PARALLEL COMPUTATION OF ENTRIES OF A^{-1}*

PATRICK R. AMESTOY†, IAIN S. DUFF‡, JEAN-YVES L'EXCELLENT§, AND FRANÇOIS-HENRY ROUET¶

Abstract. In this paper, we consider the computation in parallel of several entries of the inverse of a large sparse matrix. We assume that the matrix has already been factorized by a direct method and that the factors are distributed. Entries are efficiently computed by exploiting sparsity of the right-hand sides and the solution vectors in the triangular solution phase. We demonstrate that in this setting, parallelism and computational efficiency are two contrasting objectives. We develop an efficient approach and show its efficiency on a general purpose parallel multifrontal solver.

Key words. sparse matrices, direct methods for linear system and matrix inversion, parallel algorithms

DOI. 10.1137/120902616

1. Introduction.

There are many applications where the computation of explicit entries of the inverse of a sparse matrix is required. Perhaps the most common reason is in statistical analysis of least-squares problems where the diagonal entries of the inverse of the normal equations matrix correspond to the variances and the off-diagonal entries to covariances [5]. However, there are other applications requiring such computations, such as atomistic level simulation of nanowires [8, 12], computation of short-circuit currents [16] and path resistances in networks [13], and approximations of condition numbers [4]. In most of these cases, many entries are wanted, for example all the entries of the diagonal.

We address this issue in the context of sparse direct methods where a sparse factorization of the matrix has already been performed, and we want to compute inverse entries by forward and backward substitution using the matrix factors. In previous works we examined the computation of inverse entries in a serial context [2, 18, 14]; in particular, we focused on grouping the computations into blocks (*blocking*) and investigated how to form blocks in different contexts, in particular out-of-core [2] and in-core settings [18]. We briefly describe these works in section 2.2. In this paper, we examine the computation of inverse entries in parallel using a distributed sparse

†Université de Toulouse, INPT(ENSEEIHT)-IRIT, Toulouse Cedex 31071, France (amestoy@enseeiht.fr).

‡CERFACS, 31057 Toulouse, France (duff@cerfacs.fr), and Rutherford Appleton Laboratory, Oxon, OX11 0QX, England (iain.duff@stfc.ac.uk). This author's research was supported in part by EPSRC grant EP/I013067/1.

§Inria and Laboratoire LIP (UMR 5668 CNRS - ENS Lyon - Inria - UCBL), Université de Lyon, 69364 Lyon Cedex 07, France (jean-yves.l.excellent@ens-lyon.fr).

¶Lawrence Berkeley National Laboratory, Berkeley, CA 94720 (fhrouet@lbl.gov), and Université de Toulouse, INPT(ENSEEIHT)-IRIT, Toulouse Cedex 31071, France (frouet@enseeiht.fr). This author's research was supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. DOE under contract DE-AC02-05CH11231.

factorization. The main issue in this case is to balance the contrasting requirements of both parallelism and computational efficiency. We achieve this by developing novel strategies for blocking the computations, explain why it is necessary to exploit sparsity within each block, and explain how this can be done efficiently; this is a novel contribution compared to the above-mentioned works. We illustrate the effectiveness of the proposed approach on both small examples and runs on realistic test problems. Our strategies are applicable to any sparse distributed factorization. For our experiments, we use a version of the MUMPS code [1, 3] in which we have incorporated the algorithms discussed in this paper.

The importance of the computation of inverse entries of the matrix has resulted in many papers on this topic. Most do not assume that a factorization of the matrix exists or perhaps even assume that the matrix is not available other than as an operator. In these cases, iterative methods are used to get approximations to the inverse entries; these methods often require specific matrix properties to work efficiently and might not be as robust as our approach for general problems. We refer the reader to the article by Tang and Saad [17] and references therein.

Far less work has been done on this when factors are available. Other works that we are aware of make use of the identity of Takahashi, Fagan, and Chin [16], and so entries of the inverse within the sparsity pattern of the factors are computed in a reverse Crout order. Campbell and Davis [6, 7] even develop additional tree structures to describe this but, to our knowledge, have not released a publicly available code. Lin et al. [10, 11] also compute the entries starting with the (n, n)th entry of A^{-1} and save storage by overwriting the matrix factors by the entries of the inverse. The complexity of their algorithm is thus comparable to the matrix factorization, and they are unable to solve for further entries, as the factors have been overwritten. However, they do show very good performance for a regular 2D problem from electronic structure calculation.

In section 2, we describe the framework in which we are working and define a basic algorithm. At first glance, performing substitutions simultaneously on blocks of vectors appears to be a classic example of embarrassing parallelism, but, after indicating in section 3 that this is not so, we discuss the competing issues of parallel efficiency and computational efficiency, both being necessary to achieve the eventual goal of fast execution while respecting constraints on memory. In section 4, we present the core idea and develop this to describe our algorithms which we illustrate on some small examples. This initial description is for the computation of diagonal entries of the inverse, but we expand this discussion to the case of arbitrary entries of the inverse in section 5. We present our numerical experiments in section 6 and our conclusions in section 7.

2. Computation of entries of A^{-1}.

2.1. Environment. We start by defining the environment in which we are working and the terms that we will use later in the paper. We assume that the matrix has already been factorized and that the factors are distributed over the processes.

A distributed sparse factorization can be represented by an assembly tree where each node of the tree corresponds to the partial factorization of a dense submatrix. We associate with each node of the tree the variables that are eliminated at that node. The factorization is performed from the leaf nodes to the root (we assume without loss of generality that the matrix is irreducible). As the factorization proceeds up the tree towards the root, the submatrices in general become larger, as does the number of variables eliminated at a node. Thus, in order to maintain efficient parallelism, it is normal to use several processes to perform the elimination at a node nearer to the root. We will refer to such a node as a "parallel node." For these nodes we identify a single master process and a set of other processes that help with the factorization at the node.

2.2. Sparse inverse computation. We compute the entries of the inverse by *partially* solving the equation $AA^{-1} = I$. We assume that we have an $L\,U$ factorization of A, where U should be read as $D\,L^T$ or L^T when A is symmetric ($A = LDL^T$). As we show in the following, we accelerate the triangular solution process by exploiting the specific structure of the problem and by eliminating useless operations. One of the advantages is that this method is as stable as a standard direct solution; our approach does not introduce any approximation and is compatible with all numerical algorithms that make direct methods robust, such as scaling and pivoting.

We use the notation a_{ij}^{-1} to denote the (i,j)th entry of the inverse of A. We compute a_{ij}^{-1} using $\left(A^{-1}e_j\right)_i$, where e_j is the jth column of the identity matrix, and $(v)_i$ or v_i denotes the ith component of the vector v. Using the $L\,U$ factorization of A, we obtain a_{ij}^{-1} by solving successively the two triangular systems:

$$\begin{cases} y = L^{-1}e_j, \\ a_{ij}^{-1} = (U^{-1}y)_i. \end{cases} \tag{2.1}$$

We see from equations (2.1) that, in the forward substitution phase, the right-hand side (e_j) contains only one nonzero entry and that, in the backward step, only one entry of the solution vector is required. The following result takes advantage of both these observations along with the sparsity of A to provide an efficient computational scheme.

THEOREM 2.1 (nodes to access to compute a particular entry of the inverse; Property 8.9 in [15]). *To compute a particular entry a_{ij}^{-1} of A^{-1}, the only nodes of the tree which have to be traversed are on the path from the node j up to the root node and on the path going back from the root node to the node i.*

Using this result enables us to "prune" the tree, that is, to remove all the nodes which do not take part in the computation of an entry. Thus, for a single entry of the inverse, the pruned tree only consists of the root node, the nodes i and j, and all nodes on the unique paths between these nodes and the root. Therefore, our method is a refinement of a standard triangular solution process.

We illustrate this in Figure 1, where entry a_{23}^{-1} is computed by following the path from node 3 to the root node and then the path from the root node to node 2. Nodes 1 and 4 are not visited; they are said to be "pruned" from the tree.

Note that in an assembly tree, nodes are associated with a set of variables to be eliminated, called fully summed variables, instead of a single one. Entry a_{ij}^{-1} is then computed by following the path from the node that contains variable j (as a fully summed variable) to the root node and then the path from the root node to the node that contains variable i (as a fully summed variable).

Now suppose that we wish to compute a set R of entries of the inverse. In the applications that we have mentioned, it is quite common for $|R|$ to be very large, sometimes as large as the order of the matrix A. The storage required for processing the $|R|$ right-hand sides when using equations (2.1) can become prohibitive, and so the computation must proceed in blocks where, for each block, only B entries are computed. The concept of tree pruning can be extended to apply to these blocks of

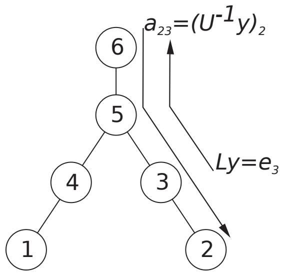


FIG. 1. *Computing a_{23}^{-1} requires traversing the path from node 3 to the root node, then the path from the root node to node 2. Nodes 1 and 4 can be pruned from the tree.*

right-hand sides.

DEFINITION 2.2 (pruned tree of a block). *The pruned tree associated with a block of B target entries of A^{-1} is defined as the union of all pruned trees resulting from the application of Theorem 2.1 to each target entry within the block.*

It is also possible to reorder the target entries of A^{-1} (and thus the right-hand sides) according to different criteria. Definition 2.3 gives one such possible reordering.

DEFINITION 2.3 (postordering of the variables associated with target entries). *Given a postordering of a (possibly pruned) assembly tree, we say that a set of variables is postordered if and only if the order of the variables is compatible with the order in which they appear as fully summed variables in the sequence of postordered nodes of the tree.*

Processing the right-hand sides by blocks normally involves repeated access to the same parts of L and U for different blocks (for example, the root node is accessed in all blocks). In [2, 18], we consider the combinatorial problem of partitioning the requested entries into blocks to minimize the overall cost in different sequential environments, namely an out-of-core and an in-core setting, respectively. In out-of-core execution, the cost is dominated by the cost of loading factors from disk, and thus the objective is to minimize the volume of accesses to the factors. In in-core execution, the main objective is to minimize the number of operations. We developed different hypergraph models and heuristics that exploit the elimination tree in order to minimize these objectives for a given block size B. In this paper, we are concerned with how to compute entries efficiently in parallel in an in-core setting; as we show in the next sections, this entails not only finding a good partition of the requested entries but also revisiting the way they are computed.

3. Parallel and sparsity issues.

Before sketching our overall algorithm for the parallel solution of multiple inverse entries, we first present in this section the algorithmic context and then describe in section 4 the key ideas that have driven our strategy. We start with the case where only diagonal entries are computed and briefly discuss the general case in section 5.

3.1. The difficulty of computing blocks of diagonal entries in parallel.

In the context of computing many entries of the inverse, we solve for several right-hand sides at the same time, and, at first glance, this seems to exhibit the classical phenomenon of being embarrassingly parallel. However, when we combine this with the fact that we wish to exploit a parallel matrix factorization that yields distributed factors (a setting common to all distributed memory solvers), the situation is not that straightforward, and we find that we lack the mechanism to fully exploit the independence of the right-hand sides.

In a distributed memory environment, one possible workaround would be to run parallel instances of the linear solver in parallel, each instance using the whole set of processes to solve for a block of right-hand sides (because all of the distributed factors might have to be accessed). This is not possible using MPI, given that the factors are distributed. Another potential workaround would be to replicate the factors on all processes, or to write the factors "shared" on disk(s), and to simulate a shared memory paradigm by launching sequential instances in parallel, each of them accessing the distributed factors. Unfortunately, this is not feasible for any distributed sparse solver that we know of. Furthermore, such a solution would then really lose the benefit of our parallel factorization, and the cost of accessing the distributed factors (which might be stored on local disks) would be prohibitive. Therefore, the blocks of entries to be computed thus have to be processed one at a time.

Note that, in a shared memory environment where each thread has access to the single copy of the factors held in the main memory, blocked solves could be performed in parallel (perhaps using threaded BLAS). We could expect a good speed-up (up to the number of cores/processors), but the approach will be limited because of constraints in the shared memory system. Parallel accesses to the unique instance of the factors and to multiple blocks of right-hand sides active at the same time might induce considerable bus contention between the threads.

3.2. Sparsity issues. In most sparse direct codes, the sparsity of a block of right-hand sides is exploited neither by blocks nor within the blocks. That is, the tree is not pruned and we perform operations on all the columns of a block. In our discussion, we assume the following computational setting.

Assumption 1. Sparsity is exploited by blocks. When processing a block of right-hand sides, the tree is pruned specifically for this block, as mentioned in Definition 2.2 from section 2.2.

Assumption 2. Sparsity is not exploited within a block: the block of right-hand sides is considered the computational unit, and blocked operations (e.g., BLAS) are performed on the whole block. At each node of the pruned tree, operations are performed on the B columns of the block (B being the block size). This means that we operate on the union of the structures of the solution vectors and that computations are performed on some explicit zeros (that we refer to as *padded zeros* in the following, as well as in [18]). A large block size will be beneficial for using BLAS operations but may involve more arithmetic operations and require more storage.

PROPERTY 1 (block size and minimum number of operations). *Given the factorization of a matrix A, the block size B leading to the minimum number of operations to compute a subset of diagonal entries of A^{-1} is $B = 1$; that is, we compute diagonal entries one at a time on the pruned tree defined by Theorem* 2.1.

Proof. By definition of the pruned tree resulting from the application of Theorem 2.1, all nodes in the pruned tree are involved in the computation of the entry concerned. Furthermore, since entries are processed one at a time, no padded zeros can be introduced so that the number of operations to compute each entry is minimal, and so the total number of operations to compute the subset of entries is also minimal. ∎

3.3. Parallel efficiency. In many sparse direct codes, mapping algorithms usually identify a set of sequential tasks (subtrees that will be processed by a single process), relying, for example, on the Geist–Ng algorithm [9]. We call the set of roots of the sequential subtrees *layer* L_0. Nodes in the upper part of the tree (i.e., above sequential subtrees) are mapped onto several processes; one of these processes is called the *master process* and is in charge of organizing the tasks corresponding to the node. In our initial computational setting resulting from the work described in [2], the right-hand sides e_j are processed following an order which tends to put together nodes which are close in the assembly tree. This is true of an ordering of the right-hand sides based on Definition 2.3, for example. Note that such orderings, as explained in more detail later, exploit locality in the assembly tree; they also limit the number of padded zeros resulting from Assumption 2 when $B > 1$ and thus the number of additional operations. However, as a consequence of locality in the assembly tree, usually only a few processes (probably only one) will be active in the lower part of the tree when processing a block of right-hand sides. A natural way of involving more processes is to interleave the right-hand sides so that every process will be active when a block of entries is computed. We describe a straightforward algorithm in Algorithm 1 and use it to explain why parallelism and the reduction in the number of operations are conflicting objectives. Modifications to this algorithm together with alternative strategies, in particular for the management of parallel nodes (nodes that are processed by more than one process), will be described later.

Algorithm 1. Interleaving algorithm.

```
Input: old_rhs: preordered list of requested diagonal entries
           node-to-master process mapping
Output: new_rhs: the interleaved list of entries

The current process is chosen arbitrarily.
while old_rhs has not been completely traversed do
  Look for the next entry in old_rhs corresponding to a node mapped on the current
  process
  if an entry has been found then
    add the entry to new_rhs
  end if
  change current process (cyclicly)
end while
```

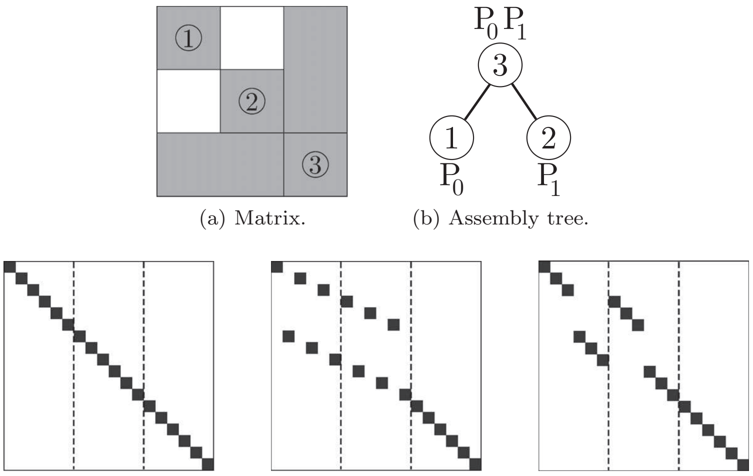
We illustrate the problems raised by this approach in the following (archetypal) example, illustrated in Figure 2.

We assume that all the diagonal entries of A^{-1} are requested and that the block size B is $N/3$, where N is the order of the matrix A. The right-hand sides are processed following a postorder (see Definition 2.3) shown in Figure 2c, which is straightforward here. Let us examine different situations:

1 *process*: in this example, the postorder is optimal since no padded zeros are introduced; thus the number of operations is equal to the number of operations performed to process one right-hand side at a time, which is minimal (Property 1).

2 *processes*: nodes 1 and 2 are mapped on processes P_0 and P_1, respectively, and node 3 is mapped on P_0 *and* P_1.

Without interleaving: when processing the first block (columns 1 to $N/3$ of the right-hand sides shown in Figure 2c), only nodes 1 and 3 are traversed. Thus, at the bottom of the tree (node 1), only one process, P_0, is active. Similarly, when processing the second block, only P_1 is active



(a) Matrix. (b) Assembly tree.

(c) Postordered RHS. (d) Interleaved RHS. (e) Interleaved RHS + postordering within blocks.

FIG. 2. *Archetypal example with different orderings for the right-hand sides (RHS).*

at the bottom of the tree.

With interleaving: right-hand sides are permuted so that the first two blocks each contain $N/6$ columns whose only nonzero corresponds to a node mapped on P_0 and $N/6$ columns whose only nonzero corresponds to a node mapped on P_1 (see Figure 2d or 2e). However, all the $N/3$ columns of the block must be operated on by each node (padded zeros are thus introduced—see Assumption 2), so the operation count is multiplied by 2 and, although two processes are active at the same time, there is no speed-up.

This example illustrates the fact that interleaving is a good way to put all processes to work but tends to destroy the benefits of the postordering (or any other permutation aimed at reducing the number of operations). A trade-off between the number of operations and parallelism (i.e., performing activities that involve as many processes as possible) thus has to be found.

4. Description of the proposed approach.

4.1. Core idea. The main problem in combining interleaving (for parallelism) with a blocking scheme based on postorder is that the postordering groups together nodes that are close in the tree while interleaving does exactly the opposite. Indeed, since mapping algorithms look for locality to limit the number of nodes visited and minimum granularity in order to achieve performance, two nodes that are close in the tree are likely to be mapped on the same process, significantly limiting parallelism. Interleaving tends to cancel the benefits of a good permutation for sequential computation. It increases the number of accesses/operations because it combines activities which correspond to distant branches/parts of the assembly tree. The problem is that a large block of right-hand-side columns chosen so that parallelism can be exploited will result in some processes doing far more operations because we regard the block as being indivisible. We still want to exploit the benefits of BLAS/dense computations but not at the expense of too many unnecessary arithmetic operations. In order to do so, we must work around Assumption 2.

Definition 4.1 (active variables at a node). *A variable i is said to be active at a node n if and only if n is on the path of the assembly tree between the node associated with variable i and the root of the assembly tree.*

Property 2 (exploiting active variables to minimize the number of operations). *Given a block of B right-hand sides and its associated pruned tree (Definition 2.2), if, for each node of the pruned tree, computation is limited to columns of the right-hand sides associated with active variables at that node, then the number of operations performed is minimal. This is true for both the forward and the backward substitution steps (equations (2.1)).*

Proof. This property is a natural generalization of Property 1 to a block since for a given column of the right-hand side and thus a given target diagonal entry i of A^{-1}, the set of nodes on which variable i is active is exactly the set of nodes of the pruned tree for variable i as defined by Theorem 2.1. Property 1 means that, for each column, the number of operations is minimal, and thus the number of operations to process the block of size B is also minimal. ▯

The block on which we do our computations is as small as it can be, and so we are as efficient in terms of operation count as possible. Thus, at the leaf nodes, the block of computations will normally be quite small, corresponding only to entries present at that node. However, because of tree pruning, at least one right-hand side will be operated on at every leaf node of the pruned tree. As we progress up the tree, the computational block will increase, with the block at any node being the union of the blocks at the children together with any new entries appearing at the node. At the root node (assuming irreducibility) the block will be of size B.

Property 2 is, however, not sufficient to design an efficient algorithm since we also need the columns associated with active variables to be contiguous (to be an interval) to fully benefit from the efficiency of Level 3 BLAS kernels. As we see in Figure 2d, this may not be the case. However, after reordering the columns within each block as in Figure 2e, active variables at each node effectively correspond to contiguous columns. The theorem below states that this is always possible.

Theorem 4.2. *Given a block of B right-hand sides and its associated set of target diagonal entries of A^{-1}, if the target entries (and thus the columns of the block) are reordered to follow a postorder with respect to the pruned tree of that block, then the subset of columns defined by Property 2 at each node of the pruned tree corresponds to contiguous columns of the permuted right-hand sides.*

Proof. By definition of the pruned tree each leaf node must be associated with at least one target diagonal entry and thus one column of the right-hand side. Furthermore, all target entries located at a given leaf node of the pruned tree must be contiguous columns of the permuted right-hand sides because of the postordering. Thus, at each leaf node of the pruned tree, we operate on an interval of columns of the permuted right-hand sides. For any nonleaf node n in the pruned tree, the active variables of n can be viewed, by definition of an active variable, as the union of the set of target variables on n with the set of active variables of each of its children in the pruned tree. If we then assume that the active variables of the descendants are associated with a contiguous set of columns, then, because of the postordering, the union of the set of active variables of the children must correspond to a set of contiguous columns. This set must also be contiguous to the set of columns associated with target variables located at node n. This recursively proves our theorem. ▯

Because of Theorem 4.2 and because the B-sized block of right-hand sides is postordered, the block at any node will always be a contiguous subset of entries from the B-sized block so that only the positions of the first and last entries need to be passed and the merging process at a node is trivial. This property is exploited to have a simple and efficient implementation.

4.2. Sketch of algorithm. Algorithm 2 provides an overall sketch of our algorithm before we examine the constituent parts in more detail. We first focus on the case where only diagonal entries of the inverse are requested; thus the traversal of the tree for the backward substitution is the reverse of that for the forward substitution. We discuss the general case later.

```
Algorithm 2. Computation of a set of diagonal entries with two levels of blocking.
  Input: A distributed factorized sparse matrix.
         Its assembly tree.
         A set of requested diagonal entries.
  Output: Requested diagonal entries

  1. Reorder the set of entries, for example using a permutation aimed at reducing
  operations.
  2. Obtain an interleaving permutation for parallelism (for example, Algorithm 1).
  3. Split the list of requested entries into blocks of size $B$.
  4. Reorder columns within each block by a postordering (see Definition 2.3 and
  Theorem 4.2).
  5. For each $B$-sized block of right-hand sides:
     a. Compute the pruned tree (Definition 2.2).
     b. For each node in the pruned tree, the block of right-hand sides to be operated
  upon is then a contiguous set of columns (Theorem 4.2) defined by Property 2.
     c. Solve for the $B$-sized block of right-hand sides to obtain the requested entries.
```

4.3. Interleaving algorithm. We now address two issues related to the interleaving process described in Algorithm 1.

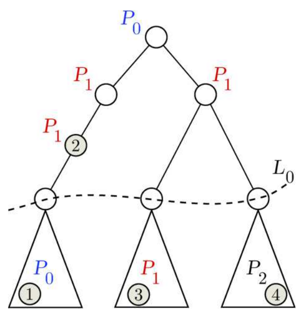
The first issue is related to the way the node-to-process mapping is done, which can strongly influence the behavior of the interleaving algorithm. We illustrate this in Figure 3, where four entries are requested. With a block size of 2, Algorithm 1 yields the partitioning $\{\{1,2\},\{4,3\}\}$. This gives poor parallelism in the first block since nodes corresponding to entries 1 and 2 are processed one after the other (the computation of the block involves two processes, but they are not active at the same time).

We suggest the following strategy. First do interleaving on the lower part of the tree (sequential subtrees at layer L_0) and then on the upper part. In the example of Figure 3, the partitioning then becomes $\{\{1,3\},\{4,2\}\}$, which provides better parallelism since two processes are active *at the same time* within each block.

We now address the issue of managing the parallel nodes when interleaving the requested entries over the processes. We note that we did not consider these nodes in Algorithm 1. Our strategy is to store the load on each process (that is, the number of entries selected on the process) and, when a node involving more than one process is encountered, the following occur:

1. The load on all the processes concerned with this node is updated.
2. The least loaded process (among all processes) becomes the current process.

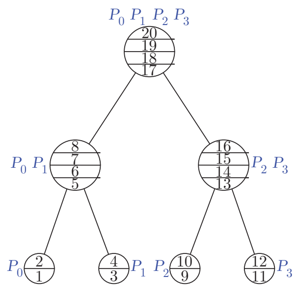
We expect this strategy to provide a fair load balancing. Note that, in this case, a



Column indices of requested entries in postorder:
| 1 2 3 4 |

Interleaving:
| 1 3 4 2 |

FIG. 3. *Adapting the interleaving procedure to exploit the L_0 resulting from Geist–Ng algorithm* [9]*. The column indices of the requested entries are indicated with shaded nodes.*



Column indices of requested entries in postorder:
|1 2 3 4|5 6 7 8|9 10 11 12|13 14 15 16|17 18 19 20|

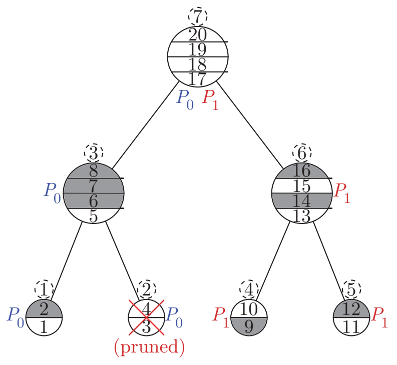
Interleaving:
|1 3 9 11|2 4 10 12|5 13 6 14| 7 15 8 16|17 18 19 20|

FIG. 4. *Adapting the interleaving procedure to manage parallel nodes. The three top level nodes are mapped on several processes, and the interleaving procedure enforces some load balancing. In this example, all the diagonal entries of the inverse are requested.*

complete mapping has to be provided, that is, for each node, the master and the list of processes that take part in the processing of the node. We illustrate this strategy in Figure 4.

4.4. Detecting and exploiting sparsity between subblocks. We now turn our attention to the fifth step in the sketch of Algorithm 2. For each node of the tree, we have to specify the subblock of right-hand sides on which the computations will be performed. Since each B-sized block of right-hand sides is postordered (step 4 of Algorithm 2), Theorem 4.2 implies that the set of columns being processed at a node is a contiguous subset of consecutive columns of the right-hand side. The proof of Theorem 4.2 also explains the recursive property of the intervals and thus defines how we can easily compute the interval of columns for each node of the pruned tree. If we assume that nodes of the pruned tree are processed in a topological order, then the subblock for each node can be computed as the union of its own subblock and the subblocks of its children. Since we are working on the pruned tree, every leaf corresponds to at least one requested entry and has an initialized set, thus ensuring that all sets are well defined.

We illustrate by a small example how sparsity can be exploited within the B-sized block of columns. In Figure 5 we show an assembly tree on seven nodes for a matrix of order 20. This tree is mapped onto two processes. The requested diagonal entries of the inverse are in grey. Note that one of the nodes (node 2) is not involved in the computation of any of these entries and so will be pruned, independently of the block size and the permutation of the right-hand sides.



Column indices of requested entries in postorder:

$\vert$ 2 6 7 8 9 12 14 16 $\vert$

Interleaving of entries, postordering of each block:

$\vert$ 2 6 9 12 $\vert$ 7 8 14 16 $\vert$

Interval of each node for the first block:

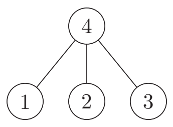
Node	Interval
1	[1 1]
2	$\varnothing$
3	$[2\ 2] \cup [1\ 1] \cup \varnothing = [1\ 2]$
4	[3 3]
5	[4 4]
6	$\varnothing \cup [3\ 3] \cup [4\ 4] = [3\ 4]$
7	$\varnothing \cup [1\ 2] \cup [3\ 4] = [1\ 4]$

FIG. 5. *Example of blocks with $B = 4$. The numbering for the nodes of the assembly tree is shown in dashed circles.*

We first reorder the right-hand sides using a postordering and then apply interleaving, before postordering the columns within each B-sized block. Then we process each block (of size $B = 4$ in the example) of the “interleaved” right-hand sides consecutively. We compute the pruned tree of that block (Definition 2.2), and, for each node of the pruned tree, we find which columns of the B-sized block will be operated on. We illustrate the algorithm on the first block (entries 2, 6, 9, 12):

- We first compute the subblocks at the leaf nodes of the pruned tree:
  - Node 1 has to process entry 2 (first column of the block); the interval is thus [1 1].
  - Node 2 was pruned.
  - Node 4 has to process the third column of the B-sized block because that column corresponds to entry 9, which is active at node 4. Its interval is thus [3 3].
  - Node 5 has to process entry 12, and thus its interval is [4 4].
- The other nodes are processed with a topological ordering. Each node then also has to include the columns processed by its children, which are associated with active variables at that node by Definition 4.1 and are thus processed at that node (Property 2).
  - Node 3 has to process entry 6 (column 2) and has one child in the pruned tree. Its subblock is an interval (Theorem 4.2) defined as $[2\ 2] \cup [1\ 1] = [1\ 2]$.
  - Node 6 has an empty set of entries. Its interval is thus $\varnothing \cup [3\ 3] \cup [4\ 4] = [3\ 4]$.
  - Finally the interval of node 7 is $\varnothing \cup [1\ 2] \cup [3\ 4] = [1\ 4]$, which is the full B-sized block because the pruned tree at that block has only one root.

In an earlier version of our algorithm, we requested a minimum size of computational block (that we called B_{sparse}) so that the block size at any node was a multiple of B_{sparse} although the contiguity property was still maintained. However, although this was attractive because of specifying minimum computational units for the BLAS, it means that we were often doing many unnecessary operations. With our present approach, as we progress up the tree, we will have bigger blocks when it is more efficient to use the BLAS, for example if we wish to exploit parallel (multithreaded) BLAS on nodes that are assigned to more than one process.



(a) Assembly tree.

$$\begin{array}{r|ccc|}
 & 1 & 2 & 3 \\
\hline
1: & X & 0 & X \\
2: & X & X & 0 \\
3: & 0 & X & X \\
4: & 0 & 0 & 0 \\
\hline
\end{array}$$

(b) Structure of the solution (column 4 is empty).

Node	Interval
1	$\{1, 3\} \rightarrow [1\ 3]$
2	[1 2]
3	[2 3]
4	[1 3]

(c) Intervals for the backward phase.

FIG. 6. *Example of assembly tree* (a) *where node numbers are identical to variable indices in a matrix of order* 4*, with one diagonal and the last row/column full. The structure of the solution* (b) *corresponds to the target entries of A^{-1}. The intervals associated with the backward phase* (c) *are shown for $B = 3$ in the case where the permutation is the natural order* $\{1, 2, 3\}$.

5. General case.

In the previous sections, we assumed that only diagonal entries were requested. In the general case where the right-hand sides or solution vectors do not have a single nonzero entry, we cannot necessarily permute the blocks in order to guarantee that during the backward phase, the sets of columns to be processed at each node will be contiguous. In this case, in order to keep the implementation efficient, the set of columns to be processed at each node is defined by the interval that borders that set. This introduces some explicit zeros (padded zeros, as described in section 3) and increases the number of operations (although it is still reduced compared to the case where sparsity is not exploited within each block). However, it avoids the use of indirect addressing and lists. The way the columns are ordered will influence the number of padded zeros. We assume that a “good” ordering (see [2]) of the columns is provided in step 1 of Algorithm 2.

We provide an example. Assume that one has to compute $a_{11}^{-1}, a_{21}^{-1}, a_{22}^{-1}, a_{32}^{-1}, a_{13}^{-1}$, and a_{33}^{-1} using the assembly tree in Figure 6, with $B = 3$. This means that, in the forward phase, $Lx = [e_1\ e_2\ e_3]$ is solved. Then, in the backward phase where we are computing the six requested entries of the inverse, the structure of the solution vectors (components to be computed) is as shown in Figure 6b. Irrespective of the permutation of the right-hand sides, one of the nodes has to process a discontiguous set of columns of the right-hand sides. For example, if we process the right-hand sides in their original order, node 1 has to process columns 1 and 3 since they belong to the same target row (and node) 1 (see Theorem 2.1). In our strategy, node 1 is provided with the interval $[1, 3]$, which bounds the necessary set $\{1, 3\}$ but includes some operations on a padded zero performed in column 2 (which of course does not belong to node 1).

To further improve efficiency when off-diagonal entries of A^{-1} are requested one could permute the columns of the current block again before the backward solution to better match a postorder on the target row indices. When multiple entries need to be computed in a column of A^{-1}, one may then introduce a fixed size subgrouping of the block of noncontiguous columns to limit the number of padded zeros while preserving the efficiency of dense kernels.

6. Experiments.

We carried our experiments on three different systems:

- *Hyperion*, Altix ICE 8200 system at the Centre Interuniversitaire de Calcul de Toulouse (CICT). Each node has two quad-core 2.8 GHz Intel Xeon 5560 processors and 32 GB of main memory.
- *Hopper*, Cray XE6 system at the National Energy Research Scientific Computing Center (NERSC). Each node has two twelve-core 2.1 GHz AMD

TABLE 1

Set of matrices used for the experiments. Those marked with $^{(*)}$ *are publicly available in the University of Florida Sparse Matrix Collection.*

Matrix name	Order N	Entries (millions)	Flops fact. ($\times 10^{12}$)	Factors (GB)	Description, origin, and type
af_shell1$^{(*)}$	504,855	17.6	7.9	0.6	Semiconductor metal forming; University of Basel; double real
pre2$^{(*)}$	659,033	5.8	0.3	1.0	Nonlinear circuit simulation; AT&T; double real
NICE20MC	715,923	28.1	5.0	8.3	Seismic processing; BRGM lab; double real
audikw_1$^{(*)}$	943,695	39.3	6.0	9.9	Automotive crankshaft model; Parasol collection; double real
bone010$^{(*)}$	986,703	36.3	4.1	8.6	3D trabecular bone; Mathematical Research Institute of Oberwolfach; double real
CONESHL	1,262,212	43.0	1.8	5.5	3D finite element, SAMCEF code; SAMTECH; double real
atmosmodd$^{(*)}$	1,270,432	8.8	0.6	8.6	Atmospheric modeling; Pelotas State University; double real
Hook_1498$^{(*)}$	1,498,023	31.2	9.6	12.3	3D model of a steel hook; Padova University; double real
Transport$^{(*)}$	1,602,111	23.5	10.9	20.0	3D flow and transport problem; Padova University; double real
CAS4R_LR15	2,423,135	19.6	0.6	4.5	3D electromagnetism; EADS Innovation Works; single complex

Opteron 6172 processors and 32 GB of main memory.

- *Edison*, Cray XC30 at NERSC. Each node has two twelve-core 2.4 GHz Intel Xeon E5-2695 v2 processors and 64 GB of main memory.

The set of matrices used in our experiments is described in Table 1. They correspond to large problems arising from industrial or academic applications. We also use a generator that creates finite-difference matrices for 7-pt discretizations of 3D regular grids.

We use a modified version of the general purpose distributed memory solver MUMPS [1, 3] to implement the algorithms described in this paper.

6.1. Sequential case. Exploiting sparsity within blocks enables us to decrease the computational cost, and this is beneficial not only in a parallel setting but also for serial execution. In this section, we report on experiments in a sequential context on the 10 matrices of Table 1. In all the cases, 10% of the diagonal entries of the inverse chosen randomly are computed; the block size is $B = 1024$. Experiments were performed on one node (i.e., 8 cores) of the above mentioned Hyperion system, using 8-way multithreaded BLAS and one MPI process. We show, in Table 2, the time and the number of operations for the solution phase, with (columns "w/ ES") and without (columns "w/o ES") exploiting sparsity within each block.

As expected, exploiting sparsity within each block of right-hand sides decreases the number of operations and reduces execution time. We see that the time improvement is a superlinear function of the number of operations. We believe that this is because much of the reduction in number of operations is for nodes near the bottom of the tree, where the GFlops/s rates are much less than near the top of the tree. Note that when exploiting sparsity within the blocks, we attain a speed of 49 GFlops/s,

TABLE 2

Influence of exploiting sparsity ("ES") within each block of right-hand sides. 10% of the diagonal inverse entries are computed using the sparse inverse functionality in MUMPS, with $B = 1024$. $^{(*)}$: Operations on complex numbers.

	Operations ($\times 10^{12}$)		Time (s)	
Matrix	w/o ES within blocks	w/ ES within blocks	w/o ES within blocks	w/ ES within blocks
af_shell1	17.2	9.3	561	228
pre2	69.3	37.5	2707	1200
audikw_1	42.2	36.9	1385	985
NICE20MC	35.4	31.1	1137	817
bone010	33.3	28.7	1213	719
CONESHL	34.1	31.3	1285	808
atmosdd	120.1	82.6	2448	1513
Hook_1498	111.2	104.9	2807	2141
Transport	117.1	60.0	2541	1262
CAS4R_LR15	$15.0^{(*)}$	$12.9^{(*)}$	1144	582

which exceeds 50% of the peak of `DGEMM` on this system. This is excellent for a sparse triangular solution, especially since we exploit sparsity at many different levels (in the factors, in the right-hand sides, and between columns of the right-hand sides).

6.2. Parallel case.

6.2.1. Assessing the strategies. We first illustrate in Table 3 the influence of the different strategies (interleaving and exploiting sparsity within blocks) on a large size problem (7-pt stencil discretization of a $225 \times 225 \times 225$ domain), using 16 nodes of the Edison system described above, with single-threaded BLAS. We show the time for the solution phase and the operation count when the number of MPI processes ranges from 32 to 256. First, we see that using the baseline strategy (i.e., neither interleaving nor exploiting sparsity within blocks), the parallel efficiency is low (for example, the speed-up from 32 to 256 MPI is $3673/2296 = 1.6$). When sparsity is exploited within blocks, the number of operations slightly decreases, but the speed-up does not improve ($2707/1679 = 1.6$). This confirms that exploiting sparsity within blocks saves operations but does not improve parallelism. If the interleaving procedure is activated but sparsity is not exploited within blocks (second to last row of the table), the operation count increases significantly (e.g., it is multiplied by more than 3 on 256 processes); speed-ups are better (almost 2 from 32 to 256 processes), but absolute times are much higher. This suggests that interleaving allows for a better parallelism but is penalized by the increased number of operations. As shown in the previous section, finding a good permutation of the requested entries is not enough to achieve good performance, contrary to what was done in our previous contributions [2, 18]. Finally, when *both* interleaving and sparsity within blocks are enabled, *both* the run times and the speed-ups significantly improve. With 256 processes, the run time decreases from 2296 seconds (baseline version as of MUMPS 4.10.0, i.e., no interleaving and no sparsity within blocks) to 1003 seconds; this is more than a 50% improvement.

We show in Table 4 experimental results using four nodes of the Hyperion system on the 10 different matrices from practical applications given in Table 1. Here we simply compare the baseline strategy with the new one where both interleaving and exploiting sparsity within blocks are enabled. The new strategy is significantly better

TABLE 3

Computation of a random set of diagonal entries of the inverse of a matrix corresponding to a 7-pt stencil discretization of a $225 \times 225 \times 225$ domain on Edison. 0.1% of the diagonal inverse entries are computed with $B = 512$, except for 256 processes for which $B = 1024$. "IL" stands for the interleaving strategy and "ES" for exploiting sparsity within blocks of right-hand sides.

Strategy		Time (seconds)				Operations ($\times 10^{14}$)			
	Processes	32	64	128	256	32	64	128	256
IL off	ES off	3673	2833	2448	2296	2.3	2.3	2.3	2.9
	ES on	2707	2256	1954	1679	2.0	2.0	2.0	2.0
IL on	ES off	4507	2992	2568	2284	8.3	8.8	9.4	10.0
	ES on	1850	1560	1398	1003	2.0	2.0	2.0	2.0

than the baseline algorithm: on matrix NICE20MC, the time is reduced by almost a factor of 6 and the overall speed-up on 32 processes is very good, especially as we are considering the solution phase. Additionally, this is particularly impressive given that sparsity is exploited at many different levels.

TABLE 4

Time in seconds for the computation of a random 1% of the diagonal entries of the inverse of 10 large matrices from our experimental set. The block size, B, is 1024. We compare the sequential performance with the parallel performance on 32 MPI processes (4 nodes of the Hyperion system), using the baseline algorithm and the combination of interleaving ("IL") and sparsity within blocks ("ES").

Matrix	Processes		
	1	32	
		Baseline Time(s)	IL+ES Time(s)
af_shell1	149	102	24
pre2	722	514	240
audikw_1	1143	380	75
NICE20MC	945	245	43
bone010	922	327	70
CONESHL	803	293	48
atmosmodd	652	276	84
Hook_1498	1860	720	173
pre2	677	386	101
CAS4R_LR15	882	482	116

6.2.2. Influence of the block size. In Table 5, we use the Hopper system described above, and the problem is the 7-pt stencil discretization of a $110 \times 110 \times 110$ domain. This is the largest problem (grid) that could fit on one node of the system. We illustrate how the block size influences the performance of the computation of requested entries of the inverse for 1 and 64 MPI processes.

In the sequential case, increasing the block size B from 64 to 128 decreases the time for the solution phase, which is probably due to the efficiency of the BLAS on larger blocks. However, when increasing B from 128 to 256 and then to 512 and 1024, the efficiency of the BLAS cannot compensate for the fact that the operation count increases as a function of B; therefore, the solution time increases. However, when exploiting sparsity within blocks, the operation count does not depend on B; therefore, on 64 processes, when right-hand sides are interleaved and sparsity is exploited within blocks, the time for the solution phase decreases as a function of B up to a block size of 512. If we compare the best original sequential time (1934.2 s with IL off and ES off) with the best parallel time (206.2 s for IL on ES on), the speed-up is almost 10 instead of less than 4 with the baseline strategy ($\frac{1982.3}{552.8}$). As explained earlier, and as fully detailed in this example, this gain comes from two effects, exploitation of sparsity and interleaving within each block.

Table 5
Influence of the block size: a random 1% of the diagonal entries of the inverse of a matrix corresponding to a 7-pt stencil discretization of a $110 \times 110 \times 110$ domain are computed. The influence of the block size B on the time for the solution phase (indicated in seconds, best in bold) is illustrated for both sequential and parallel executions, with interleaving ("IL") and exploiting sparsity within blocks ("ES"), on one node of the Hopper system.

	Strategy		Time (seconds)					Operations ($\times 10^{12}$)				
Procs	Blocksize $B =$		64	128	256	512	1024	64	128	256	512	1024
1	ES off		2005.6	1934.2	**1982.3**	2310.2	2971.6	13.2	13.5	14.0	15.0	17.1
	ES on		1966.0	1836.5	**1790.8**	1831.8	1850.7	12.9	12.9	12.9	12.9	12.9
64	IL off	ES off	556.8	**552.8**	585.0	683.2	812.3	13.1	13.4	13.9	14.8	17.1
		ES on	540.8	**519.5**	526.2	543.9	549.0	12.9	12.9	12.9	12.9	12.9
	IL on	ES off	652.8	**630.2**	650.7	733.7	800.7	52.0	52.2	52.5	53.0	54.0
		ES on	267.6	241.1	232.8	**206.2**	222.0	12.9	12.9	12.9	12.9	12.9

7. Conclusions. We have shown how it is possible to balance the conflicting requirements of arithmetic efficiency and parallelism when computing inverse entries of large sparse matrices. To do this we had to use a novel interleaving algorithm in addition to a standard postordering to decide which entries to group together. We also had to exploit sparsity within the resulting blocks of right-hand sides.

The effect of these two strategies leads to a significantly improved sparse triangular solution phase for both sequential and parallel environments. In parallel environments, we note that the mapping of the factors is currently inherited from the factorization phase, possibly leading to suboptimal speed-ups (for both sparse and dense right-hand sides).

Finally, exploiting sparsity within blocks of right-hand sides is an important strategy in general, not just for the computation of entries of A^{-1}. It can be beneficial in applications with multiple sparse right-hand sides and in applications where only part of the solution is requested.

REFERENCES

[1] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM J. Matrix Anal. Appl., 23 (2001), pp. 15–41.

[2] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, Y. Robert, F.-H. Rouet, and B. Uçar, *On computing inverse entries of a sparse matrix in an out-of-core environment*, SIAM J. Sci. Comput., 34 (2012), pp. A1975–A1999.

[3] 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.

[4] Å. Björck, *Numerical Methods for Least Squares Problems*, SIAM, Philadelphia, 1996.

[5] L. Bouchet, J. Roques, P. Mandrou, A. Strong, R. Diehl, F. Lebrun, and R. Terrier, *INTEGRAL SPI observation of the Galactic central radian: Contribution of discrete sources and implication for the diffuse emission* 1, Astrophys. J., 635 (2005), pp. 1103–1115.

[6] Y. Campbell and T. Davis, *Computing the Sparse Inverse Subset: An Inverse Multifrontal Approach*, Technical report TR-95-021, Computer and Information Sciences Department, University of Florida, Gainesville, FL, 1995.

[7] Y. Campbell and T. Davis, *A Parallel Implementation of the Block-Partitioned Inverse Multifrontal Zsparse Algorithm*, Technical report TR-95-023, Computer and Information Sciences Department, University of Florida, Gainesville, FL, 1995.

[8] S. Cauley, J. Jain, C. K. Koh, and V. Balakrishnan, *A scalable distributed method for quantum-scale device simulation*, J. Appl. Phys., 101 (2007), 123715.

[9] G. Geist and E. Ng, *Task scheduling for parallel sparse Cholesky factorization*, Int. J. Parallel Program., 18 (1989), pp. 291–314.

[10] L. Lin, J. Lu, L. Ying, R. Car, and W. E, *Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems*, Commun. Math. Sci., 7 (2009), pp. 755–777.

[11] L. Lin, C. Yang, J. Lu, L. Ying, and W. E, *A fast parallel algorithm for selected inversion of structured sparse matrices with application to 2D electronic structure calculations*, SIAM J. Sci. Comput., 33 (2011), pp. 1329–1351.

[12] M. Luisier, A. Schenk, W. Fichtner, and G. Klimeck, *Atomistic simulation of nanowires in the $sp^3d^5s^*$ tight-binding formalism: From boundary conditions to strain calculations*, Phys. Rev. B, 74 (2006), 205323.

[13] J. Rommes and W. H. A. Schilders, *Efficient methods for large resistor networks*, IEEE Trans. Comput. Aided Des. Integrated Circ. Syst., 29 (2010), pp. 28–39.

[14] F.-H. Rouet, *Memory and Performance Issues in Parallel Multifrontal Factorizations and Triangular Solutions with Sparse Right-Hand Sides*, Ph.D. thesis, Institut National Polytechnique de Toulouse, Toulouse, France, 2012.

[15] Tz. Slavova, *Parallel Triangular Solution in the Out-of-Core Multifrontal Approach for Solving Large Sparse Linear Systems*, Ph.D. thesis, Institut National Polytechnique de Toulouse, Toulouse, France, 2009.

[16] K. Takahashi, J. Fagan, and M. Chin, *Formation of a sparse bus impedance matrix and its application to short circuit study*, in Proceedings of the 8th PICA Conference, Minneapolis, MN, 1973.

[17] J. M. Tang and Y. Saad, *A probing method for computing the diagonal of a matrix inverse*, Numer. Linear Algebra Appl., 19 (2012), pp. 485–501.

[18] I. Yamazaki, X. S. Li, F.-H. Rouet, and B. Uçar, *On partitioning and reordering problems in a hierarchically parallel hybrid linear solver*, in PDSEC Workshop of the IEEE, International Parallel and Distributed Processing Symposium, 2013.