernodes. Therefore, in this paper we focus mostly on intra-supernode cache misses, which are the dominant class of cache misses in the top-level caches. Inter-supernode misses, which are the dominant class in sparse out-of-core factorizations, have been carefully analyzed in that context, e.g., [24, 25].

We formalize the notion that inter-supernode cache data reuse is insignificant using the *cold-cache assumption*. Under this assumption, the cache is always empty when we start to factor a supernode. In other words, we ignore the reuse that results from data remaining in the cache when the processing of a supernode is completed.

We assume that the cache contains M words. We measure cache efficiency by *data reuse ratio*, which is the ratio of the total number of operations that an algorithm performs to the number of cache misses that it generates. This metric provides an approximation to the slowdown caused by cache misses; an algorithm with a high-data reuse ratio does not slow down by much, because it uses the cache effectively. An algorithm with a low ratio, on the other hand, slows down significantly because it has poor locality.

As we shall see, the data reuse ratio might be high when one supernode is factored and low when another is factored, even during the factorization of a single matrix. Therefore, we bound the ratio for the processing of each supernode separately. In other words, there is no simple way to characterize a sparse factorization algorithm as cache efficient or cache inefficient; an algorithm might exhibit good data reuse early in the factorization and poor data reuse later, for example.

We are now almost ready to analyze the sparse algorithms. The next theorem estimates the data-reuse ratio in the multifrontal algorithm. The estimate counts both capacity and compulsory misses (capacity misses are estimated under the cold-cache assumption).

Before we state the theorem, however, we must explain a technical assumption that the proof makes. The theorem below states, in effect, four bounds on data reuse: two upper bounds and two lower bounds. One of them, the $O(\sqrt{M})$ upper bound, relies on a known upper bound for data reuse in dense matrix multiplication [13]. This bound only holds for so-called *conventional matrix-matrix multiplication*, in which a product $A = BC$ is computed using sums of products, $A_{ij} = \sum_k B_{ik}C_{kj}$. In essence, the bound only holds for the data dependency of this matrix-multiplication algorithm, although any ordering of the summations is allowed. The bound may not hold for other matrix-multiplication algorithms. Furthermore, for the theorem below to hold, the data-dependency graph of the factorization of the diagonal block must include, as a subgraph, a matrix multiplication. This is true for all the conventional Cholesky elimination methods, but again, may not hold for completely different algorithms. Hence, the theorem assumes that the processing of the supernode is performed using so-called conventional algorithms.

The $\Omega(\sqrt{M})$ lower bound also depends on the use of conventional algorithms. This data-reuse bound applies to all the dense-matrix computations that are used during the processing of a supernode, as long as conventional algorithms are used. In fact, the same bound also applies to the use of Strassen's matrix-multiplication algorithm [9]. But in principle, there might be other ways to perform these computations for which the cache cannot be exploited.

THEOREM 3.1. *The data-reuse ratio associated with the processing of supernode j in the multifrontal algorithm is*

$$\Theta\left(min\left(\lambda_j, \sqrt{M}\right)\right) \ .$$

We assume that the processing of the supernode uses conventional Cholesky, triangular solve, and matrix multiplication algorithms, in the sense that these algorithms follow the data dependency graphs of the conventional algorithms; multiway addition is allowed in these dependency graphs (so the bound holds for any ordering of summations).

We associate the `extend-add` *operation in which a child j updates its parent with the processing of supernode j, not with the processing of the parent.*

Proof. Let us first count the number of floating-point operations performed during the processing of supernode j. The processing of supernode j also includes a factorization of its diagonal block, which is λ_j-by-λ_j, solving ξ_j independent triangular linear systems with λ_j unknowns each, and a symmetric rank-λ_j update to a symmetric ξ_j-by-ξ_j matrix, and later, an update to the parent of j. The numbers of floating-point operations required to perform the three dense-matrix computations is $\Theta(\lambda_j^3)$, $\Theta(\lambda_j^2\xi_j)$, and $\Theta(\lambda_j\xi_j^2)$, respectively. The number of floating-point operations to perform the parent's update is $\xi_j(\xi_i+1)/2 = \Theta(\xi^2)$.

Now let us count cache misses. Writing out the columns of L back to memory requires $\Theta(\lambda_j\xi_j)$ cache misses.

The `extend-add` operation in which a child j updates its parent $p(j)$ generates $\Theta(\xi_j^2)$ cache misses. For every element in the update matrix of j, the algorithm may generate up to 3 cache misses to read the value of the element and its row and column indices, and up to 4 cache misses to update a destination element in the frontal matrix of the parent. Two of these 4 may be required to read indirection information that maps the row and column indices to a memory location, a third is a read miss to retrieve an element of the frontal matrix, and the fourth is a write miss to write the updated value back to the frontal matrix. (These estimates assume that the algorithm uses a length-n integer array to map Λ_j and Ξ_j to compressed indices within the frontal/update matrices; this is the standard implementation of the multifrontal algorithm.) This proves the upper bound on cache misses for the `extend-add`. The lower bound is simply due to the cold-cache assumption, which implies that the update matrix of supernode j is not in the cache when it needs to update its parent.

ETNA
Kent State University
etna@mcs.kent.edu

The number of cache misses and floating-point operations associated with reading elements of A and adding them to the frontal matrix is the same, and bounded by $O(\lambda_j \xi_j)$. The number of cache misses to write out elements of L is$\Theta(\lambda_j^2 + \lambda_j \xi_j)$.

We now prove the lower bounds on data reuse. We assume that the three dense-matrix computations partition their arguments into n_b-by-n_b square submatrices (possibly with smaller matrices at the end of the tiling), where $n_b = \min(\lambda_j, \sqrt{M/3})$. For this value of n_b, any operation on up to three submatrices can be performed entirely in the cache, so the data-reuse ratio for these three dense operations is $\Omega(\min(\lambda_j, \sqrt{M/3}))$. This approach is now standard in dense-matrix algorithms; see, for example, [12, Section 1.4] or [28, Section 2.3].

The lower bounds include not only compulsory misses, but also $\Theta(\lambda_j \xi_j + \xi_j^2)$ compulsory misses. Therefore, the lower bounds hold even when we count the $\Theta(\lambda_j \xi_j + \xi_j^2)$ cache misses generated by reading elements of A, writing elements of L, and updating the parent of j. This completes the proof of the lower bounds.

We prove the upper bounds on data reuse in two steps. First, we prove that the data-reuse ratio is $O(\lambda_j)$. This bound is trivial. The total number of floating-point and other operations is $\Theta(\lambda_j^3 + \lambda_j^2 \xi_j + \lambda_j \xi_j^2)$, and the number of compulsory misses (under the cold-cache assumption) is $\Theta(\lambda_j^2 + \lambda_j \xi_j + \xi_j^2)$, so the data-reuse ratio is $O(\lambda_j)$.

The case where λ_j is small is somewhat harder to analyze. Our strategy is to show that if λ_j is large, so that $O(\lambda_j)$ does not provide a tight upper bound, then processing the supernode requires multiplying large matrices, which will cause many cache misses due to the memory size M. We reduce the processing of a supernode to matrix multiplication because there is a known bound on data reuse in matrix multiplication. The bound is due to Hong and Kung [13]; a more direct proof, which also specifies the constants involved appears in [30]. There are not known bounds on data reuse for other computations required during the processing of a supernode, such as Cholesky factorization and solving triangular linear systems, so we cannot rely on cache misses performed during these computations.

Assume that $\lambda_j > \sqrt{47.25M}$. If this is not the case, then $\lambda_j = O(\sqrt{M})$, so the $O(\lambda_j)$ upper bound on data reuse satisfies the theorem. If $\lambda_j > \sqrt{47.25M}$ then $\lambda_j/3 > \sqrt{5.25M}$, so $\lambda_j/3$ satisfies the condition of Theorem 1 in [30]. By that theorem, any implementation of the conventional matrix-multiplication algorithm on $(\lambda_j/3)$-by-$(\lambda_j/3)$ matrices must perform $\Omega(\lambda_j^3/\sqrt{M})$ cache misses. Consider a diagonal factor block of a supernode, viewed as a 3-by-3 block matrix with roughly equal block sizes,

$$\begin{pmatrix} F_{11} & F_{21}^T & F_{31}^T \\ F_{21} & F_{22} & F_{32}^T \\ F_{31} & F_{32} & F_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T & L_{31}^T \\ & L_{22}^T & L_{32}^T \\ & & L_{33}^T \end{pmatrix} .$$

During the factorization of this factor block, the algorithm must multiply L_{31} by L_{21}^T, to subtract the product $L_{31}L_{21}^T$ from F_{32}. These two matrices are each $(\lambda_j/3)$-by-$(\lambda_j/3)$, so to multiply them, the algorithm must perform $\Omega(\lambda_j^3/\sqrt{M})$ cache misses. If $\xi_j = O(\lambda_j)$, this implies that the data reuse is bounded by $O(\sqrt{M})$,since in that case the total number of operations that are performed during the processing of the supernode is $\Theta(\lambda_j^3 + \lambda_j^2 \xi_j + \lambda_j \xi_j^2) = \Theta(\lambda_j^3)$.

If $\lambda_j > \sqrt{47.25M}$ but $\xi_j > \lambda_j$, the dominant term in the expression $\Theta(\lambda_j^3 + \lambda_j^2 \xi_j + \lambda_j \xi_j^2)$ is $\Theta(\lambda_j \xi_j^2)$, and then the number of cache misses performed during the factorization of the diagonal block can not longer provide a useful upper bound on data reuse. If that is the case, we show that the number of cache misses performed during the computation of the update matrix is $\Omega(\lambda_j \xi_j^2/\sqrt{M})$, which provides an upper bound of $O(\sqrt{M})$ on the data reuse ratio. We again reduce the computation that is actually performed to a general matrix-

matrix multiplication. Even though the computation of the update matrix is a matrix-matrix multiplication, it is not general, because it multiplies a matrix by its transpose; no cache-miss lower bounds have been proved for this computation. Consider the entire frontal matrix as a 3-by-3 block matrix, where the first block F corresponds to the factor block and the other two blocks, of roughly equal size, correspond to rows and columns of the update matrix, which we denote by U:

$$\begin{pmatrix} F_1 & F_2^T & F_3^T \\ F_2 & 0 & 0 \\ F_3 & 0 & 0 \end{pmatrix} = \begin{pmatrix} L_1 & & \\ L_2 & 0 & \\ L_3 & 0 & 0 \end{pmatrix} \begin{pmatrix} L_1^T & L_2^T & L_3^T \\ & 0 & 0 \\ & & 0 \end{pmatrix} - \begin{pmatrix} 0 & 0 & 0 \\ 0 & U_{11} & U_{21}^T \\ 0 & U_{21} & U_{22} \end{pmatrix} .$$

The update block U_{21} is formed by the product $U_{21} = L_3 L_2^T$. The computation that produces U_{21} multiplies a λ_j-by-$(\xi_j/2)$ matrix by a $(\xi_j/2)$-by- λ_j matrix. By our assumptions on ξ_j and λ_j, we have $\xi_j/2 > \lambda_j > \sqrt{5.25M}$. It is easy to show, by a trivial extension of Theorem 1 in [30], that in that case, the matrix-matrix multiplication must perform $\Omega(\lambda_j \xi_j^2/\sqrt{M})$, which proves the data-reuse ratio bound of $O(\sqrt{M})$.

This concludes the proof. □

The analysis of the left-looking algorithm is, again, more complex. The complexity arises from two factors. The first factor is the same as the one that made the analysis under the infinite-cache assumption complex: the fact that the number of cache misses depends on the interaction between the updating and the updated supernodes. The second factor is the fact that the updated supernode might not fit in the cache.

We again make the same assumption concerning the use of conventional dense-matrix algorithms.

THEOREM 3.2. *The data-reuse ratio associated with the update from supernode k to supernode j in the left-looking algorithm is at most*

$$O\left(min\left(\lambda_{jk}, \sqrt{M}\right)\right)$$

and at least

$$\Omega\left(min\left(\lambda_k, \lambda_{jk}, \sqrt{M}\right)\right) \;,$$

where $\lambda_{jk} = |\Xi_k \cap \Lambda_j|$.

We assume that the update uses a conventional matrix multiplication algorithm.

Proof. Let $\xi_{jk} = |\Xi_k \cap (\Lambda_j \cup \Xi_j)|$. To perform the update, the algorithm must read into the cache $\lambda_k \xi_{jk}$ values of supernode k, due to the cold-cache assumption. The operation updates $\lambda_{jk}\xi_{jk}$ values of supernode j. These might be resident in the cache, or they might not, depending on the size of supernode j relative to the size of the cache and depending on previous updates to j. Therefore, the total number of cache misses during the update, even with an infinite cache, must be in the range between $\lambda_k \xi_{jk}$ and $\lambda_k \xi_{jk} + 2\lambda_{jk}\xi_{jk}$. The number of floating-point operations is $2\lambda_k \lambda_{jk} \xi_{jk}$. Therefore, if $\lambda_{jk} \leq \lambda_k$, the data-reuse ratio for an infinite cache is $\Theta(\lambda jk) = \Theta(\min(\lambda_{jk}, \lambda_k))$. If $\lambda_{jk} > \lambda_k$, the data-reuse ratio for an infinite cache is $O(\lambda jk)$ and $\Omega(\lambda k) = \Omega(\min(\lambda_{jk}, \lambda_k))$. These expressions, together with the fact that the data-reuse ratio for the update is always bounded by $\Theta(\sqrt{M})$, prove the theorem. □

The data reuse in final operations in the processing of supernode j, the Cholesky factorization of the diagonal block and the triangular solve that produces the subdiagonal block, is $\Theta(\min(\lambda_j, \sqrt{M}))$, since these are dense operations. The data-reuse ratio in this phase is comparable to the ratio in the multifrontal algorithm; but the updates from supernodes to the left may exhibit dramatically different data-reuse ratios.

ETNA
Kent State University
etna@mcs.kent.edu

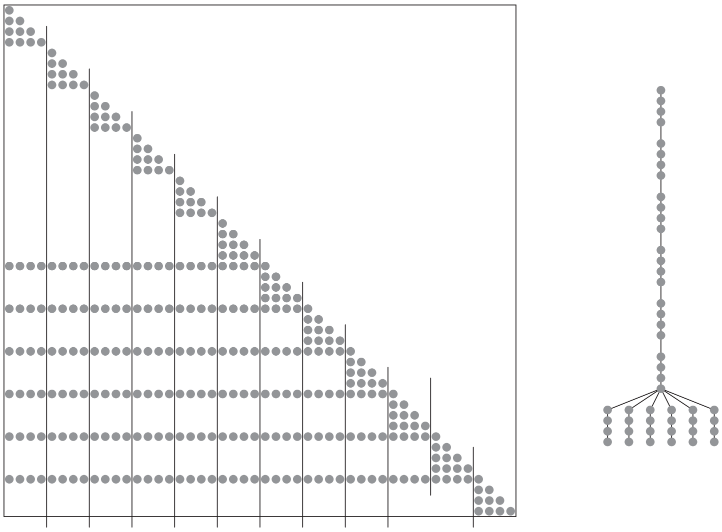

FIG. 4.1. *On the left, an example of a matrix on which the left-looking algorithm suffers from poor data reuse. Here $n = 48$ and $\lambda = 4$. When factored, this matrix (and any other matrix from the family discussed in the text) does not fill at all. On the right, the elimination tree of this matrix. Every group of 4 vertices (consecutive columns) forms a fundamental supernode.*

4. An Example of Poor Data Reuse in the Left-Looking Algorithm. In this section we show that the left-looking algorithm can sometimes perform asymptotically more cache misses then the multifrontal algorithm. Our analysis uses a family of matrices that cause poor locality of reference in the left-looking algorithm, but good locality in the multifrontal algorithm. Later in the paper we experimentally substantiate the theoretical analysis presented here.

When does the left-looking algorithm suffer from poor data reuse? By Theorem 3.2, the data reuse is poor when λ_{jk} is small. That is, when only one or few columns in supernode k update supernode j. When we update supernode j, we read supernode k into the cache, but perform relatively few arithmetic operations, because we only update few columns of supernode j. The matrices that we construct in this section cause this situation to happen during most of the supernode-supernode updates.

The nonzero structure of the matrices that we analyze is completely determined by their dimension n and by the width λ of all the supernodes (all the supernodes have exactly the same width). We assume that λ divides n. A matrix from this family is shown in Figure 4.1. The structure of the matrices is given by the expressions

$$\Lambda_k = \{k\lambda + 1, k\lambda + 2, \ldots, k\lambda + \lambda\}$$
$$a_{ij} \neq 0 \text{ iff } i, j \in \Lambda_k \text{ for some } k \text{ or } i = k\lambda + 1 \text{ for some } k \geq \frac{n}{2\lambda}\,.$$

It is easy to see that such matrices do not fill at all when factored. The supernodal elimination tree of these matrices consist of $n/2\lambda$ leaves, all of which are connected to a single supernode, which is connected by a simple path to the root.

To simplify the analysis, we select specific values of n, and λ, as a function of the cache size M. The analysis can be generalized to other values of n and λ but our selection shows the essential behavior. We place two constraints on n and λ. First, we require that any supernode plus a single column from its update matrix fit within the cache. Formally,

$$\frac{\lambda(\lambda+1)}{2} + \lambda\frac{n}{2\lambda} + \frac{n}{2\lambda} \le M \ .$$

We will later show that this ensures a high level of data reuse in the multifrontal algorithm. Second, we require that at most a quarter of the nonzeros in the $(2,1)$ block of the matrix, when viewed as a 2-by-2 block matrix with square blocks, fit in cache simultaneously. Formally,

$$\frac{1}{4}\cdot\frac{n^2}{4\lambda} = \frac{n^2}{16\lambda} \ge M \ .$$

We will show later that this ensures that the left-looking algorithm suffers from poor data reuse. It remains to show that there are values of n and λ that satisfy both constraints. We select $n = M/2$ and $\lambda = \sqrt{M}$. For the first constraint we have

$$\begin{aligned}\frac{\lambda(\lambda+1)}{2} + \lambda\frac{n}{2\lambda} + \frac{n}{2\lambda} &= \frac{M+\sqrt{M}}{2} + \frac{M}{4} + \frac{M}{4\sqrt{M}} \\ &= \frac{3}{4}M + \frac{3}{4}\sqrt{M}\end{aligned}$$

which is at most by M for any $M \ge 9$. For the second constraint we have

$$\frac{n^2}{16\lambda} = \frac{M^2}{4}\cdot\frac{1}{16\sqrt{M}} = \frac{M^{1.5}}{64}$$

which is at least M for any $M \ge 4096$. We conclude that our two constraints can be met for any $M \ge 4096$. (A more detailed analysis could bring down the value of M for which our analysis holds.)

In the multifrontal algorithm, the level of data reuse in the factorization of each supernode is at least λ, even under the cold cache assumption. The constraint on the value of M ensures that for each supernode, the diagonal and subdiagonal blocks can be factored in cache, and that the update matrix can be computed and written to main memory column by column without causing the eviction of already factored supernode elements. Therefore, the computation of the update matrix requires $\lambda\xi_k^2$ multiply-add operations, but the number of capacity misses is at most ξ_k^2 write misses plus ξ_k^2 read misses when update matrix is read from memory. Since there are n/λ supernodes and since $\xi_k \le n/2\lambda$, the total number of capacity misses is at most

$$\frac{n}{\lambda}\cdot 2\max_k \xi_k^2 = \frac{n^3}{2\lambda^3} \ .$$

To analyze the left-looking algorithm, we view the matrix as a 4-by-4 block matrix with square blocks. The entire $(4,1)$ and $(4,2)$ blocks update all the supernodes in the $(4,3)$ block. That is, all of the nonzeros in the $(4,1)$ and $(4,2)$ blocks update each supernode in the $(4,3)$ block. Since at most half of these nonzeros fit in cache, at least half of them must be read from main memory during the updating of each supernode (the other half may reside in cache from the update of the previous supernode). There are $n/4\lambda$ supernodes in the $(4,3)$ block. Therefore, the total number of capacity misses in the left-looking algorithm is at least

$$\frac{n}{4\lambda}\cdot\frac{n^2}{16\lambda} = \frac{n^3}{64\lambda^2} \ .$$

ETNA
Kent State University
etna@mcs.kent.edu

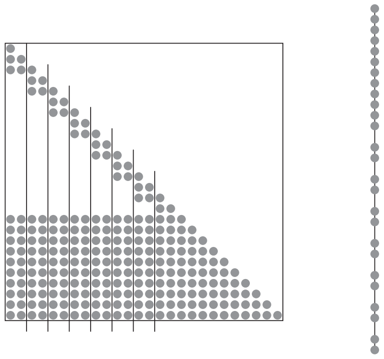

FIG. 5.1. *On the left, an example of a matrix on which the multifrontal algorithm suffers from poor data reuse. Here $n = 26$, $\ell = 12$ and $\lambda = 2$. When factored, this matrix (and any other matrix from the family discussed in the text) does not fill at all. On the right, the elimination tree of this matrix. Grouped vertices (consecutive columns) represent fundamental supernodes.*

The following theorem summarizes this analysis.

THEOREM 4.1. *For any large enough cache size M, there is a matrix on which the left-looking algorithm incurs at least a factor of $\sqrt{M}/32$ more cache misses than the multifrontal algorithm.*

In terms of constants, this analysis is fairly crude. It can be tightened using more elaborate arguments of the same type, but it already shows the essential issue as is. The issue is that the supernodes are fairly wide, but each supernode updates only one column in subsequent supernodes. We note that partitioning the matrix into narrower supernodes would not improve the data-reuse ratio of the multifrontal algorithm.

Furthermore, in both algorithms we only counted capacity cache misses that occur in the context of supernode-supernode updates. The total amount of arithmetic operations in these updates is $\Theta(n^3/\lambda^2)$. The multifrontal algorithm achieves here a data reuse of at least $\Theta(\lambda) = \Theta(\sqrt{M})$, which is optimal even for dense-matrix computations [13, 29]. On the other hand, the left-looking algorithm enjoys no asymptotic data reuse at all.

We experimentally show later in the paper that the multifrontal algorithm indeed performs much better than the left-looking algorithm on this class of matrices.

5. An Example of Poor Data Reuse in the Multifrontal Algorithm. We now show that there are also matrices on which the multifrontal algorithm incurs asymptotically more cache misses. These results, too, will be verified experimentally later in the paper. The analysis in this section is fairly similar to the analysis in the previous section, so we present it more tersely.

The family of matrices that we construct here are designed to have narrow supernodes. By Theorem 3.1, narrow supernodes lead to poor data reuse.

The nonzero structure of the matrices that we analyze here is determined by their dimension n, by the width λ of all the supernodes except the last, and by the width ℓ of the last supernode. We assume that λ divides $n - \ell$. A matrix from this family is shown in Figure 4.1. The structure of the matrices is given by the expressions

$$\Lambda_k = \{k\lambda + 1, k\lambda + 2, \ldots, k\lambda + \lambda\} \text{ for } k \leq \frac{n-\ell}{\lambda}$$
$$\Lambda_{1+(n-\ell)/\lambda} = \{n - \ell + 1, \ldots, n\}$$

$$a_{ij} \neq 0 \text{ iff } i,j \in \Lambda_k \text{ for some } k$$
$$\text{or } j \in \Lambda_k \text{ and } i = (k+1)\lambda + 1 \text{ for some } k \leq \frac{n-\ell}{\lambda} \text{ or } i > n-\ell.$$

Again, these matrices do not fill when factored. The supernodal elimination tree of these matrices is a simple path.

We select n, ℓ, and λ so that the last supernode plus one other supernode fit tightly into the cache:

$$\frac{\ell(\ell+1)}{2} + \ell\lambda = M$$

(more formally, we maximize ℓ and λ so that the expression on the left is bounded by M). We also assume that ℓ is much larger than λ.

In the left-looking algorithm, there are no capacity misses until we factor the last supernode. Each individual width-λ supernode fits into cache, and it remains in cache until we load its parent and update it. When we reach the last supernode, we might need to reread the last ℓ rows in all the previous supernodes into the cache, so the number of capacity misses is bounded by $\ell(n-\ell)$. In this case, the number of capacity misses is bounded by the number of compulsory misses.

The multifrontal algorithm suffers roughly $\ell^2/2$ capacity misses for every width-λ supernode. Each frontal matrix fills the cache almost completely, so by the time a frontal matrix is allocated and cleared, the update matrix of the child is no longer in cache, and reading it to perform the extend-add operation will generate about $\ell^2/2$ misses. Therefore, in the multifrontal algorithm the number of capacity misses is at least

$$\frac{n-\ell}{\lambda} \cdot \frac{\ell^2}{2} .$$

This number is a factor of $\ell/2\lambda$ larger than the number of misses in the left-looking algorithm. For λ much smaller than ℓ, this is a factor of about

$$\frac{\sqrt{M}}{\sqrt{2}\lambda} ,$$

and in particular, for $\lambda = 1$ the ratio is $\sqrt{M/2}$. This ratio is asymptotically the inverse of the ratio in the example of the previous section.

The number of compulsory misses (the size of A and its factor) is roughly $\ell(n-\ell)$. The total amount of arithmetic operations is $\Theta(\ell^2(n-\ell))$. Since the total number of cache misses, both compulsory and capacity, in the left-looking algorithm is only $\Theta(\ell(n-\ell))$, it achieves a data-reuse level of about $\Theta(\ell) = \Theta(\sqrt{M})$. In contrast, the multifrontal algorithm achieves only a data-reuse level of about $\Theta(\lambda)$, or almost no data use for small λ.

The reader might be concerned that if frontal matrices were only half the size of the cache (smaller ℓ), the multifrontal algorithm could achieve perfect data reuse. For the given class of matrices, the multifrontal algorithm could maintain two in-cache arrays that can each hold one frontal matrix, and simply assign one of them to each supernode. Since the etree is a path, we never need more than two frontal matrices. However, if we delete one row in each supernode so that all the width-λ supernodes are children of the width-ℓ supernode, no such optimization would be possible.

The preceding discussion proves the following theorem:

THEOREM 5.1. *For any large enough cache size M, there is a matrix on which the multifrontal algorithm incurs at least a factor of $\Omega(\sqrt{M})$ more cache misses than the left-looking algorithm.*

ETNA
Kent State University
etna@mcs.kent.edu

6. Memory Requirements for Sparse Cholesky Algorithms. The previous two sections proved that for some matrices the data locality in the multifrontal algorithm is better than in the left-looking one, while for other matrices left-looking does better. Is the left-looking algorithm superior in any other way? In this section we show that the answer is yes: the multifrontal algorithm uses more memory, sometimes by a significant amount.

The left-looking algorithm uses essentially no auxiliary memory beyond the memory that is required to store the factor L. It does need an integer array of length n to assist in indirect access to sparse columns, and it needs a temporary array in order to perform the updates using a dense matrix-multiplication routine. These two auxiliary arrays are never larger than L and are typically much smaller. Therefore, the algorithm uses $\Theta(|L|)$ storage, usually with a constant very close to 1.

We will show below that the multifrontal algorithm sometimes uses much more than $\Theta(|L|)$ memory. This reduces the memory available to other programs running on the same computer, it may cause failure when there is not enough memory, and it may cause worse data locality. When L fits in cache (or in main memory), the left-looking algorithm works solely with data structures in the cache (main memory), but the multifrontal algorithm may experience cache misses (virtual-memory page faults).

The exact amount of storage that is used by a sequential multifrontal algorithm (parallel algorithms use more) depends on three factors: the nonzero structure of the factor, the order in which supernodes are factored, and the allocation schedule that is used. The nonzero structure determines the size of supernodes and the dependencies among them, which are captured in the elimination tree. The ordering of the factorization determines when update matrices must be allocated. The allocation schedule determines when they can be freed: when a frontal matrix is allocated, all the update matrices of its children can update it and be deallocated. From that point on, any child that is factored can immediately update the frontal matrix. But when a frontal matrix is allocated later, the update matrices of its children must be kept allocated until the parent is allocated.

Existing multifrontal algorithm use one of two allocation strategies, or slight variants thereof. One strategy is to allocate the parent's frontal matrix as soon as possible, to ensure that at most one update matrix of one child is allocated at any given time. There is no point in allocating the parent before the first child is factored, but to prevent two update matrices of two children from being allocated simultaneously, the parent is allocated just after the first child is factored. This is the strategy that we analyze first. Another common strategy is to allocate the parent only after all the children have been factored. We analyze this variant near the end of the section.

We start with the *allocate-parent-after-the-first-child* strategy. The analysis again uses a class of synthetic matrices of size $n = (2^k - 1) + (m - 1)$ for some integers k and m. The elimination tree of these matrices consists of a complete binary tree with $2^k - 1$ vertices, and a linear chain with m vertices, one of which is the root of the binary tree. Each column in the binary-tree portion of the matrix except the last is a separate fundamental supernode. The m vertices in the chain part of the tree form a single supernode. Each column in the tree portion updates the entire training supernode. Figure 6.1 shows an example of a matrix from this class.

These matrices do not fill during elimination (as in Section 4, we could have constructed these synthetic matrices so that they start up sparse and fill to the pattern shown). The number of nonzeros in them, and in the factor L, is

$$m\left(2^k - 1\right) + 2\left(2^k - 1\right) - 1 + \frac{m(m-1)}{2} = \Theta\left(2^k m + \frac{m^2}{2}\right) .$$

ETNA
Kent State University
etna@mcs.kent.edu

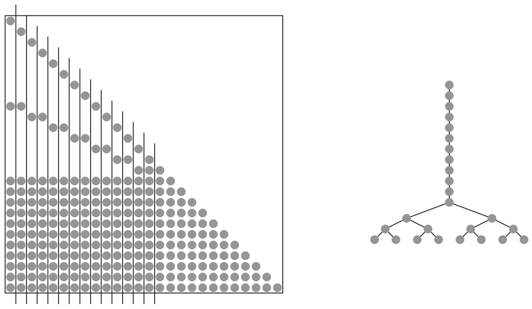

FIG. 6.1. *On the left, an example of a matrix on which the multifrontal algorithm suffers from excessive memory usage. Here* $k = 4$ *and* $m = 12$*, so* $n = (2^4 - 1) + (12 - 1)$*. When factored, this matrix (and any other matrix from the family discussed in the text) does not fill at all. On the right, the elimination tree of this matrix. In the first* 14 *columns, every column is a fundamental supernode; the last* 12 *form a fundamental supernode.*

For $m > 2^{k+1}$, the second term inside the Θ notation is larger.

When the multifrontal algorithm factors the last leaf column (the rightmost one), a frontal matrix must be allocated in every vertex from that leaf to the root of the binary tree. This is true because each vertex v along that path (except the leaf) has one child c that was already factored, so either c's update matrix or v's frontal matrix must be currently allocated. Each of these k frontal or update matrices is at least m-by-m, so the total amount of memory currently allocated is km^2. For $m > 2^{k+1}$, this is a factor of at least k larger than the size of L.

We chose this particular family of matrices because even memory-optimized versions of the multifrontal algorithm require km^2 memory to factor them. Two optimizations can reduce the amount of memory used by the multifrontal algorithm. The first is to allocate a frontal matrix for vertex v only after the update matrix from the first child c of v has been factored. This eliminates the need to keep v's frontal matrix in memory while the subtree rooted at v is factored. The second optimization, due to Liu [17], reorders the children of v so that the child whose subtree requires the most memory is factored first. Since in our example the tree is binary and is completely symmetric with respect to child ordering, these optimizations do not reduce the maximal memory usage.

Practitioners have found that the size of the stack of update and frontal matrices in the multifrontal algorithm is typically small relative to the size of L. This synthetic example shows that there are matrices for which this rule of thumb does not hold. This was also observed by Rothberg and Schreiber [24] on a real-world matrix, which caused excessive I/O activity in an out-of-core multifrontal code.

We note that for large values of m, the matrices in our example are fairly dense, and some codes would treat them as dense. This can be achieved either by noticing the number of nonzeros is a large fraction of the total number of elements, or by an automatic amalgamation algorithm that might, in this case, amalgamate the entire matrix into a single supernode. However, even such codes would treat the matrix as sparse for some lower value of m, for which the multifrontal algorithm could still use much more memory than the left-looking algorithm.

We also note that an aggressive amalgamation strategy that amalgamates vertices with multiple children would be beneficial on matrices similar to our synthetic matrices, since it is exactly the multiple-child condition that causes excessive memory usage.

We now show that a similar result holds for the *allocate-parent-last* strategy, and even for codes that select the best of the two strategies based on the nonzero structure of the matrix

ETNA
Kent State University
etna@mcs.kent.edu

TABLE 7.1
The test matrices from the PARASOL *test-matrix collection. The table shows, for each matrix, the dimension of the matrix, the number of nonzero entries in the matrix, the number of nonzero entries in the Cholesky factor under* METIS *ordering and without supernode amalgamation, and the number of floating-point operations required to compute the factor (again assuming no amalgamation). The third column is used as the horizontal axis in the plots that follow.*

Matrix	dim(A)	nnz(A)	nnz(L)	flops(L)
bmw7st-1	1.41e5	3.74e6	2.55e7	1.08e10
crankseg-1	5.28e4	5.33e6	3.34e7	3.19e10
crankseg-2	6.38e4	7.11e6	4.36e7	4.58e10
inline-1	5.04e5	1.87e7	1.76e8	1.52e11
ldoor	9.52e5	2.37e7	1.43e8	7.39e10
m-t1	9.74e4	4.93e6	3.38e7	2.14e10
msdoor	4.16e5	1.03e7	5.30e7	1.77e10
oilpan	7.38e4	1.84e6	9.21e6	2.81e09
ship-003	1.22e5	9.73e6	6.08e7	8.23e10
shipsec1	1.41e5	3.98e6	3.91e7	3.70e10
shipsec5	1.80e5	6.15e6	5.35e7	5.62e10
thread	2.97e4	2.25e6	2.45e7	3.60e10
vanbody	4.70e4	1.19e6	5.89e6	1.30e09
x104	1.08e5	5.14e6	2.72e7	1.42e10

and the elimination tree.

The analysis below relies on the fact that we can modify the example shown in Figure 6.1 so that the first $n/2$ columns form an elimination tree of any form. We can do so while maintaining the invariants that each one of these columns forms a separate supernode and its update matrix is $(1+n/2)$-by-$(1+n/2)$, updating its parent and the entire $(n/2)$-by-$(n/2)$ trailing submatrix.

If the allocation strategy always allocates the parent only after all its children have been factored, they by making all the first $n/2$ columns siblings (children of column $n/2+1$), then we force the algorithm to store $n/2$ update matrices, each $(1+n/2)$-by-$(1+n/2)$. The total storage required for this tree and this allocation strategy is approximately $n^3/8$, a factor of $\Theta(n)$ more than that required for the Cholesky factor itself.

If the algorithm can choose between the two strategies discussed above, then the worst elimination-tree structure is one in which the first $n/2$ columns form a tree whose depth is similar to the degree d of tree vertices. This happens when the degree d is approximately $\log n/\log\log n$. In such a tree, allocating the parent after the first child is factored or allocating it after all children have been factored, both require storage proportional to $\Theta(n^2)$ times d. In this case, the total amount of storage required is a factor of $\Theta(\log n/\log\log n)$ more than required for the Cholesky factor.

Finally, we note that due to the cold-cache assumption, the allocation schedule has no influence on the cache-miss bounds that we proved in Section 3. In practice, the cache is not flushed after every supernode, so the allocation and extend-add schedule does have an influence on the actual number of cache misses. It is not clear, however, which allocation strategy exploits the cache better, and whether the difference is significant.

7. Results. This section presents experimental results that compare the multifrontal and the left-looking algorithms on two sets of matrices. We used two classes of matrices for testing our algorithm, matrices from the PARASOL test-matrix collection, and the synthetic matrices that were described in Section 4. The PARASOL matrices are described in Table 7.1.

ETNA
Kent State University
etna@mcs.kent.edu

The experiments were designed to resolve one practical question and to demonstrate several aspects of our analysis. The question that the experiments resolve (to the extent possible in an experimental analysis), is whether the left-looking algorithm exhibits poorer data locality than the multifrontal one on matrices that arise in practice. Our analysis indicates that this is not the case: the two algorithms exhibit similar cache-miss rates on matrices from the PARASOL collection, and the left-looking algorithm is usually faster. This result is consistent with the experimental results of Ng and Peyton [22].

The experiments also validate the theoretical analysis that we presented in Section 4, and shows that cache misses can have a dramatic effect on the performance of these algorithms.

7.1. Test Environment. We performed the experiments on several machines. One machine, which we used for assessing the differences between the left-looking and the multifrontal algorithm on a set of real-world matrices, is an Intel-based workstation. This machine has a 2.4 GHz Pentium 4 processors with a 512 KB level-2 cache. This machine has 2 GB of main memory (dual-channel with DDR memory chips). Due to the size of main memory, paging is not a concern on this machine: the entire address space of a process can reside in the main memory.

This machine runs Linux with a 2.4.22 kernel. We compiled our code with the GCC C compiler, version 3.3.2, and with the `-O3` compiler option. We used ATLAS[1] Version 3.4.1 [31] by Clint Whaley and others for the BLAS. This version exploits vector instructions on Pentium 4 processors (these instructions are called SSE2 instructions). Using these BLAS routines on this machines, our in-core left-looking sparse factorization code factors real-world matrices at rates of up to 2.2×10^9 flops (e.g., the matrix THREAD from the PARASOL test-matrix collection). The same sparse code factors a dense matrix of dimension 4000 at a rate of approximately 2.8×10^9.

We also performed experiments on a slower Pentium 3 computer. On this machine we were able to directly count cache misses and other processor events. (The software that we used to measure these events does yet not support the Pentium 4 processor.)

This machine is an Intel-based server with two 0.6 GHz Pentium 3 processors. One processor was disabled by the operating system in order to avoid measurement errors. These processors have a 16 KB 4-way set associative level-1 data cache with 32 bytes cache lines and a 256 KB 8-way set associative level-2 cache, also with 32 byte lines. There is a separate level-1 instruction cache, but the level-2 cache is used for both data and instructions. In our experiments, the use of the level-2 cache for storing instructions is insignificant. We explain later how we came to this conclusion. The processor also has a transaction-lookaside buffer (TLB), which is used to translate virtual addresses to physical addresses. The TLB has 32 entries for mapping 8 KB data pages and is 4-way set associative. TLB misses, like cache misses, can also degrade performance, but we did not count them in these experiments.

This machine also has 2 GB of main memory consisting of 100 MHz DRAM chips.

This machine runs Linux with a 2.4.20 kernel. We compiled our code with the GCC C compiler, version 3.3.2, and with the `-O3` compiler option and we used a Pentium-3-specific version of ATLAS. We also performed limited experiments with another implementation of the BLAS, the so-called *Goto* BLAS, version 0.9[2]. The results, which are not shown in the paper, exhibited similar performance ratios between the different sparse factorization algorithms. The results that we present, therefore, are not highly dependent on the implementation of the BLAS.

We measured cache misses and other processor events using the PERFCTR kernel module

[1] `http://math-atlas.sourceforge.net`
[2] `http://www.cs.utexas.edu/users/flame/goto/`

ETNA
Kent State University
etna@mcs.kent.edu

and using the PAPI library, which provides unified access to performance counters on several platforms. We used version 2.3.4.1 of PAPI and the version of PERFCTR that came bundled with PAPI. Together, these tools allowed us to instrument our code so that it counts one specific processor event during the numerical factorization phase (or none). We measured multiple events by running the same code on the same input matrix with the same parameters several times. The running times of these multiple runs were very close to each other, which implies that this method of measurements is robust.

The graphs and tables use the following abbreviations: TAUCS (our sparse code), MUMPS (MUMPS version 4.3), LL (left-looking), and MF (multifrontal).

7.2. Baseline Tests. To establish a performance baseline for our experiments, we compare the performance of our code, called TAUCS, to that of MUMPS version 4.3 [4, 2, 3]. MUMPS is a parallel and sequential in-core multifrontal factorization code for symmetric and unsymmetric matrices. We used METIS[3] [14] version 4.0 to symmetrically reorder the rows and columns of the matrices prior to factoring them. We tested the sequential version, with options that tell MUMPS that the input matrix is symmetric positive definite and that instruct it to use METIS to preorder the matrix. We used the default values for all the other run-time options. This setup result in a multifrontal factorization that is quite similar to TAUCS's in-core multifrontal factorization.

TAUCS's multifrontal factorization allocates the parent's frontal matrix after the first child has been factored.

We compiled MUMPS, which is implemented in Fortran 90, using Intel's Fortran Compiler for Linux, version 7.1, and with the compiler options that are specified in the MUMPS-provided `makefile` for this compiler, namely `-O`. We linked MUMPS with the same version of the BLAS that are used for all the other experiments, and verified the linking by calling `ATL_buildinfo`, an ATLAS routine that prints ATLAS's version.

We compared the two codes on matrices from the PARASOL test-matrix collection. The matrices were selected arbitrarily from the test-matrix collection. TAUCS was able to factor all of the matrices in our test suite, but MUMPS ran out of memory on the two largest matrices.

The results of the baseline tests, which are presented in Figure 7.1, show that the performance of TAUCS is quite similar to the performance of MUMPS. TAUCS's relaxed-supernode multifrontal factorization is usually slightly faster on these matrices. This test is not a comprehensive comparison of these codes, and we do not claim that TAUCS is faster in general. The differences can be due to the different compilers that were used, to different supernode amalgamation strategies, or to different ways of using the BLAS and LAPACK (our code factors dense matrices of dimension 4000 in a little over 7.6 seconds, whereas MUMPS factors the same matrices in about 23.6 seconds; in our case the factorization is performed by a single call to LAPACK's factorization routine DPOTRF, so MUMPS clearly uses LAPACK differently). However, the results do indicate that the performance of TAUCS, which we use compare the performance of the left-looking and the multifrontal algorithms, is representative of high-quality modern sparse Cholesky factorization codes.

7.3. Relative Performance on Real-World Matrices. Figure 7.1 shows that on real-world matrices arising from finite-element models, the left-looking algorithm performs better. This is true for both amalgamated/relaxed supernodes, and for exact fundamental supernodes. (Although there can be fairly significant differences between relaxed and exact supernodes; this is well known.) In this family of matrices, there are actually no exceptions to this observation. The difference in running times is often more than 10% and sometimes more than 20%. The matrices were all ordered using METIS version 4.0 prior to factoring them.

[3] `http://www-users.cs.umn.edu/~karypis/metis/`

ETNA
Kent State University
etna@mcs.kent.edu

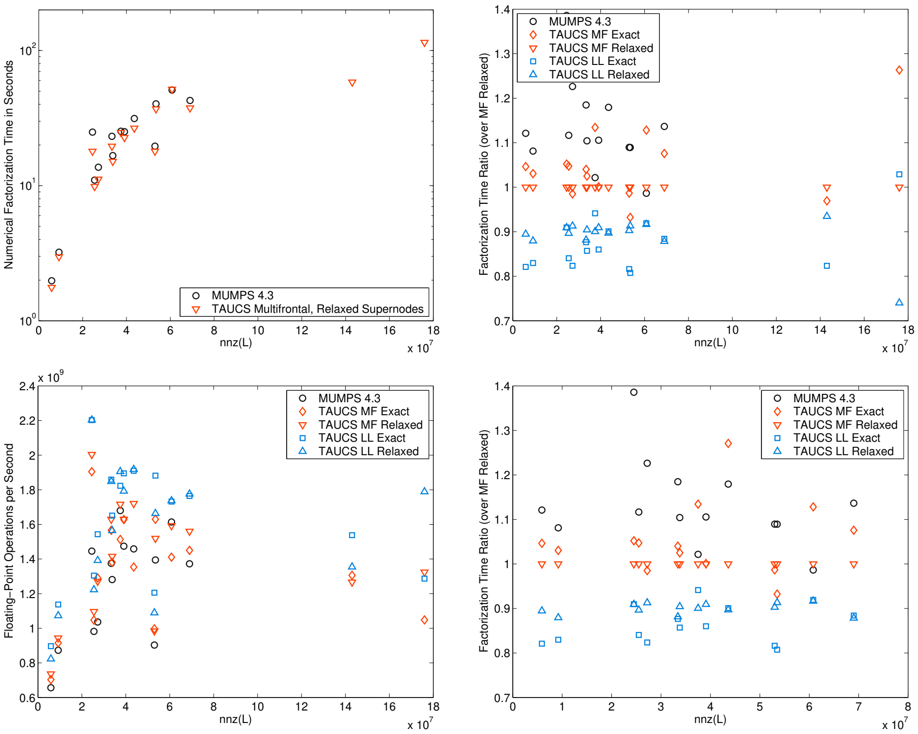

FIG. 7.1. *Numerical factorization times of matrices from the* PARASOL *test-matrix collection on a 2.4 GHz Linux machine.* MUMPS *failed to factor the two largest matrices (the code ran out of memory). The plot on the top left shows the factorization times, in seconds, as a function of the number of nonzeros in the Cholesky factor. The plot shows the factorization times of two codes,* MUMPS *and our multifrontal code with relaxed supernodes (of all our codes, this one is the most similar to* MUMPS*). The plot on the top right shows the factorization times of four of our codes and* MUMPS*. In this plot the factorization times are shown as ratios to the factorization time of our multifrontal code with relaxed supernodes, as a function of the number of nonzeros in the factor. Lower marks indicate better performance. The lower-right plot shows the same data, but excluding the two largest matrices, in order to present more clearly the performance on the smaller matrices. The lower-left plot shows the performance in floating-point operations per second; this plot uses the number of floating point operations that are required to factor the matrices without amalgamating supernodes.*

7.4. Examples of Poor Data Locality in the Left-Looking Algorithm. The next set of experiments demonstrates that on some matrices, the left-looking algorithm can perform significantly poorer than the multifrontal algorithm, due to a higher cache-miss rate. The matrices that we use in these experiments are the matrices analyzed theoretically in Section 4.

Figure 7.2 shows that the left-looking algorithm often performs significantly poorer than the multifrontal algorithm, sometimes by more than a factor of 3. The figure also shows that for every fixed matrix dimension n, the left-looking-to-multifrontal ratio first rises with the supernode size λ, than falls, eventually to a value close to 1. This is not surprising. For very small λ, the cache-miss rate of both algorithms is so high, that both perform very poorly. This is shown clearly in the plot on right, which indicates a low flop/s rate for small λ. For very large λ, most of the work during the numerical factorization step is performed in the context of factoring dense diagonal blocks, not in the context of updates. In such cases, the performance of both algorithms is governed by the performance of a sequence of dense Cholesky factorizations that they perform in exactly the same way. This behavior was not analyzed in Section 4 (we only analyzed rigorously the behavior for specific values of n and

ETNA
Kent State University
etna@mcs.kent.edu

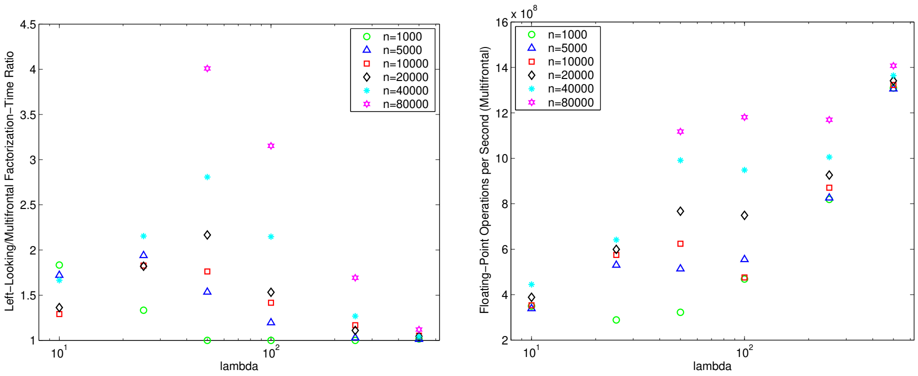

FIG. 7.2. *The performance of the left-looking and the multifrontal algorithms on matrices from the family described in Section 4. The plot on the left shows the ratio between the numerical factorization times of the left-looking algorithm and the multifrontal algorithm. Higher marks mean that the left-looking algorithm is slower. The plot on the right shows the computational rate of the multifrontal algorithm on these matrices. The horizontal axis in both plots corresponds to the width λ of supernodes.*

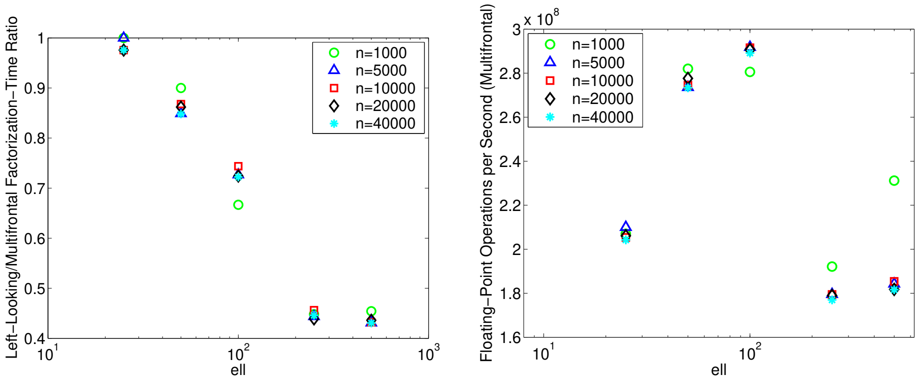

FIG. 7.3. *The performance of the left-looking and the multifrontal algorithms on matrices from the family described in Section 5. The plot on the left shows the ratio between the numerical factorization times of the left-looking algorithm and the multifrontal algorithm. Lower marks (below* 1*) mean that the left-looking algorithm is faster. The plot on the right shows the computational rate of the multifrontal algorithm on these matrices. The horizontal axis in both plots corresponds to the width ℓ of the last supernode. In all cases, the width λ of all the other supernodes is* 5*. Experiments with $\lambda = 1$ and $\lambda = 10$ exhibit similar ratios; the computational rates are generally higher for larger values of λ.*

λ as a function of the cache size), but it is evident from the experiments.

7.5. Examples of Poor Data Locality in the Multifrontal Algorithm. The next set of experiments demonstrates that on some matrices, the multifrontal algorithm can perform significantly poorer than the left-looking algorithm, due to a higher cache-miss rate. The matrices that we use in these experiments are the matrices analyzed theoretically in Section 5.

Figure 7.3 shows that the left-looking algorithm can also perform significantly better than the multifrontal algorithm, sometimes by more than a factor of 2. The figure shows that the performance difference grows with the size ℓ of the last supernode. In experiments with other values of λ, 1 and 10, which are not shown here, we observed that the performance ratios do not vary much with λ, but the overall computational rate rises with λ. This rise is consistent with our analysis of both the left-looking and the multifrontal algorithms.

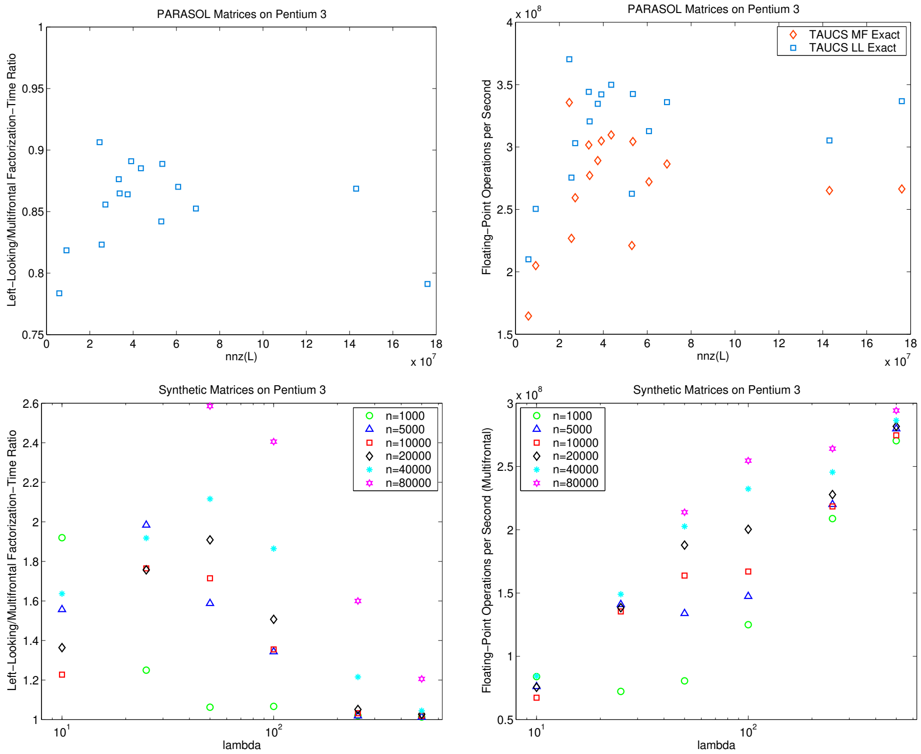

FIG. 7.4. *Numerical factorization performance of matrices from the* PARASOL *test-matrix collection (top) and of synthetic matrices (bottom) on a 0.6 GHz Pentium 3 Linux machine. These results are without supernode amalgamation.*

7.6. Experimental Results with Hardware Performance Counters. Figure 7.4 shows the overall performance of the TAUCS on the Pentium 3 machine. The plots show that qualitatively the behavior of the multifrontal and left-looking algorithms is similar to their behavior on the Pentium 4 machine. The absolute performance is lower, of course. On the PARASOL matrices, the left-looking is roughly 10–20% faster than the multifrontal algorithm. On the synthetic matrices of Section 4, the left-looking is consistently slower, and it runs up to 2.6 times slower. The overall performance of both codes improve with λ, and the worst left-looking/multifrontal ratios are observed for medium values of λ (the peak is at higher λ's for higher n's).

Figure 7.5 presents processor-event information for the PARASOL matrices. The plots show factorization times and five key processor events: the total instructions executed, the number of floating-point instructions executed, the number of accesses to the level-1 data cache, the number of level-1 data-cache misses, and the number of level-2 cache misses. The plot on the left shows ratios of left-looking-to-multifrontal event counts. Values below 1 indicate fewer instructions/accesses/misses in the left-looking algorithm than in the multifrontal algorithm. The plot on the right shows the number of events as a fraction of the total instruction count in the multifrontal factorization. This plot shows mainly the importance of various events. For example, if the number of level-2 cache misses is tiny, it does not matter much whether one algorithm generates more misses than the other.

We also measured the total number of level-2 accesses. That number, which is not shown in the plots, is always slightly higher but very close to the number of level-1 data cache

ETNA
Kent State University
etna@mcs.kent.edu

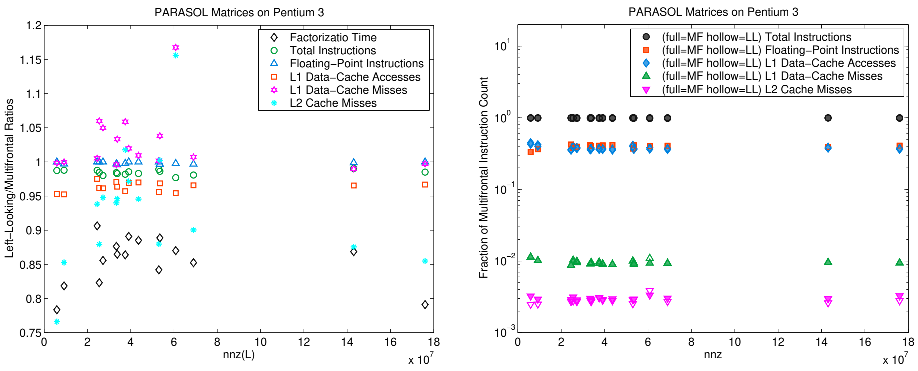

FIG. 7.5. *Results of the hardware-performance-counters measurements on the* PARASOL *matrices on the Pentium 3 machine. The plot on the left shows left-looking/multifrontal ratios of specific measurements, and the plot on the right shows each measurement as a fraction of the instruction count in the multifrontal factorization.*

misses. This implies that the number of instruction-cache misses is insignificant, so most of the level-2 traffic is due to data, not instructions. We have observed the same behavior in all the experiments, not only those on the PARASOL matrices.

The data shows that the left-looking algorithm issues fewer instructions, accesses the level-1 cache fewer times, and generates significantly fewer level-2 cache misses. On the other hand, it often generates more level-1 cache misses. This indicates that the data reuse in registers is better in the left-looking algorithm, hence the reduced instruction count (fewer loads/stores) and the reduced number of level-1 accesses. The facts that the number of level-1 misses is higher in the left-looking algorithm and that the number of level-2 misses is lower probably imply that the exact cache miss rate has little effect on the algorithm. The differences in cache miss counts are usually less than 15% in either direction, and these differences are probably too small to make a difference. It seems that the superior performance of the left-looking algorithm on these matrices is due to lower instruction counts, which is probably a result of the way sparse indexing operations are implemented.

The corresponding results for the synthetic matrices of Section 4 are presented in Figures 7.6 and 7.7. These results show dramatic differences in cache-miss counts between the left-looking and the multifrontal algorithms. The most dramatic differences are in the level-2 miss counts, where the left-looking sometimes generate 12 times more level-2 misses than the multifrontal algorithm. The number of level-1 data-cache misses is sometimes more than 4 times higher in the left-looking algorithm than in the multifrontal algorithm. The most dramatic differences occur at medium λ values, where we also observe the highest differences in factorization times. In some cases the factorization-time ratios peak at the highest level-1 miss ratio (e.g., $n = 10,000$ and $\lambda = 25$) and sometimes at the highest level-2 miss ratio (e.g., $n = 20,000$ and $\lambda = 50$). This implies that misses in both caches have a significant effect on performance. We also observe that the number of cache misses in both algorithms drops with increasing λ. Since performance improves with increasing λ, this again suggests that cache misses are a major determinant of performance on these matrices. At small λ, the left-looking algorithm performs a higher number of instructions (this is shown most clearly on the right plots). This is highly correlated with the higher number of level-1 data accesses, which implies that the increase is due to poor reuse of data in registers.

We did not perform similar experiments on the synthetic matrices of Section 5.

ETNA
Kent State University
etna@mcs.kent.edu

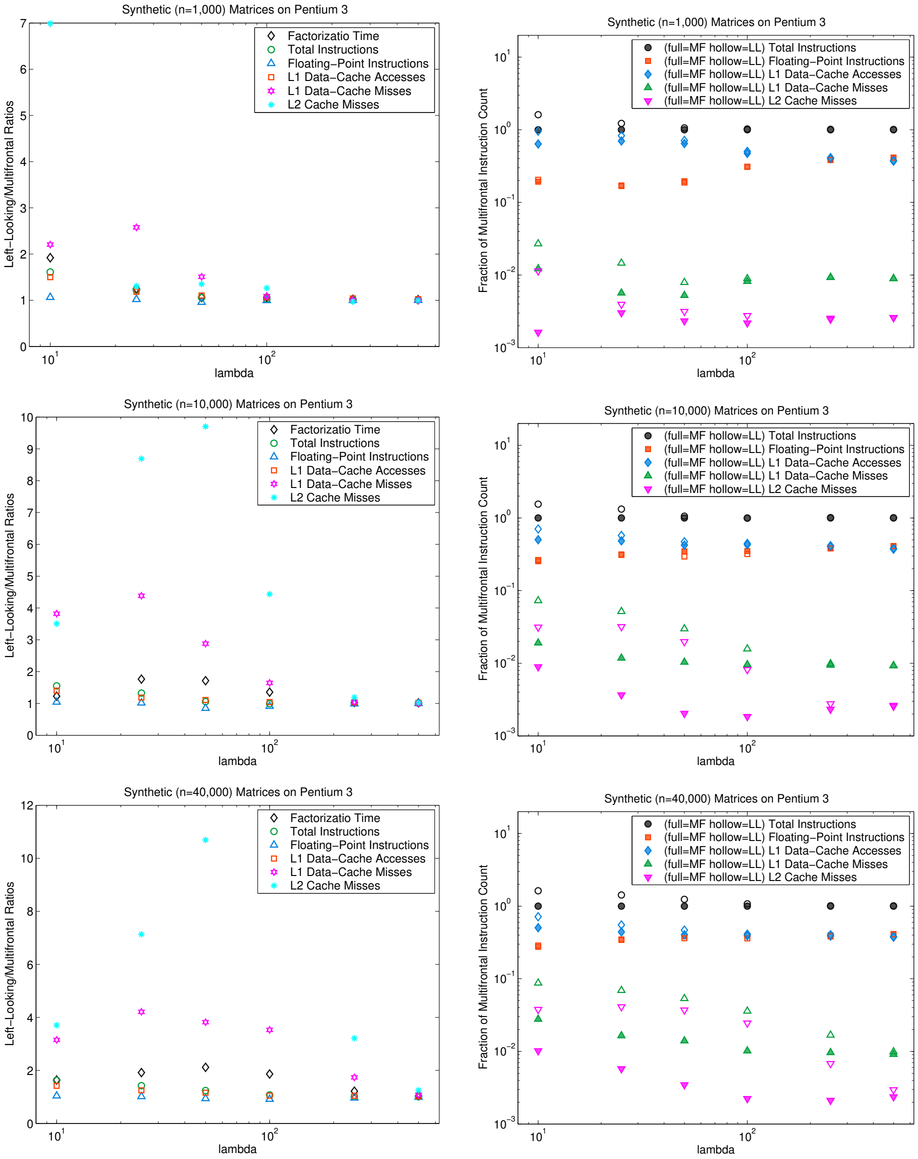

FIG. 7.6. *Results of the hardware-performance-counters measurements on the synthetic matrices ($n = 1000, 10000, 40000$) on the Pentium 3 machine. The plot on the left shows left-looking/multifrontal ratios of specific measurements, and the plot on the right shows each measurement as a fraction of the instruction count in the multifrontal factorization. Figure 7.7 shows similar results for other matrix dimensions.*

8. Discussion and Conclusions. Supernodes were identified in the 1980's as a key ingredient in high-performance sparse-factorization algorithms [5, 8]. Their initial role was to enable vectorization and to reduce indexing overhead in the multifrontal algorithm. In the 1990's, researchers recognized that supernodes can also reduce cache misses in these algorithms, and that supernodes can be exploited not only in multifrontal algorithms, but also in left-looking algorithms [6, 22, 23]. It was implicitly assumed that wider supernodes lead to fewer cache misses.

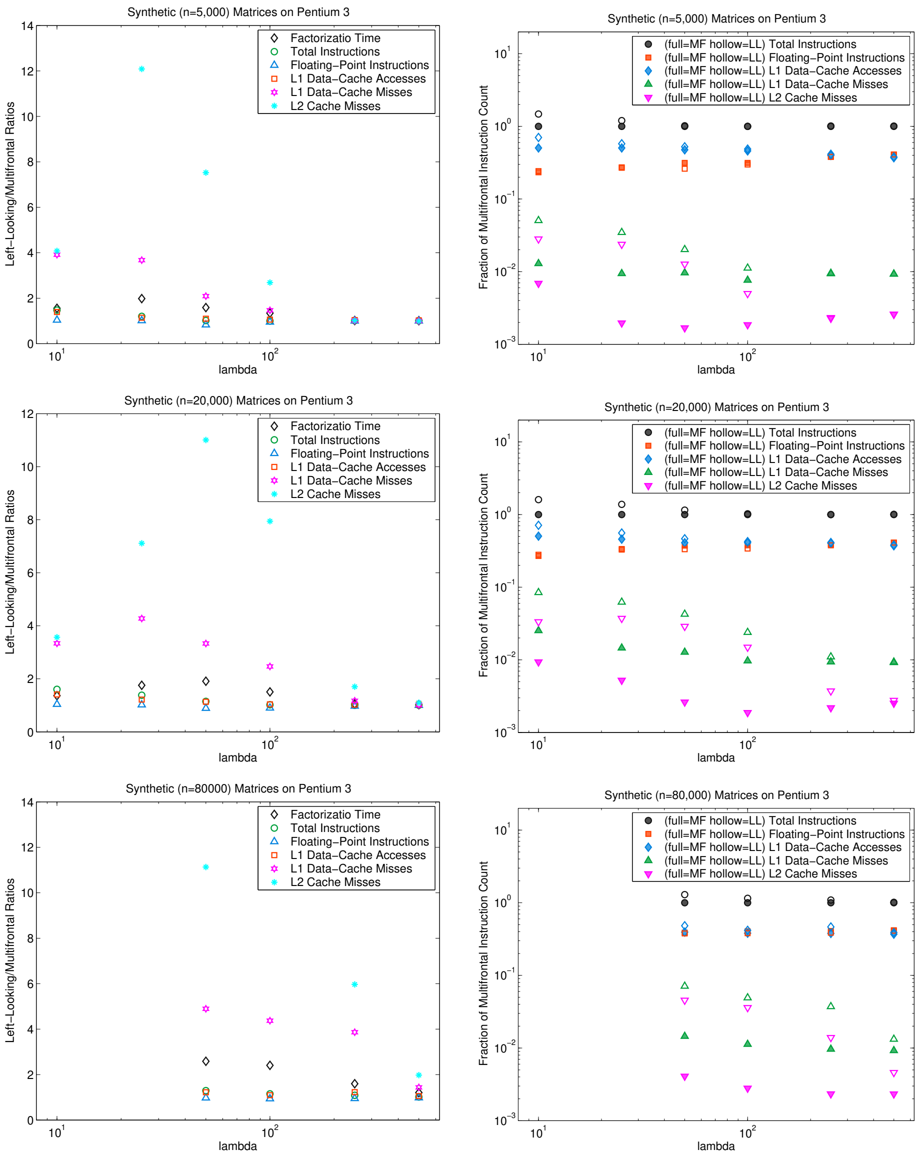

FIG. 7.7. *Results of the hardware-performance-counters measurements on the synthetic matrices ($n = 5000, 20000, 80000$) on the Pentium 3 machine.*

We show in this paper that wider supernodes do, indeed, lead to better cache efficiency in the multifrontal algorithm. But wider supernodes do not automatically lead to high cache efficiency in the left-looking algorithm. In fact, there are matrices where one algorithm exploits the cache effectively while the other does not, and vice versa. The issue, therefore, is more complex than was originally thought.

To provide a concrete recommendation to practitioners, we compared the two algorithms on a set of real-world matrices arising from finite-element analysis. On these matrices, the left-looking is usually faster. These results are consistent with the recommendations of Ng and Peyton from about a decade ago [22].

ETNA
Kent State University
etna@mcs.kent.edu

We also show that on some classes of matrices, the multifrontal algorithm may require asymptotically more storage than the left-looking algorithm. This may cause failure due to lack of memory (or lack of virtual addresses), and this may push the multifrontal algorithm into slower layers of the memory hierarchy.

These two last conclusions suggest that implementors should prefer the left-looking algorithm. However, the fact that the cache efficiency of the multifrontal algorithm depends only on the width of supernodes makes it more predictable. In particular, supernode amalgamation is more predictable under the multifrontal algorithm.

Our theoretical analyses focus only on intra-supernode data reuse in the cache. Inter-supernode data reuse, which occurs on subtrees with small supernodes, also favors the left-looking algorithm, since it does not need a stack of update matrices. This issue, along with sophisticated schedules to maximize inter-supernode data reuse, are discussed in the papers on out-of-core sparse factorization algorithms [11, 24, 25].

Acknowledgements. Thanks to Didi Bar-David for configuring the computers to support PAPI. Thanks to the two anonymous referees for helpful comments and suggestions.

REFERENCES

[1] A. AGGARWAL AND J. S. VITTER, *The input/output complexity of sorting and related problems*, Comm. ACM, 31 (1988), pp. 1116–1127.

[2] P. R. AMESTOY, I. S. DUFF, J. KOSTER, AND J. L'EXCELLENT, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM J. Matrix Anal. Appl., 23 (2001), pp. 15–41.

[3] P. R. AMESTOY, I. S. DUFF, J. L'EXCELLENT, AND J. KOSTER, *MUltifrontal Massively Parallel Solver (MUMPS version 4.3), user's guide*, July 2003, available online at http://www.enseeiht.fr/lima/apo/MUMPS/doc.html.

[4] P. R. AMESTOY, I. S. DUFF, AND J.-Y. L'EXCELLENT, *Multifrontal parallel distributed symmetric and unsymmetric solvers*, Comput. Methods Appl. Mech. Engrg., 184 (2000), pp. 501-520.

[5] C. ASHCRAFT AND R. GRIMES, *The influence of relaxed supernode partitions on the multifrontal method*, ACM Trans. Math. Software, 15 (1989), pp. 291–309.

[6] 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. Matrix Anal. Appl., 20 (1999), pp. 720–755.

[7] I. S. DUFF, A. M. ERISMAN, AND J. K. REID, *Direct Methods for Sparse Matrices*, Oxford University Press, 1986.

[8] I. S. DUFF AND J. K. REID, *The multifrontal solution of indefinite sparse symmetric linear systems*, ACM Trans. Math. Software, 9 (1983), pp. 302–325.

[9] P. C. FISCHER AND R. L. PROBERT, *A note on matrix multiplication in a paging environment*, in ACM '76: Proceedings of the Annual Conference, 1976, pp. 17–21.

[10] A. GEORGE AND J. W. H. LIU, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[11] J. R. GILBERT AND S. TOLEDO, *High-performance out-of-core sparse LU factorization*, in Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing, San-Antonio, Texas, 1999, 10 pages on CDROM.

[12] G. H. GOLUB AND C. F. V. LOAN, *Matrix Computations*, Johns Hopkins University Press, 3rd ed., 1996.

[13] J.-W. HONG AND H. T. KUNG, *I/O complexity: the red-blue pebble game.*, in Proceedings of the 13th Annual ACM Symposium on Theory of Computing, 1981, pp. 326–333.

[14] G. KARYPIS AND V. KUMAR, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.

[15] A. LAMARCA AND R. E. LADNER, *The influence of caches on the performance sorting*, J. Algorithms, 31 (1999), pp. 66–104.

[16] J. W. H. LIU, *On the storage requirement in the out-of-core multifrontal method for sparse factorization*, ACM Trans. Math. Software, 12 (1986), pp. 249–264.

[17] ———, *The multifrontal method and paging in sparse Cholesky factorization*, ACM Trans. Math. Software, 15 (1989), pp. 310–325.

[18] J. W. H. LIU, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.

[19] J. W. H. LIU, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM Rev., 34 (1992), pp. 82–109.

[20] J. W. H. LIU, E. G. NG, AND B. W. PEYTON, *On finding supernodes for sparse matrix computations*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 242–252.
[21] A. C. MCKELLER AND E. G. COFFMAN, JR., *Organizing matrices and matrix operations for paged memory systems*, Comm. ACM, 12 (1969), pp. 153–165.
[22] E. G. NG AND B. W. PEYTON, *Block sparse Cholesky algorithms on advanced uniprocessor computers*, SIAM J. Sci. Comput., 14 (1993), pp. 1034–1056.
[23] E. ROTHBERG AND A. GUPTA, *Efficient sparse matrix factorization on high-performance workstations—exploiting the memory hierarchy*, ACM Trans. Math. Software, 17 (1991), pp. 313–334.
[24] E. ROTHBERG AND R. SCHREIBER, *Efficient methods for out-of-core sparse cholesky factorization*, SIAM J. Sci. Comput., 21 (1999), pp. 129–144.
[25] V. ROTKIN AND S. TOLEDO, *The design and implementation of a new out-of-core sparse Cholesky factorization method*, ACM Trans. Math. Software, 30 (2004), pp. 19–46.
[26] R. SCHREIBER, *A new implementation of sparse gaussian elimination*, ACM Trans. Math. Software, 8 (1982), pp. 256–276.
[27] S. SEN, S. CHATTERJEE, AND N. DUMIR, *Towards a theory of cache-efficient algorithms*, J. ACM, 49 (2002), pp. 828–858.
[28] G. W. STEWART, *Matrix Algorithms, Volume 1: Basic Decompositions*, SIAM, 1998.
[29] S. TOLEDO, *Locality of reference in LU decomposition with partial pivoting*, SIAM J. Matrix Anal. Appl., 18 (1997), pp. 1065–1081.
[30] ———, *A survey of out-of-core algorithms in numerical linear algebra*, in External Memory Algorithms, J. M. Abello and J. S. Vitter, eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1999, pp. 161–179.
[31] R. C. WHALEY AND J. J. DONGARRA, *Automatically tuned linear algebra software*, tech. report, Computer Science Department, University Of Tennessee, 1998, available online at http://www.netlib.org/atlas.