SIAM J. SCI. COMPUT.
Vol. 31, No. 6, pp. 4774–4794
© 2010 Society for Industrial and Applied Mathematics

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

REDUCING THE I/O VOLUME IN SPARSE OUT-OF-CORE MULTIFRONTAL METHODS*

EMMANUEL AGULLO†, ABDOU GUERMOUCHE‡, AND JEAN-YVES L'EXCELLENT†

Abstract. Sparse direct solvers, and in particular multifrontal methods, are methods of choice to solve the large sparse systems of linear equations arising in certain simulation problems. However, they require a large amount of memory (e.g., in comparison to iterative methods). In this context, out-of-core solvers may be employed: disks are used when the required storage exceeds the available physical memory. In this paper, we show how to process the task dependency graph of multifrontal methods in a way that minimizes the input/output (I/O) requirements. From a theoretical point of view, we show that minimizing the storage requirement can lead to a huge volume of I/O compared to directly minimizing the I/O volume. Then experiments on large real-world problems also show that applying standard algorithms to minimize the storage is not always efficient at reducing the volume of I/O and that significant gains can be obtained with the use of our algorithms to minimize I/O. We finally show that efficient memory management algorithms can be applied to all the variants proposed.

Key words. sparse matrices, direct methods, out-of-core, memory, input/output volume, tree traversal, postorder

AMS subject classifications. 65F05, 65F50, 05C05, 05C50, 68R05

DOI. 10.1137/080720061

1. Introduction.

We are interested in solving sparse systems of linear equations of the form $Ax = b$ by a direct method. Such methods work in three phases: (i) an analysis phase that orders the variables of the problem to limit the computations and prepares the work for the factorization; (ii) a numerical factorization phase, where A is factored using an LU, LL^t, or LDL^t decomposition; and (iii) a solve phase, where triangular factors are used to obtain the solution of the problem. Because of their large memory requirements, several authors have worked on out-of-core sparse direct solvers [24, 2, 7, 15, 20, 21, 22]. Left-looking and multifrontal methods are two main classes of sparse direct methods that can be extended to an out-of-core context. Left-looking approaches significantly reduce the minimal memory requirements. Multifrontal methods may lead to large frontal matrices that prevent processing arbitrarily large problems [21] if those frontal matrices are not assembled and factored with out-of-core algorithms. On the other hand, on problems for which the largest frontal matrix fits in memory, the multifrontal method remains interesting [8, 18] and motivates the design of robust software solutions [2, 20].

In this paper, we consider matrices with a symmetric structure, or approaches like [5] when the structure of the matrix is unsymmetric. In such cases, the multifrontal method uses an elimination tree [19], which is a transitive reduction of the matrix

*Received by the editors April 2, 2008; accepted for publication (in revised form) October 19, 2009; published electronically February 3, 2010. Partially supported by ANR project SOLSTICE, ANR-06-CIS6-010. A (shorter) preliminary version of this contribution appeared in *Proceedings of the HiPC'07 14th International Conference On High Performance Computing*, Goa, India, 2007, Lecture Notes in Comput. Sci. 4873, Springer, New York, 2007, pp. 47–58.

http://www.siam.org/journals/sisc/31-6/72006.html

†INRIA/LIP-ENS Lyon Laboratoire de l'Informatique du Parallélisme (UMR CNRS-ENS Lyon-INRIA-UCBL), ENS Lyon, 46 allée d'Italie, 69354 Lyon cedex, France (Emmanuel.Agullo@inria.fr, Jean-Yves.L.Excellent@ens-lyon.fr).

‡Laboratoire Bordelais de Recheche en Informatique (UMR 5800)–351, cours de la Liberation F-33405 Talence cedex, France (Abdou.Guermouche@labri.fr).

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

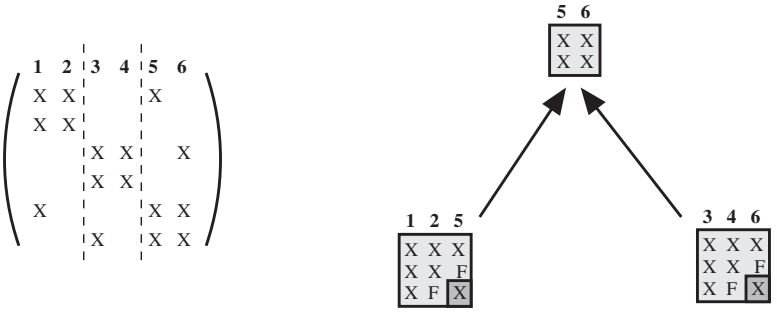


FIG. 1.1. *A matrix and its associated assembly tree when variables are eliminated in natural order. Light and dark grey parts of the frontal matrices represent the fully summed and contribution blocks, respectively.*

foreach *node* ν **in** *the tree (postorder traversal)* **do**
  $al_{\mathcal{N}}$: Allocate memory for the frontal matrix associated with ν;
  if ν *is not a leaf* **then**
    $as_{\mathcal{N}}$: Assemble contribution blocks from children;
  $f_{\mathcal{N}}$: Perform a partial factorization of the frontal matrix of ν, writing factors to disk on the fly;
  keep the contribution block for later use;

ALGORITHM 1.1. *Multifrontal method with factors on disk.*

graph and is the smallest data structure representing dependencies between the operations. In practice, we use a structure called *assembly tree*, obtained by merging nodes of the elimination tree whose corresponding columns belong to the same supernode [6].

A supernode is a contiguous range of columns (in the factor matrix) having similar structure. The multifrontal factorization of a sparse matrix A is done by a succession of partial factorizations of small dense matrices called *frontal matrices*, which are associated to the nodes of the tree. Figure 1.1 gives an idea of how such a tree of frontal matrices can be associated to a matrix (see [11, 12] for more precise information on this process and on the multifrontal method in general). Each frontal matrix is divided into two parts: the *fully summed* block, which corresponds to the part factored during the elimination process, and the *contribution block* (or Schur complement), which corresponds to the block updated while factoring the fully summed block. Since the factors are terminal data (not reused before the solution step) for the factorization phase, it appears natural to write them to disk as soon as they are produced.

Once the factorization of the fully summed block is done, the contribution block is passed to the parent node and when the contribution blocks from all children are available on the parent, those can be assembled (i.e., summed with the values contained in the frontal matrix of the parent). Focusing on memory handling issues, such a multifrontal method may be presented as in Algorithm 1.1, where an assembly step (line $as_{\mathcal{N}}$ of the algorithm) always requires the frontal matrix of the parent to be in memory. An in-core assembly also requires all the contribution blocks from the children to be in memory, whereas contribution blocks can be partially on disk during an out-of-core assembly operation.

Until section 7.3, let us assume that there are no numerical difficulties requiring us to delay the factorization of pivot rows/columns from the fully summed block of

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

a node to its ancestors [11]. Delayed pivots are avoided when processing symmetric positive definite matrices or when static pivoting is enabled (see [9] for a discussion of static pivoting in the multifrontal approach).

We consider several minor variants of the multifrontal algorithm. We call *last-in-place* a variant of the assembly scheme (available, for example, in a code like `MA27` [10]), where the memory of the frontal matrix at the parent node is allowed to overlap with the contribution block of the last child. In that case, we save space by not summing the memory of the contribution block of the child with the memory of the frontal matrix of the parent. We also propose a new variant, where we overlap the memory for the frontal matrix of the parent with the memory of the child having the largest contribution block (even if that child is not processed last).

By relying on a postorder traversal, the multifrontal algorithm can use a stack mechanism to store the contribution blocks: the contribution blocks produced last are the first ones assembled. Still, there is a lot of freedom to order the siblings at each level of the tree so that the tree traversal can have a significant impact on both the number of contribution blocks stored simultaneously and the memory usage. Liu [17] and Guermouche, L'Excellent, and Utard [14, 13]) have explored the impact of the tree traversal on the memory behavior and proposed tree traversals that minimize the storage requirements when factors are systematically written to disk. With this last assumption, Liu suggested in the conclusion of [17] that minimizing the storage requirements was well adapted to an out-of-core execution.

In this paper we focus on the volume of input/output (I/O) related to the stack of contribution blocks and describe postorders of the tree that minimize the volume of I/O. By expressing this volume in a formal way, we show that minimizing the storage requirements is different from minimizing the volume of I/O. For each variant of the multifrontal algorithm, we present a postorder that minimizes memory, so-called `MinMEM` algorithm; then, we describe a new algorithm called `MinIO` that, depending on the physical memory available, minimizes the I/O volume. On real-life problems, we show that our algorithms significantly reduce the volume of I/O compared to approaches focusing on the storage requirements (such as [17]). We finally show that simple stack mechanisms can be used to implement the corresponding memory management algorithms.

The paper is organized as follows. In sections 2 and 3, we explain how to model and minimize the volume of I/O induced by the *classical* and *last-in-place* schemes, respectively. In section 4, we discuss the new variant of the *in-place* algorithm. We then show in section 5 that the volume of I/O induced by `MinMEM` may be arbitrarily larger than the volume induced by `MinIO`. Section 6 illustrates the difference between `MinMEM` and `MinIO` on matrices arising from real-life problems and shows the utility of the new *in-place* variant we proposed. In section 7, we discuss the memory management of the proposed schemes.

2. Limiting the amount of I/O. Before discussing the volume of I/O, we introduce some general notations. In a limited memory environment, we define M_0 as the amount of core memory available for the factorization. As described in the introduction, the multifrontal method is based on a tree in which a parent node is allocated in memory after all its child subtrees have been processed. When considering a generic parent node and its n children numbered $j = 1, \ldots, n$, we note

- cb_j, the storage for the contribution block passed from child j to the parent;
- m / m_j, the storage of the frontal matrix associated with the parent node / with its jth child (note that $m \geq cb_j$ and $m_j \geq cb_j$);

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

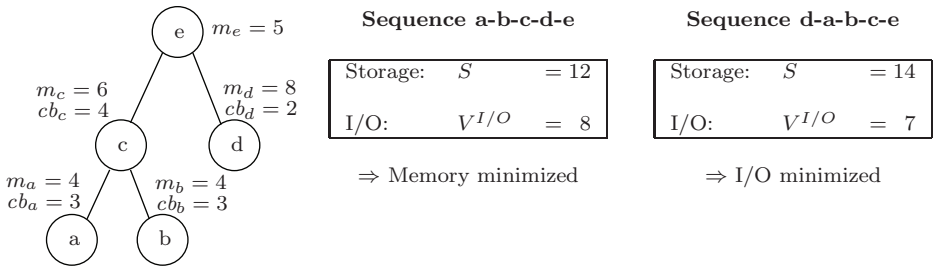


FIG. 2.1. *Influence of the postorder on the storage requirement and on the volume of I/O (with $M_0 = 8$).*

- S / S_j, the storage required to process the subtree rooted at the parent / at child j (if $S_j \leq M_0$, no I/O is necessary to process the subtree rooted at j);
- $V^{I/O}$ / $V_j^{I/O}$ the volume of I/O required to process the subtree rooted at the parent / at child j given an available memory of size M_0.

Any convenient unit can be used for the above quantities, such as bytes, gigabytes (GB), or number of scalar entries. It is important to note that every tree whose corresponding nodes respect the constraints above can be associated to a matrix: one can build the structure of a frontal matrix associated to each node, and from the structure of each frontal matrix, it is easy to find a corresponding initial matrix.

2.1. Illustrative example. To illustrate the memory behavior, let us take the small example described in Figure 2.1(left): we consider a root node (e) with two children (c) and (d). The frontal matrix of (e) requires a storage $m_e = 5$ (let us assume, for example, that this means 5 GB). The contribution blocks of (c) and (d) require a storage $cb_c = 4$ and $cb_d = 2$, while the storage requirements for their frontal matrices are $m_c = 6$ and $m_d = 8$, respectively. (c) has itself two children (a) and (b) with characteristics $cb_a = cb_b = 3$ and $m_a = m_b = 4$. We assume that the core memory available is $M_0 = 8$.

To respect a postorder traversal, there are two possible ways to process this tree: (a-b-c-d-e) and (d-a-b-c-e). (Note that (a) and (b) are identical and can be swapped.) We now describe the memory behavior and I/O operations in each case. We first consider the postorder (a-b-c-d-e). (a) is first allocated ($m_a = 4$) and factored (we write its factors of size $m_a - cb_a = 1$ to disk), and $cb_a = 3$ remains in memory. After (b) is processed, the memory contains $cb_a + cb_b = 6$. A peak of storage $S_c = 12$ is then reached when the frontal matrix of (c) is allocated (because $m_c = 6$). Since only 8 (GB) can be kept in core memory, this forces us to write to disk a volume of data equal to 4 GB. Thanks to the postorder and the use of a stack, these 4 GB are the ones that will be reaccessed last; they correspond to the bottom of the stack. During the assembly process we first assemble contributions that are in memory, and then read 4 GB from disk to assemble them in turn in the frontal matrix of (c). Note that (here but also more generally), in order to fit the memory requirements, the assembly of data residing on disk may have to be performed by panels (interleaving the read and assembly operations). After the factors of (c) of size $m_c - cb_c = 2$ are written to disk, its contribution block $cb_c = 4$ remains in memory. When the leaf node (d) is processed, the peak of storage reaches $cb_c + m_d = 12$. This leads to a new volume of I/O equal to 4 (and corresponding to cb_c). After (d) is factored, the storage requirement is equal to $cb_c + cb_d = 6$, among which only $cb_d = 2$ is in core (cb_c is already on disk). Finally, the frontal matrix of the parent (of size $m_e = 5$) is allocated,

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

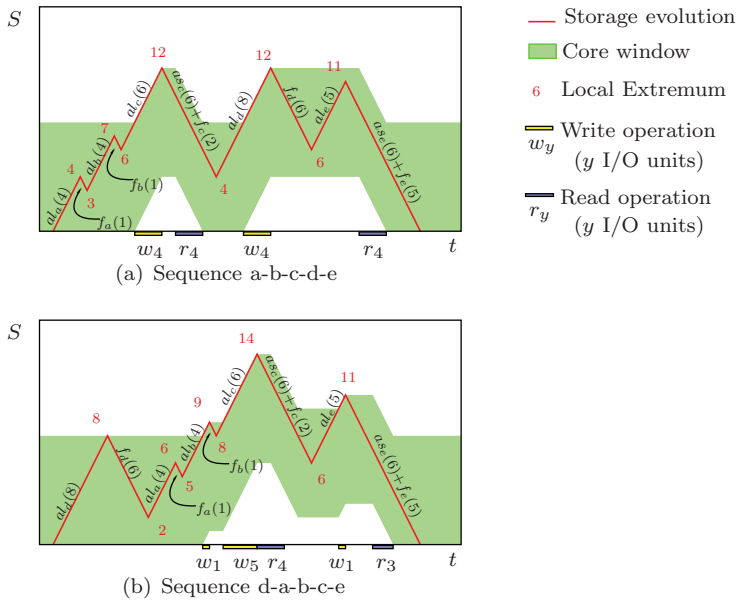


FIG. 2.2. *Evolution of the storage requirement S when processing the sample tree of Figure* 2.1 *with the two possible postorders and subsequent I/O operations. Notations al_N, as_N, and f_N were introduced in Algorithm* 1.1.

leading to a storage $cb_c + cb_d + m_e = 11$: after cb_d is assembled in core (into the frontal matrix of the parent), cb_c is read back from disk and assembled in turn. Overall the volume of data written to (and read from) disk[1] is $V_e^{I/O}(a-b-c-d-e) = 8$, and the peak of storage was $S_e(a-b-c-d-e) = 12$.

When the tree is processed in order (d-a-b-c-e) (see Figure 2.2(b)), the storage requirement successively takes the values $m_d = 8$, $cb_d = 2$, $cb_d + m_a = 6$, $cb_d + cb_a = 5$, $cb_d + cb_a + m_b = 9$, $cb_d + cb_a + cb_b = 8$, $cb_d + cb_a + cb_b + m_c = 14$, $cb_d + cb_c = 6$, $cb_d + cb_c + m_e = 11$, with a peak $S_e(d-a-b-c-e) = 14$. Nodes (d) and (a) can be processed without inducing I/O; then 1 unit of I/O is done when allocating (b), 5 units when allocating (c), and finally 1 unit when the frontal matrix of the root node is allocated. We obtain $V_e^{I/O}(d-a-b-c-e) = 7$.

We observe that the postorder (a-b-c-d-e) minimizes the peak of storage and that (d-a-b-c-e) minimizes the volume of I/O. This shows that minimizing the peak of storage is different from minimizing the volume of I/O.

All the process described above is illustrated in Figure 2.2, which represents the evolution of the storage in time for the two postorders (a-b-c-d-e) and (d-a-b-c-e) (subfigures 2.2(a) and 2.2(b), respectively). The storage increases when memory is allocated for a new frontal matrix of size x ($al_N(x)$); it decreases when contribution blocks of size y are assembled into the frontal matrix of their parent ($as_N(y)$) and when factors of size z are written to disk ($f_N(z)$). When the storage is larger than the available memory M_0, this means that part of the stack is on disk. The core window is shaded in the figure so that the white area below the core window corresponds to the volume of data on disk. Finally write and read operations on the stack are noted

[1] We do not count I/O for factors, which are independent from the postorder chosen: factors are systematically written to disk in all variants considered.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

w_x and r_y, where x and y are written and read sizes, respectively. We can see that each time the storage is about to exceed the upper bound of the core window, a write operation is necessary. The volume of data read from disk depends on the size of the contribution blocks residing on disk that need to be assembled.

2.2. Expressing the volume of I/O. Since contribution blocks are stored using a stack mechanism, some contribution blocks (or parts of contribution blocks) may be kept in memory and consumed without being written to disk. Assuming that the contribution blocks are written only when needed (possibly only partially), that factors are written to disk as soon as they are computed, and that a frontal matrix must fit in core memory, we focus on the computation of the volume of I/O on this stack of contribution blocks.

When processing a child j, the contribution blocks of all previously processed children have to be stored. Their memory size sums up with the storage requirements S_j of the considered child, leading to a global storage equal to $S_j + \sum_{k=1}^{j-1} cb_k$. After all the children have been processed, the frontal matrix (of size m) of the parent is allocated, requiring a storage equal to $m + \sum_{k=1}^{n} cb_k$. Therefore, the storage required to process the complete subtree rooted at the parent node is given by the maximum of all theses values, that is,

$$S = \max\left(\max_{j=1,n}\left(S_j + \sum_{k=1}^{j-1} cb_k\right), m + \sum_{k=1}^{n} cb_k\right). \tag{2.1}$$

Knowing that the storage requirement S for a leaf node is equal to the size of its frontal matrix m and applying this formula recursively (as done in [17]) allows us to determine the storage requirement for the complete tree.

In our out-of-core context, we now assume that we are given a core memory of size M_0. If $S > M_0$, some I/O will be necessary. The data that must be written to disk are given by Property 1, which we have already used in a nonformal way in the example of section 2.1.

PROPERTY 1. *For a given postorder of the tree, the contribution blocks are accessed with a stack mechanism. When some I/O is necessary, the bottom of the stack should be written first because it will be reaccessed last. This results in an optimal volume of I/O.*

To simplify the discussion we first consider a set of subtrees and their parent, and suppose that $S_j \leq M_0$ for all children j. The volume of contribution blocks that will be written to disk corresponds to the difference between the memory requirement at the moment when the peak S is obtained and the size M_0 of the memory allowed (or available). Indeed, each time an I/O is done, an amount of temporary data located at the bottom of the stack is written to disk. Furthermore, data will only be reused (read from disk) when assembling the parent node. More formally, the expression of the volume of I/O $V^{I/O}$, using formula (2.1) for the storage requirement, is

$$V^{I/O} = \max\left(0, \max\left(\max_{j=1,n}\left(S_j + \sum_{k=1}^{j-1} cb_k\right), m + \sum_{k=1}^{n} cb_k\right) - M_0\right). \tag{2.2}$$

As each contribution written is read once, $V^{I/O}$ will refer to the volume of data written.

We now suppose that there exists a child j such that $S_j > M_0$. We know that the subtree rooted at child j will have an intrinsic volume of I/O $V_j^{I/O}$ (recursive

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

definition based on a bottom-up traversal of the tree). Furthermore, we know that the memory for the subtree rooted at child j cannot exceed the physical memory M_0. Thus, we will consider that it uses a memory exactly equal to M_0 ($A_j \stackrel{def}{=} \min(S_j, M_0)$) and that it induces an intrinsic volume of I/O equal to $V_j^{I/O}$. With this definition of A_j as the *active memory*, i.e., the amount of core memory effectively used to process the subtree rooted at child j, we can now generalize formula (2.2). We obtain

(2.3)
$$V^{I/O} = \max\left(0, \max\left(\max_{j=1,n}\left(A_j + \sum_{k=1}^{j-1} cb_k\right), m + \sum_{k=1}^{n} cb_k\right) - M_0\right) + \sum_{j=1}^{n} V_j^{I/O}.$$

To compute the volume of I/O on the complete tree, we recursively apply formula (2.3) at each level (knowing that $V^{I/O} = 0$ for leaf nodes). The volume of I/O for the factorization is then given by the value of $V^{I/O}$ at the root.

2.3. Tree traversals. It results from formula (2.3) that minimizing the volume of I/O is equivalent to minimizing the expression $\max_{j=1,n}(A_j + \sum_{k=1}^{j-1} cb_k)$, since it is the only term sensitive to the order of the children.

THEOREM 2.1 (Liu [17, Theorem 3.2]). *Given a set of values $(x_i, y_i)_{i=1,\dots,n}$, the minimal value of $\max_{i=1,\dots,n}(x_i + \sum_{j=1}^{i-1} y_j)$ is obtained by sorting the sequence (x_i, y_i) in decreasing order of $x_i - y_i$, that is, $x_1 - y_1 \geq x_2 - y_2 \geq \cdots \geq x_n - y_n$.*

Thanks to Theorem 2.1 (proved in [17]), we deduce that we should process the children nodes in decreasing order of $A_j - cb_j = \min(S_j, M_0) - cb_j$. (This implies that if all subtrees require a storage $S_j > M_0$, then `MinIO` will simply order them in increasing order of cb_j.) An optimal postorder traversal of the tree is then obtained by applying this sorting at each level of the tree, constructing formulas (2.1) and (2.3) from bottom to top. We will name `MinIO` this algorithm.

Note that, in order to minimize the peak of storage (defined in formula (2.1)), children had to be sorted (at each level of the tree) in decreasing order of $S_j - cb_j$ rather than $A_j - cb_j$. Therefore, on the example from section 2.1, the subtree rooted at (c) ($S_c - cb_c = 12 - 4 = 8$) had to be processed before the subtree rooted at (d) ($S_d - cb_d = 8 - 2 = 6$). The corresponding algorithm (that we name `MinMEM` and that leads to the postorder (a-b-c-d-e)) is different from `MinIO` (that leads to (d-a-b-c-e)): minimizing the storage requirement is thus different from minimizing the I/O volume; it may induce a volume of I/O larger than needed. Conversely, when the stack fits in core memory, M_0 is larger than S_j and $A_j = S_j$ for all j. In that case, `MinMEM` and `MinIO` lead to the same tree traversal and to the same peak of core memory.

3. In-place assembly of the last contribution block. In this variant (used in `MA27` [10] and its successors, for example) of the *classical* multifrontal algorithm, the memory of the frontal matrix of the parent is allowed to overlap with (or to include) that of the contribution block from the last child. The contribution block from the last child is then expanded (or assembled *in-place*) in the memory of the parent. Since the memory of a contribution block can be large, this scheme can have a strong impact on both storage and I/O requirements. In this new context, the storage requirements needed to process a given node (formula (2.1)) becomes

(3.1)
$$S = \max\left(\max_{j=1,n}\left(S_j + \sum_{k=1}^{j-1} cb_k\right), m + \sum_{k=1}^{\boxed{n\text{-}1}} cb_k\right).$$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

The only difference with formula (2.1) comes from the *in-place* assembly of the last child (see the boxed superscript in the sum in formula (3.1)). In the rest of the paper we will use the term *last-in-place* to denote the memory management scheme where an *in-place* assembly scheme is used for the contribution block coming from the last child. Liu has shown [17] that formula (3.1) could be minimized by ordering children in decreasing order of $\max(S_j, m) - cb_j$.

In an out-of-core context, the use of this *in-place* scheme induces a modification of the amount of data that has to be written to/read from disk. As previously for the memory requirement, the volume of I/O to process a given node with n children (formula (2.3)) becomes

$$V^{I/O} = \max\left(0, \max\left(\max_{j=1,n}\left(A_j + \sum_{k=1}^{j-1} cb_k\right), m + \sum_{k=1}^{\boxed{n-1}} cb_k\right) - M_0\right) + \sum_{j=1}^{n} V_j^{I/O}.$$

Once again, the difference comes from the *in-place* assembly of the contribution block coming from the last child. Because $m + \sum_{k=1}^{n-1} cb_k = \max_{j=1,n}(m + \sum_{k=1}^{j-1} cb_k)$, this formula can be rewritten as

$$V^{I/O} = \max\left(0, \max_{j=1,n}\left(\max(A_j, m) + \sum_{k=1}^{j-1} cb_k\right) - M_0\right) + \sum_{j=1}^{n} V_j^{I/O}. \tag{3.2}$$

Thanks to Theorem 2.1, minimizing the above quantity can be done by sorting the children nodes in decreasing order of $\max(A_j, m) - cb_j$ at each level of the tree.

4. In-place assembly of the largest contribution block. In order to further reduce the storage requirement (in comparison to (3.1)), we introduce in this section a new scheme that aims at overlapping the memory of the parent with the memory of the *largest* child contribution block. Compared to (2.1) corresponding to the *classical* scheme, cb_{max} must be subtracted from the term $m + \sum_j cb_j$. Since cb_{max} is a constant that does not depend on the order of children, minimizing the storage (`MinMEM`) is done by using the same tree traversal as for the classical scheme (decreasing order of $S_j - cb_j$). We call this new scheme *max-in-place*, as it constitutes a natural extension to the *in-place* assembly scheme from the previous section. We will see how the memory management can be adapted in section 7.

In an out-of-core context, it is not immediate or easy to generalize `MinIO` to the in-place assembly of the largest contribution block. The problem comes from the fact that the largest contribution block, if it does not correspond to the last child, may have to be written to disk to leave space for the subtrees that come after it in the postorder. Let us illustrate the difficulties one may encounter on the example provided in Figure 4.1. We first remark that the optimal order for the `MinIO` + *last-in-place* variant gives a sequence of children nodes (a-b-c), to which corresponds a volume of I/O equal to 5 (see section 4). Let us now consider the *max-in-place* case. Assuming for the moment that the order is still (a-b-c), we process child (a) and child (b) without performing I/O. In order to allocate the memory for $m_c = 10$, at least 5 units of data have to be written to disk among cb_a and cb_b; for example, one may write all of cb_b and 3 units of data from cb_a. We process (c) and have in memory $cb_c = 4$ together with two units of data of cb_a. Assembling the largest contribution cb_a in-place then requires reading back the 3 units of data from cb_a from disk and writing 1 unit of data from cb_c to disk to make space for the frontal matrix of node (d), of

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

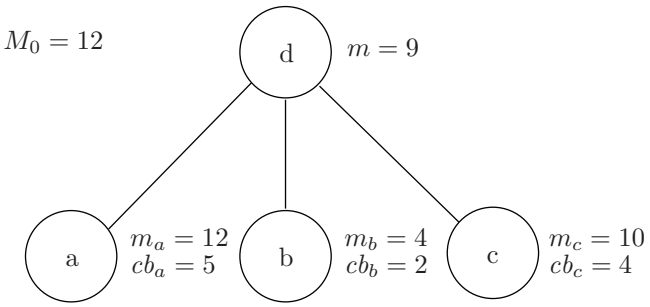


FIG. 4.1. *Example of a tree where* MinIO + last-in-place *is better than the* max-in-place *variant.*

size $m = 9$. This is far less natural, and it requires overall more I/O than an *in-place* assembly of the contribution block of the last child (which is in memory). By trying all other possible orders (a-c-b), (b-a-c), (b-c-a), (c-a-b), (c-b-a), we can observe with this example that it is not possible to obtain a volume of I/O with a *max-in-place* assembly smaller than the one we obtained with a *last-in-place* assembly (equal to 5). Thus, the *max-in-place* strategy in an out-of-core context appears complicated and nonoptimal, at least in some cases. Therefore, we propose to only apply the *max-in-place* strategy on parts of the tree that can be processed in-core. This is done in the following way: we first apply MinMEM + *max-in-place* in a bottom-up process to the tree. As long as this leads to a storage smaller than M_0, we keep this approach to reduce the intrinsic in-core memory requirements. Otherwise, we *switch* to MinIO + *last-in-place* to process the current family and any parent family. In the following we name MinIO + *max-in-place* the resulting heuristic.

5. Theoretical comparison of MinMEM and MinIO.

THEOREM 5.1. *The volume of I/O induced by* MinMEM *(or any memory-minimization algorithm) may be arbitrarily larger than the volume induced by* MinIO.

Proof. In the following, we provide a formal proof for the *classical* and *last-in-place* assembly schemes, but it also applies to the strategies defined in section 4 for the *max-in-place* scheme (which is identical to *last-in-place* on families where I/O are needed). Let M_0 be the core memory available and $\alpha(> 2)$ an arbitrarily large real number. We aim at building an assembly tree (to which we may associate a matrix; see the beginning of section 2) for which

- S (MinIO) $> S$ (MinMEM) and
- the I/O volume induced by MinMEM (or any memory-minimization algorithm), $V^{I/O}$ (MinMEM), is at least α times larger than the one induced by MinIO, $V^{I/O}$ (MinIO)—i.e., $V^{I/O}(\texttt{MinMEM})/V^{I/O}(\texttt{MinIO}) \geq \alpha$.

We first consider the sample tree T_0 of Figure 5.1(a). It is composed of a root node (r) and three leaves (a), (b), and (c). The frontal matrices of (a), (b), (c), and (r), respectively, require a storage $m_a = m_b = m_c = M_0$ and $m_r = M_0/2$. Their respective contribution blocks are of size $cb_a = cb_b = cb_c = M_0/2$ and $cb_r = M_0/3$. Both for the *classical* and *last-in-place* assembly schemes, it follows that the storage required to process T_0 is $S_0(\texttt{MinMEM}) \stackrel{def}{=} S_r(\texttt{MinMEM}) = 2M_0$, leading to a volume of I/O $V_0^{I/O} \stackrel{def}{=} V_r^{I/O} = M_0$. We now define a set of properties $\mathcal{P}_k$, $k \geq 0$, as follows.

Property $\mathcal{P}_k$. Given a subtree T, T has the property $\mathcal{P}_k$ if and only if (i) T is of height $k + 1$; (ii) the peak of storage for T is $S(\texttt{MinMEM}) = 2M_0$; and (iii) the frontal

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

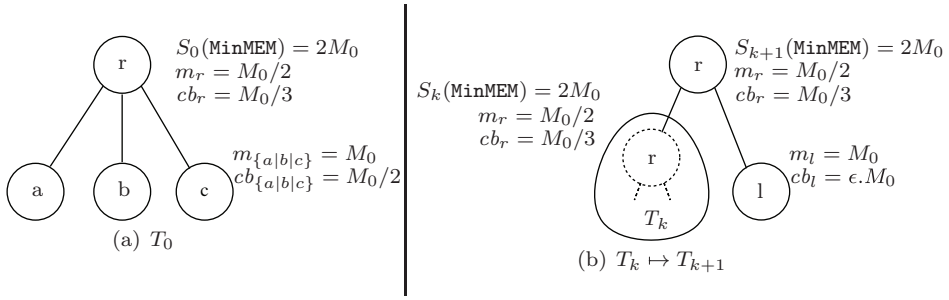


FIG. 5.1. *Recursive construction of an assembly tree illustrating Theorem* 5.1.

matrix at the root (r) of T is of size $m_r = M_0/2$ with a contribution block of size $cb_r = M_0/3$.

By definition, T_0 has property $\mathcal{P}_0$. Given a subtree T_k which verifies $\mathcal{P}_k$, we now build recursively another subtree T_{k+1} which verifies $\mathcal{P}_{k+1}$. To proceed we root T_k and a leaf node (l) to a new parent node (r), as illustrated in Figure 5.1(b). The frontal matrix of the root node has characteristics $m_r = M_0/2$ and $cb_r = M_0/3$, and the leaf node (l) is such that $m_l = S_l = M_0$ and $cb_l = \epsilon M_0$. The value of ϵ is not fixed yet, but we suppose $\epsilon < 1/10$. The active memory usage for T_k and (l) are $A_k = \min(S_k, M_0) = M_0$ and $A_l = \min(S_l, M_0) = M_0$. Because all trees T_k (including T_0) verify the constraints defined at the beginning of section 2, it is possible to associate a matrix to each of these trees. `MinMEM` processes such a family in the order (T_k-l-r) because $S_k - cb_k > S_l - cb_l$. This leads to a peak of storage equal to $S_{k+1}(\texttt{MinMEM}) = 2M_0$ (obtained when processing T_k). Thus T_{k+1} verifies $\mathcal{P}_{k+1}$. We note that `MinMEM` leads to a volume of I/O equal to $V^{I/O}_{k+1}(\texttt{MinMEM}) = M_0/3 + V^{I/O}_k(\texttt{MinMEM})$ (formulas (2.3) and (3.2) for the *classical* and *last-in-place*, respectively).

Since $S_k(\texttt{MinIO})$ is greater than or equal to $S_k(\texttt{MinMEM})$, we can deduce that `MinIO` would process the family in the order (l-T_k-r) because $A_l - cb_l > A_k - cb_k$ (or $\max(A_l, m_r) - cb_l > \max(A_k, m_r) - cb_k$ in the *last-in-place* case). In that case, we obtain a peak of storage $S_{k+1}(\texttt{MinIO}) = \epsilon M_0 + S_k(\texttt{MinIO})$ and a volume of I/O $V^{I/O}_{k+1}(\texttt{MinIO}) = \epsilon M_0 + V^{I/O}_k(\texttt{MinIO})$.

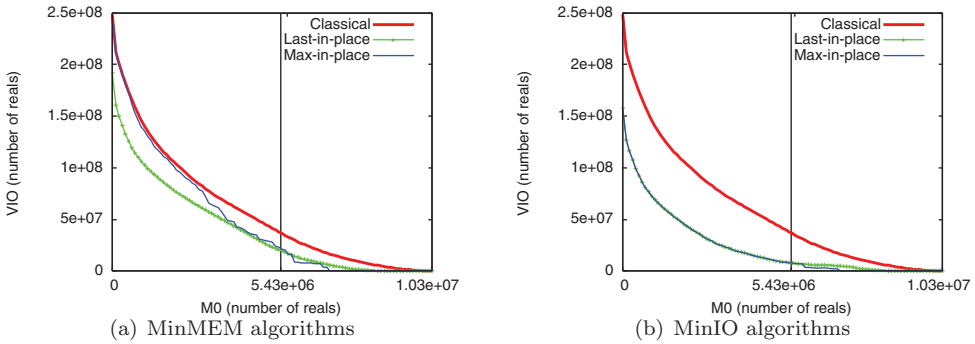
Recursively, we may build a tree T_n by applying n times this recursive procedure. As $S_0(\texttt{MinIO}) = 2M_0$, we deduce that $S_n(\texttt{MinIO}) = (2+n\epsilon)M_0$, which is strictly greater than $S_n(\texttt{MinMEM}) = 2M_0$. Furthermore, because $V^{I/O}_0(\texttt{MinMEM}) = V^{I/O}_0(\texttt{MinIO}) = M_0$, we conclude that $V^{I/O}_n(\texttt{MinMEM}) = nM_0/3 + M_0$, while $V^{I/O}_n(\texttt{MinIO}) = n\epsilon M_0 + M_0$. We thus have $V^{I/O}_n(\texttt{MinMEM})/V^{I/O}_n(\texttt{MinIO}) = (1+n/3)/(1+n\epsilon)$. Fixing $n = \lceil 6\alpha \rceil$ and $\epsilon = 1/\lceil 6\alpha \rceil$, we finally get $V^{I/O}_n(\texttt{MinMEM})/V^{I/O}_n(\texttt{MinIO}) \geq \alpha$.

We have shown that the I/O volume induced by `MinMEM`, $V^{I/O}(\texttt{MinMEM})$, is at least α times larger than the one induced by `MinIO`. To conclude we have to show that it would have been the case for any memory-minimization algorithm (and not only `MinMEM`). This is actually obvious, since the postorder which minimizes the memory is unique: (l) has to be processed after T_k at any level of the tree. □

6. Experimental results. In this section we experiment with the behavior of the strategies presented in sections 2, 3, and 4 on 30 matrices, numbered from 1 to 30: `AUDIKW_1`, `BCSSTK`, `BMWCRA_1`, `BRGM`, `CONESHL_MOD`, `CONV3D_64`, `GEO3D-20-20-20`,

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.



(a) MinMEM algorithms (b) MinIO algorithms

FIG. 6.1. *I/O volume on matrix* TWOTONE *with* PORD *ordering as a function of the core memory available, for the three assembly schemes presented above, for both* MinMEM *and* MinIO *algorithm. The vertical bar represents the size of the largest frontal matrix.*

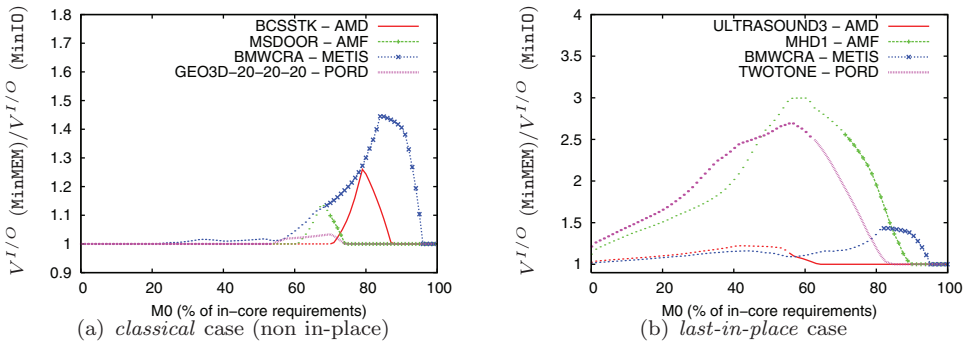
GEO3D-50-50-50, GEO3D-80-80-80, GEO3D-20-50-80, GEO3D-25-25-100, GEO3D-120-80-30, GEO3D-200-200-200, GUPTA1, GUPTA2, GUPTA3, MHD1, MSDOOR, NASA1824, NASA2910, NASA4704, SAYLR1, SHIP_003, SPARSINE, THERMAL, TWOTONE, ULTRASOUND3, ULTRASOUND80, WANG3, and XENON2. These matrices are available from the Parasol [25], University of Florida [26], or GridTLSE [27] collections except matrix CONV3D_64, which was earlier described in [4] and comes from Commissariat à l'Énergie Atomique (CEA-CESTA) (generated with code AQUILON).

We used several ordering heuristics—AMD [3], AMF, METIS [16], and PORD [23]—that result in different task dependency graphs (or assembly trees) for a given matrix and impact the computational complexity. The volumes of I/O were computed by instrumenting the analysis phase of the MUMPS solver [4]. The matrices have a size from very small up to very large (a few million equations) and can lead to huge factors (and storage requirements). For example, the factors of matrix CONV3D_64 with AMD ordering represent 53 *GB* of data.

As previously mentioned, the I/O volume depends on the amount of core memory available. Figure 6.1 illustrates this general behavior on a sample matrix, TWOTONE ordered with PORD, for the three assembly schemes presented above, for both MinMEM and MinIO. For all assembly schemes and algorithms used, we first notice that exploiting all the available memory is essential to limit the I/O volume. Before discussing the results we remind the reader that the I/O volumes presented are valid under the hypothesis that the largest frontal matrix may hold in-core. With a core memory lower than this value (i.e., the area on the left of the vertical bar in Figure 6.1), the I/O volumes presented are actually lower bounds on the effective I/O volume: they are computed as if we could process the out-of-core frontal matrices with a read-once write-once scheme. They, however, remain meaningful because the extra I/O cost due to the specific treatment of frontal matrices will be independent of the assembly scheme used. We first notice that the *last-in-place* assembly schemes strongly decrease the amount of I/O compared to the *classical* assembly scheme of section 2. In fact, using an *in-place* assembly scheme is very useful in an out-of-core context: on most of our matrices, we observed that it reduces the I/O volume by a factor of two. With the *classical* assembly scheme, we observe (on matrix TWOTONE) that MinIO and MinMEM produce the same I/O volume (their graphs are identical). Let us come back to formula (2.3) to explain this behavior. We have minimized $\max(\max_{j=1,n}(A_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k)$ by minimizing the first term because

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php



(a) *classical* case (non in-place) (b) *last-in-place* case

FIG. 6.2. *I/O volume obtained with* `MinMEM` *divided by the one obtained with* `MinIO`. *For each assembly scheme,* M_0 *(x-axis) is normalized with the minimum memory requirement of the stack in an in-core context (application of* `MinMEM`*). For each matrix/ordering, the filled (right) part of the curve matches the area where the amount of core memory is larger than the size of the largest frontal matrix. The dotted (left) part is there as an indication but would only be useful for codes able to manage frontal matrices out-of-core.*

the second one is constant; on this particular matrix the second term is generally the largest, and there is nothing to gain. In other words, the larger the frontal matrices (m in the formula) compared to the other metrics (contribution blocks cb_k and active memory requirements for the subtrees A_j), the lower the probability that reordering the children will impact the I/O volume. From our set of matrices, we have extracted four cases (one for each ordering strategy) for which the gains are significant, and we report them in Figure 6.2(a). To better illustrate the gains resulting from the `MinIO` algorithm, we analyze the I/O ratios as a function of the amount of core memory available (in percentage of the core memory requirements). For instance, the point ($x = 80\%, y = 1.3$) (obtained for both `BCSSTK` and `BMWCRA`) means that `MinMEM` leads to 30% more I/O than `MinIO` when 80% of the in-core memory requirement is provided. Values lower than 1 are not possible because `MinIO` is optimal.

We now focus in Figure 6.2(b) on the *in-place* assembly scheme (described in section 3). We again present four cases (one for each ordering strategy) for which `MinIO` was significantly more efficient than `MinMEM`: for instance, the I/O volume was reduced by a factor of two for a wide range of core memory amounts on the `MHD1-AMF` matrix. Rather than showing the graphs obtained for our whole collection of matrices, we report in Figure 6.3 the largest gains observed for each matrix on the range of values of M_0 larger than the size of the largest frontal matrix. We observe that there is much more to gain with the *last-in-place* assembly scheme than with the *classical* scheme. The largest gain is obtained for the case `SPARSINE-PORD`, where `MinIO` is better than `MinMEM` by a factor of 5.58. Generally, the largest profits from `MinIO` are obtained when matrices are preprocessed with orderings which tend to build irregular assembly trees: `AMF`, `PORD`, and—to a lesser extent—`AMD` (see [13] for more information on the impact of ordering on tree topologies). This is because on such trees, there is a higher probability to be sensitive to the order of children.

In Figure 6.4(a), we show by how much the `MinIO` algorithm with a *max-in-place* assembly scheme improved the `MinIO` *last-in-place* one, again on four matrices of the collection (one for each ordering heuristic) for which we observed large gains. We observe in Figure 6.4(a) that the *last-in-place* and *max-in-place* `MinIO` schemes induce the same volume of I/O when the available core memory decreases: the ratio

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

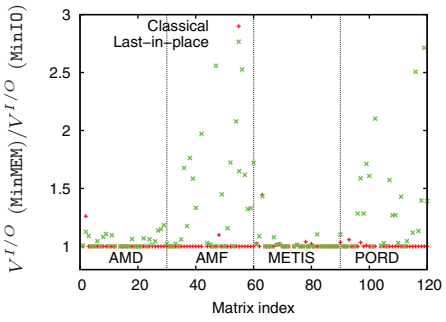
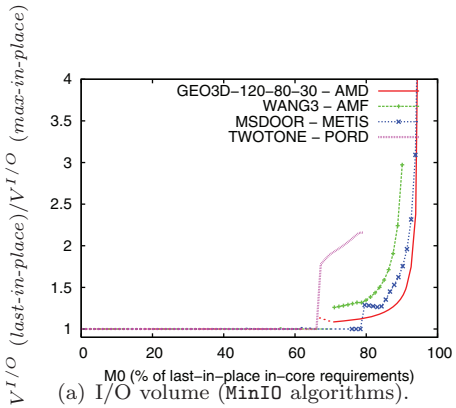
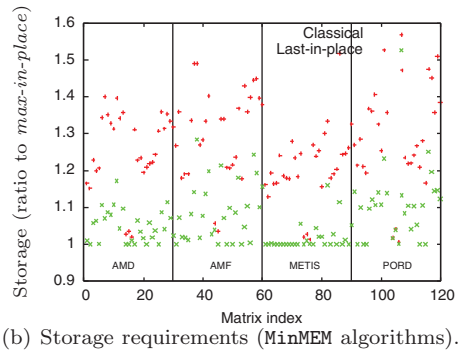


FIG. 6.3. *I/O volume obtained with* `MinMEM` *divided by the one obtained with* `MinIO`*. Thirty test matrices are ordered with four reordering heuristics separated by vertical bars. For each matrix-ordering, we report the largest gain obtained over all values of* M_0 *that exceed the size of the largest frontal matrix. The ratios for* `GEO3D-25-25-100-AMF` *and* `SPARSINE-PORD` *in the* last-in-place *scheme are equal to* 5.12 *and* 5.58 *and are not represented in the graph.*



(a) I/O volume (`MinIO` algorithms).



(b) Storage requirements (`MinMEM` algorithms).

FIG. 6.4. *Impact of* max-in-place *assembly scheme.*

is equal to 1. This is because, in this case, the `MinIO` heuristic for the *max-in-place* assembly variant switches to the *last-in-place* scheme (as explained in section 4) and has exactly the same behavior, since the switch happens very early.

Finally, Figure 6.4(b) shows that the peak of storage (critical for the in-core case) can also be significantly decreased thanks to a *max-in-place* allocation. In the range of M_0 values large enough to process the matrix in core with a *max-in-place* assembly scheme but too small to process it in core with a *last-in-place* scheme, I/O is induced only with a *max-in-place* scheme. Therefore, the right-extreme parts of the curves in Figure 6.4(a) tend to (or are equal to) infinity.

7. Memory management. The different `MinMEM` and `MinIO` algorithms presented in this paper provide a particular postorder of the assembly tree. They can be applied during the analysis phase of a sparse direct solver. Then the numerical factorization phase relies on this traversal and should respect the forecasted optimal metrics (memory usage, I/O volume). In this section we suppose that a postorder has been given by one of the algorithms presented earlier, and we present memory management algorithms for the numerical factorization phase that match the different assembly schemes we have considered. We show that our models can lead to a rea-

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

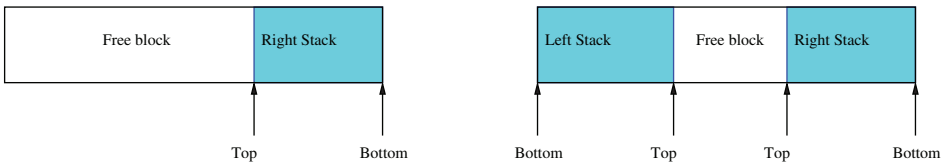


FIG. 7.1. *Subdivision of the main workarray, W, into one stack (left) or two stacks (right) of contribution blocks. The free block can be used to store the temporary frontal matrices.*

sonable implementation during the numerical factorization phase, relying on simple stack mechanisms. Remember that we consider that the factors are written to disk on the fly. As soon as a block of the frontal matrix is factored, it can be written to disk, possibly asynchronously. Thus we have to only store temporary frontal matrices and contribution blocks. We assume that those must be stored in a preallocated contiguous workarray W of maximum size M_0, the available core memory. In this workarray, we manage one or two stacks depending on our needs, as illustrated in Figure 7.1. Another approach would consist in relying on dynamic allocation routines (such as `malloc` and `free`). Although those may still be efficient from a performance point of view, the use of such a preallocated workarray has several advantages over dynamic allocation, allowing for a tighter memory management as long as complicated garbage collection mechanisms can be avoided. In particular, several memory operations are possible with a workarray managed by the application that would be difficult or even impossible with standard dynamic allocation tools:

- In-place assemblies: with dynamic allocation, expanding the memory of the contribution block of a child node into the memory of the frontal matrix of the parent node would require us to rely on a routine that extends the memory for the contribution block (such as `realloc`). This may imply an extra copy, which cancels the advantages of in-place assemblies; with a preallocated workspace, we simply shift some integer pointers.
- Assuming that the frontal matrix uses a dense row-major or column-major storage and that the factors have been copied to disk, we can copy the contribution block of such a frontal matrix into a contiguous memory area that overlaps with the original location. With dynamic allocation, we would need to allocate the memory for the contribution block, perform the copies, and then free the memory for the original frontal matrix. Even assuming that the contribution block is compacted in-place (inside the memory allocated for the frontal matrix), then it is not clear how to free the rest of the frontal matrix with dynamic allocation tools, whereas this can be done by shifting an integer pointer in our case.

Finally, the preallocated workarray allows for a very good locality, for example, when assembling the contributions from children into the frontal matrix of the parent, the entries of all the contribution blocks are contiguous in memory.

7.1. In-core stack memory. Before dealing with the out-of-core management of contribution blocks (section 7.2), we first describe mechanisms corresponding to an in-core stack management. Those can be applied when the storage requirement is smaller than the available memory M_0.

7.1.1. Recalling the classical and last-in-place assembly schemes. The *classical* and *last-in-place* approaches are already used in existing multifrontal codes.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

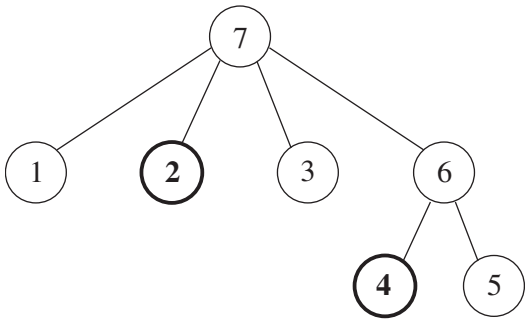


FIG. 7.2. *Example of a tree with* 7 *nodes. Nodes in bold correspond to the nodes with the largest contribution block among the siblings. (This property will only be used in section* 7.1.2*.)*

We recall them in this section in order to introduce notions that we will use in the other subsections. We have seen earlier (first part of Property 1) that since we have a postorder traversal, the access to the contribution blocks has the behavior of a stack (in general, one uses the stack on the right of W). In other words, thanks to the postorder, we have the following property.

PROPERTY 2. *If the contribution blocks are stacked when they are produced, each time a frontal matrix is allocated, the contribution blocks from its children are available at the top of the stack.*

For example, at the moment of allocating the frontal matrix of node (6) in the tree of Figure 7.2, the stack contains, from bottom to top, cb_1, cb_2, cb_3, cb_4, cb_5. The frontal matrix of (6) is allocated in the free block, then cb_5 and cb_4 (in that order) are assembled into it and removed from the stack. Once the assembly at the parent is finished, the frontal matrix is factorized, the factors are written to disk, and the contribution block (cb_6) is moved to the top of the stack.

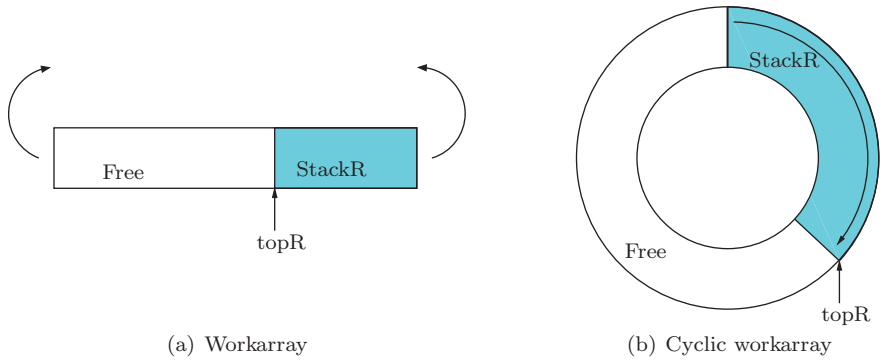
The only difference between the *classical* and the *last-in-place* assembly schemes is that in the *last-in-place* case, the memory for the frontal matrix of the parent is allowed to overlap with the memory of the child available at the top of the stack. In the example, this means that if the free block on the left is not large enough for the frontal matrix of (6), that frontal matrix is allowed to overlap with the memory of the contribution block of (5), of size cb_5, leading to significant memory gains. The contribution block of the child is expanded into the memory of the frontal matrix of the parent, and the contribution blocks from the other children are then assembled normally.

7.1.2. In-place assembly of the largest contribution block. The new *max-in-place* assembly scheme introduced in section 4 consists of overlapping the memory of the parent with the memory of the largest child contribution block. For this to be possible, the largest contribution block must be available in a memory area next to the free block where the frontal matrix of the parent will be allocated. By using a special stack for the largest contribution blocks (the one on the left of W; see Figure 7.1), Property 2 also applies to the largest contribution blocks. Thus, when processing a parent node,

- the largest child contribution is available at the top of the left stack and can overlap with the frontal matrix of the parent; and
- the other contributions are available at the top of the right stack, just like in the *classical* case.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php



(a) Workarray (b) Cyclic workarray

FIG. 7.3. *Folding a linear workarray (left) into a cyclic workarray (right).*

This is illustrated by the tree of Figure 7.2. When traversing that tree, we first stack cb_1 on the right of W, then stack cb_2 (identified as the largest among its siblings) on the left of W, then cb_3 on the right, cb_4 on the left, and cb_5 on the right. When node (6) is processed, the workarray W contains the following:

cb_2	cb_4	Free block	cb_5	cb_3	cb_1

The memory for the frontal matrix of (6) can overlap with cb_4 so that cb_4 is assembled *in-place*; cb_5 is then assembled normally. Note that the same type of situation will occur for the root node (7): cb_2 (now available at the top of the left stack) will first be assembled in-place; then cb_6, cb_3, and cb_1 (in that order) will be assembled from the right stack.

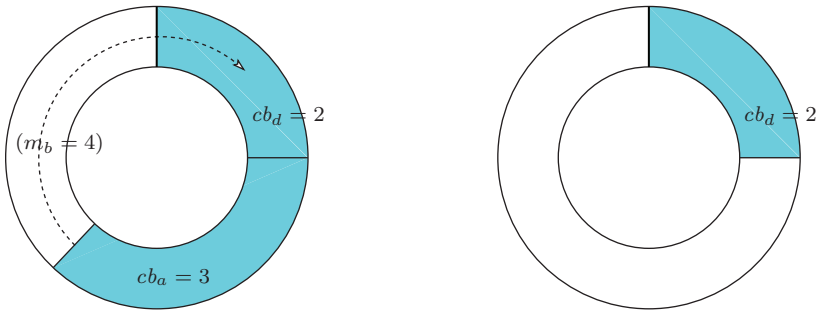
7.2. Out-of-core context. We now assume that contribution blocks may be written to disk when needed. When there is no more memory, Property 1 suggests that the bottom of the stack(s) should be written to disk first. Therefore, the question of how to reuse the corresponding workspace arises. We give a first natural answer to this question in section 7.2.1, but it has some drawbacks and does not apply to all cases. Based on information that can be computed during the analysis phase, we then propose in section 7.2.2 a new approach that greatly simplifies the memory management for all the considered assembly schemes.

7.2.1. Dynamic cyclic memory management. In the *classical* and *last-in-place* cases, only one stack is required. In order for new contribution blocks (stored at the top of the stack) to be able to reuse the space available at the bottom of the stack after write operations, a natural approach consists of using a cyclic array. From a conceptual point of view, the cyclic memory management is obtained by joining the end of the memory zone to its beginning, as illustrated in Figure 7.3. In this approach, the decision to free a part of the bottom of the stack is taken dynamically when the memory is almost full. We illustrate this on the sample tree of Figure 2.1 processed in the postorder (d-a-b-c-e) with a *classical* assembly scheme. After processing nodes (d) and (a), one discovers that I/O has to be performed on the first contribution block produced (cb_d) only at the moment of allocating the frontal matrix of (b), of size $m_b = 4$ (see Figure 7.4(a)).

Note that a significant drawback of this approach is that a specific management has to be applied to the border, especially when a contribution block or a frontal

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php



(a) Dynamic cyclic approach (b) Top-down approach

FIG. 7.4. *Memory state while processing the tree of Figure* 2.1 *in the postorder (d-a-b-c-e). The size of the workarray is* $M_0 = 8$. *With a dynamic approach (left), one discovers that I/O will be performed on* cb_d *only before dealing with node (b). With the approach of section* 7.2.2 *(right), we know a priori that* cb_d *must be fully written to disk thanks to the analysis phase.*

matrix is split on both sides of the memory area (as occurs for frontal matrix m_b in Figure 7.4(a)).

Moreover, in the *max-in-place* case, such an extension is not as natural because of the existence of two stacks. That is why we propose in the next subsection another approach, which avoids a specific management of the borders for the *classical* and *last-in-place* cases and allows us to efficiently handle the *max-in-place* case.

7.2.2. Using information from the analysis: Static top-down formulation. In order to minimize the I/O volume in the previous approach, a contribution is only written to disk when the memory happens to be full: the decision of writing a contribution block (or a part of it) is taken dynamically. However, a better approach can be adopted. We explain it by listing some properties, each new property being the consequence of the previous one.

PROPERTY 3. *While estimating the volume of I/O, the analysis phase can forecast whether a given contribution block will have to be written to disk or not.*

This property results from an instrumentation of the analysis phase that we described in the following. When considering a parent node with n child subtrees, the volume of I/O $V_{family}^{I/O}$ performed on the children of that parent node is given by the first member (the recursive amount of I/O on the subtrees is not counted) of formulas (2.3) and (3.2), respectively, for the *classical* and *in-place* cases. For example,

$$V_{family}^{I/O} = \max\left(0, \max\left(\max_{j=1,n}\left(A_j + \sum_{k=1}^{j-1} cb_k\right), m + \sum_{k=1}^{n} cb_k\right) - M_0\right) \tag{7.1}$$

in the *classical* assembly scheme. Given $V_{family}^{I/O}$ and knowing that we are going to write the contribution blocks produced first in priority, one can easily determine if the contribution block cb_j of the jth child must be written to disk:

- if $\sum_{i=1}^{j} cb_i \leq V_{family}^{I/O}$, the volume of I/O for that family is not reached even when cb_j is included; therefore, cb_j must be entirely written to disk;
- if $\sum_{i=1}^{j-1} cb_i < V_{family}^{I/O} < \sum_{i=1}^{j} cb_i$, then cb_j should be partially written to disk, and the volume written is $V_{family}^{I/O} - \sum_{i=1}^{j-1} cb_i$;
- otherwise, cb_j remains in-core.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

In the tree of Figure 2.1 processed in the order (d-a-b-c-e), the volume of I/O for the family defined by the parent (e) and the children (d) and (c) is equal to 3. According to what is written above, this implies that $cb_d = 2$ must be entirely written to disk and that 1 unit of I/O must be performed on cb_c.

PROPERTY 4. *Because the analysis phase can forecast whether a contribution block (or part of it) will be written to disk, one can also decide to write it (or part of it) as soon as possible, that is, as soon as the contribution is produced. This will induce the same overall I/O volume.*

Thanks to Property 4, we will assume in the following that Strategy 1 defined below always holds.

STRATEGY 1. *We decide to write all the contribution blocks which have to be written as soon as possible.*

This is illustrated in Figure 7.4(b): as soon as the contribution block of node (d) (cb_d) is produced, we know that it has to be written to disk, and we can decide to write it as soon as possible, i.e., before processing node (a). Therefore, we can free the memory for the contribution block of (d) before allocating the frontal matrix of (a) (by using synchronous I/Os or, more efficiently, with pipelined asynchronous I/Os).

PROPERTY 5. *Each time a contribution block has to be written, it is alone in memory: all the previous contribution blocks are already on disk.*

In other words, it is no longer required to write the bottom of a stack, as it was suggested in Property 1. A slightly stronger property is the following.

PROPERTY 6. *If a subtree requires some I/O, then at the moment of processing the first leaf of that subtree, the workarray W is empty.*

This is again because we should write the oldest contribution blocks first, and those have been written as soon as possible. A corollary from the two previous properties is the following.

PROPERTY 7. *When we stack a contribution block on a nonempty stack, we will never write it. Otherwise, we would have written the rest of the stack first. In particular, if a given subtree can be processed in-core with a memory $S \leq M_0$, then at the moment of starting this subtree, the contiguous free block of our workarray W is necessarily at least as large as S.*

It follows that by relying on Strategy 1 a cyclic memory management is not needed anymore: a simple stack is enough for the *classical* and *last-in-place* assembly schemes, and a double stack is enough for the *max-in-place* assembly scheme. In the latter case, a double stack is required only for processing in-core subtrees, since our *max-in-place* + `MinIO` heuristic switches to *last-in-place* for subtrees involving I/O (as explained in section 4).

We illustrate this strategy on the *max-in-place* + `MinIO` variant of section 4 (although it applies to all `MinIO` approaches). We assume that the analysis phase has identified in-core subtrees (processed with `MinMEM` + *max-in-place*) and out-of-core subtrees (processed with `MinIO` + *last-in-place*). We also assume that the contribution blocks that must be written to disk have been identified. The numerical factorization is then illustrated by Algorithm 7.1. It is a top-down recursive formulation, more natural in our context, which starts with the application of `AlgoOOC_rec`() on the root of the tree. A workarray W of size M_0 is used.

7.3. Limits of the model. Until now, we have considered multifrontal solvers without delayed pivot eliminations between children and parent nodes. In that case, the forecasted metrics from the analysis are exactly respected during the numeri-

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

```
% W: workarray of size M_0
% n: number of child subtrees of tree T
for j = 1 to n do
    if the subtree T_j rooted at child j can be processed in-core in W then
        % We know that the free contiguous block in W is large
          enough thanks to Property 7
        Apply the max-in-place approach (see section 7.1.2);
    else
        % Some I/O are necessary on this subtree, therefore W is
          empty (Property 6)
        % We do a recursive call to AlgoOOC_rec(), using all the
          available workspace
        AlgoOOC_rec(subtree T_j);
    Write cb_j to disk or stack it (decision based on Property 3 and Strategy 1);
Allocate frontal matrix of the parent node; it can overlap with cb_n;
for j = n downto 1 do
    Assemble cb_j in the frontal matrix of the root of T (reading from disk v_j
    units of data, possibly by panels);
Factorize the frontal matrix; except for the root node, this produces a
contribution block;
```

ALGORITHM 7.1. `AlgoOOC_rec`(*tree* T).

cal factorization, and the tree traversals obtained are optimal. In particular, Algorithm 7.1 can be applied and implemented as presented.

Let us now allow dynamic pivoting that results in delayed pivot eliminations from some children nodes to their parents or ancestors [11]. The size of the associated contribution blocks increases to include the delayed pivot rows/columns, leading to an increase of the quantities cb and m. Because such numerical difficulties can hardly be predicted but often remain limited in practice with proper preprocessing, it seems reasonable to us to keep the tree traversal obtained with the original metrics from the analysis. In the case of an in-core stack, the memory management algorithms from section 7.1 can still be applied—including the memory management for our new *max-in-place* scheme presented in section 7.1.2—as long as the memory size is large enough.

In the context of an out-of-core stack, the approaches from section 7.2.2 do not apply directly because the storage for a subtree may be larger than forecasted when numerical difficulties occur. Imagine, for example, that a subtree which was scheduled to be processed in-core no longer fits in memory because of delayed eliminations within the subtree. Alternative strategies to deal with those numerical difficulties are required, which are outside the scope of this paper. Recovering from a situation where the strategy has been too optimistic may require a significant amount of extra, unpredicted I/O and/or memory copies. A safer approach could consist of relaxing the forecasted metrics with a predefined percentage and artificially limiting the amount of delayed eliminations to remain within that percentage. Finally, storing delayed rows/columns into separate data structures with a separate out-of-core management when necessary might be another option.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

TABLE 8.1
Summary. Contributions of this paper are in bold.

Assembly scheme	Algorithm	Objective function		Workspace management	
		Memory minimization (in-core stack)	**I/O minimization (out-of-core stack)**	In-core	**Out-of-core**
classical	MinMEM	• Optimum ([13], adapting[17])	• **Arbitrarily bad in theory** • **Reasonable in most cases**	One stack is enough	**Cyclic or static top-down**
	MinIO	• Not suited	• **Optimum**		
last-in-place	MinMEM	• Optimum[17]	• **Arbitrarily bad in theory** • **Bad in practice on some irregular assembly trees**		
	MinIO	• Not suited	• **Optimum**		
max-in-place	**MinMEM**	• **Optimum**	• Not suited	**Two stacks**	**Top-down**
	MinIO	• **Optimum**	• **Efficient heuristic**		

8. Conclusion and future work. Table 8.1 summarizes the contributions of this paper. We have reminded the existing memory-minimization algorithms for the *classical* and *last-in-place* assembly schemes when the stack of contribution blocks remains in core memory. We have shown that these algorithms are not optimal to minimize the I/O volume when part of the stack is allowed to be written to disk and that they can lead to an arbitrarily bad volume of I/O compared to an optimal strategy. We have proposed optimal algorithms for the I/O volume minimization and have shown that significant gains could be obtained on real-world problems, especially with the *last-in-place* assembly scheme. We have also presented a new assembly scheme (which consists of extending the largest child contribution into the frontal matrix of the parent) and a corresponding postorder which is optimal to minimize memory. This new assembly scheme leads to a very good heuristic when the objective is to minimize the I/O volume. In the table, "Not suited" means that the algorithm should not be applied in that case. For example, when the stack is in core memory and the objective is to decrease the peak of core memory, it makes no sense to try to minimize the volume of I/O related to the stack (`MinIO`). The "Not suited" of column "I/O minimization," row "*max-in-place*/`MinMEM`" comes from the fact that we cannot guarantee keeping the largest contribution block in core memory (see discussion at the end of section 4).

From a practical point of view, we have shown that efficient memory management schemes (not inducing extra core memory traffic) could be obtained for all variants and have proposed simple memory management algorithms that provide a good basis for actual implementations.

This work can be particularly important for large-scale problems (millions of equations) in limited-memory environments. It is applicable for shared-memory solvers relying on threaded `BLAS` libraries. In a parallel distributed context, it will help to limit the memory requirements and to decrease the I/O volume in the serial (often critical) parts of the computations.

We are currently working on adapting and generalizing our results to a more flexible task allocation scheme, where the parent node is allowed to be allocated before all children have been processed [14]. Again, instead of limiting the storage requirement of the methods, the goal is to minimize the volume of I/O involved. The work presented in this paper is a basis to this new and more difficult yet flexible context.

Acknowledgment. The authors are grateful to the two anonymous referees for their useful comments and to Patrick Amestoy for his remarks on a preliminary version of this paper.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/16/12 to 194.199.19.9. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

REFERENCES

[1] E. Agullo, A. Guermouche, and J.-Y. L'Excellent, *On reducing the I/O volume in a sparse out-of-core solver*, in Proceedings of the HiPC'07 14th International Conference On High Performance Computing, Goa, India, 2007, Lecture Notes in Comput. Sci. 4873, Springer, New York, 2007, pp. 47–58.

[2] E. Agullo, A. Guermouche, and J.-Y. L'Excellent, *A parallel out-of-core multifrontal method: Storage of factors on disk and analysis of models for an out-of-core active memory*, Parallel Comput., Special Issue on Parallel Matrix Algorithms, 34 (2008), pp. 296–317.

[3] P. R. Amestoy, T. A. Davis, and I. S. Duff, *Algorithm* 837: *AMD, an approximate minimum degree ordering algorithm*, ACM Trans. Math. Software, 33 (2004), pp. 381–388.

[4] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, *Hybrid scheduling for the parallel solution of linear systems*, Parallel Comput., 32 (2006), pp. 136–156.

[5] P. R. Amestoy and C. Puglisi, *An unsymmetrized multifrontal LU factorization*, SIAM J. Matrix Anal. Appl., 24 (2002), pp. 553–569.

[6] C. Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, and H. D. Simon, *Progress in sparse matrix methods for large linear systems on vector computers*, 1 (1987), Int. J. Supercomput. Appl., pp. 10–30.

[7] F. Dobrian and A. Pothen, *Oblio: A Sparse Direct Solver Library for Serial and Parallel Computations*, Technical report, Old Dominion University, Norfolk, VA, 2000.

[8] F. Dobrian, *External Memory Algorithms for Factoring Sparse Matrices*, Ph.D. thesis, Old Dominion University, Norfolk, VA, 2001.

[9] I. S. Duff and S. Pralet, *Towards stable mixed pivoting strategies for the sequential and parallel solution of sparse symmetric indefinite systems*, SIAM J. Matrix Anal. Appl., 29 (2007), pp. 1007–1024.

[10] I. S. Duff and J. K. Reid, *MA*27*—A Set of Fortran Subroutines for Solving Sparse Symmetric Sets of Linear Equations*, Technical report R.10533, AERE, Harwell, England, 1982.

[11] 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.

[12] I. S. Duff and J. K. Reid, *The multifrontal solution of unsymmetric sets of linear systems*, SIAM J. Sci. Stat. Comput., 5 (1984), pp. 633–641.

[13] A. Guermouche, J.-Y. L'Excellent, and G. Utard, *Impact of reordering on the memory of a multifrontal solver*, Parallel Comput., 29 (2003), pp. 1191–1218.

[14] A. Guermouche and J.-Y. L'Excellent, *Constructing memory-minimizing schedules for multifrontal methods*, ACM Trans. Math. Software, 32 (2006), pp. 17–32.

[15] P. Hénon, P. Ramet, and J. Roman, *PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems*, Parallel Comput., 28 (2002), pp. 301–321.

[16] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1999), pp. 359–302.

[17] 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. 127–148.

[18] J. W. H. Liu, *The multifrontal method and paging in sparse Cholesky factorization*, ACM Trans. Math. Software, 15 (1989), pp. 310–325.

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

[20] J. K. Reid and J. A. Scott, *An Out-of-Core Sparse Cholesky Solver*, ACM Trans. Math. Software, 36 (2009), pp. 1–33.

[21] E. Rothberg and R. Schreiber, *Efficient methods for out-of-core sparse Cholesky factorization*, 21 (1999), SIAM J. Sci. Comput., pp. 129–144.

[22] 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.

[23] J. Schulze, *Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods*, BIT, 41 (2001), pp. 800–841.

[24] *BCSLIB-EXT*, Commercial code originally developed at Boeing Computer Services; acquired by Access Analytics in December 2007.

[25] Parasol, http://www.parallab.uib.no/parasol.

[26] University of Florida, http://www.cise.ufl.edu/research/sparse/matrices.

[27] TLSE (Test for Large Systems of Equations), http://www.gridtlse.org.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.