Parallel computation of entries of A-1

Patrick Amestoy, Iain S. Duff, Jean-Yves L'Excellent, François-Henry Rouet

Parallel computation of entries of A^{-1}

Patrick R. Amestoy, Iain S. Duff , Jean-Yves L'Excellent, François-Henry Rouet

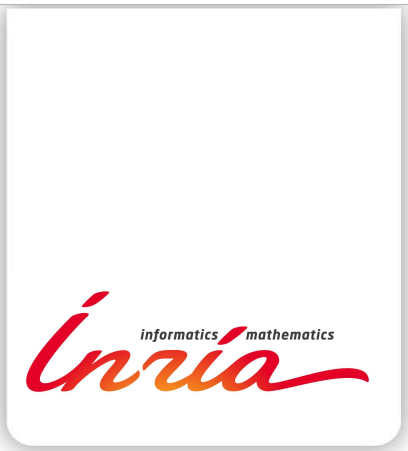
RESEARCH REPORT

N° 8142

December 2012

Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--8142--FR+ENG



Parallel computation of entries of A^{-1}*

Patrick R. Amestoy†, Iain S. Duff‡ §, Jean-Yves L'Excellent¶,
François-Henry Rouet†

Project-Team ROMA

Research Report n° 8142 — December 2012 — 16 pages

Abstract: In this paper, we are concerned about computing in parallel 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 efficacy by runs using the MUMPS code that implements a parallel multifrontal method.

Key-words: sparse matrices, matrix inverse, direct methods, direct solver, parallelism

* Also available as CERFACS, ENSEEIHT-IRIT, and RAL reports

† Université de Toulouse, INPT(ENSEEIHT)-IRIT, France (`{amestoy,frouet}@enseeiht.fr`).
‡ CERFACS, 42 Avenue Gaspard Coriolis, 31057, Toulouse, France (`duff@cerfacs.fr`).
§ R 18, RAL, Oxon, OX11 0QX, England (`iain.duff@stfc.ac.uk`).
¶ INRIA and Université de Lyon, Laboratoire LIP (UMR 5668 – CNRS, ENS Lyon, INRIA, UCBL), France (`jean-yves.l.excellent@ens-lyon.fr`).

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Calcul d'entrées de A^{-1} en parallèle

Résumé : Nous considérons le calcul en parallèle de plusieurs entrées de l'inverse d'une matrice creuse de grande taille. Nous supposons que la matrice a été factorisée à l'aide d'une méthode directe et que les facteurs creux de la matrice sont distribués sur les processeurs. Les entrées de l'inverse peuvent être calculées efficacement en exploitant le fait que les vecteurs constituant les seconds membres et la solution sont creux. Nous montrons que dans ce contexte, le parallélisme et l'efficacité du calcul sont deux objectifs contradictoires. Nous développons une approche efficace et montrons son intérêt grâce à des tests utilisant le code MUMPS, qui implante une méthode multifrontale parallèle pour machines à mémoire distribuée.

Mots-clés : matrice inverse, matrices creuses, méthodes directes, parallélisme

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 computed. We are concerned about the efficient computation of the inverse entries by forward and backward substitution using the matrix factors. In [2] we examined the case when the factors are held out-of-core with consequent high cost if they must be read repeatedly. In that case we were concerned with combinatorial aspects of grouping or blocking the right-hand sides corresponding to the requested inverse entries.

In this paper, we examine the computation of inverse entries in parallel using a distributed sparse 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 and show their effectiveness both on small examples and on 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.

Quite recently, 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.

Far less work has been done on this when factors are available. Other work that we are aware of makes use of the identity of Takahashi et al [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 provide the framework in which we are working and define the basic algorithm that we implement. At first glance, performing substitutions simultaneously on blocks of vectors appears 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 the requirement to maintain 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.

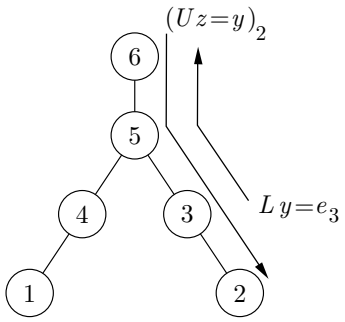


Figure 1: Computing a_{23}^{-1} requires traversing the path from node 3 to the root node, then the path from the root node to node 2. Nodes 1 and 4 can be pruned from the tree.

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 processors to perform the elimination at a node nearer the root so that we can obtain a better memory balance by distributing the factors. As a shorthand, we will often refer to such a node as a "parallel node". For these nodes we identify a single master processor and a set of other processors that help with the factorization at the node.

2.2 Sparse inverse computation

We use the notation a_{ij}^{-1} to denote the (i,j)-th entry of the inverse of A. The approach we use to compute the entries of the inverse relies on the usual solution of equations and uses the equation $AA^{-1} = I$. More specifically, we compute a particular entry 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. If we have an LU factorization of A, a_{ij}^{-1} is obtained by solving successively the two triangular systems:

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

We see from the equations (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 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. 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, where nodes are associated with a set of fully-summed variables, entry a_{ij}^{-1} is 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 are 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 the equations (1) then becomes prohibitive and so the computation must proceed in blocks, where for each block a limited number of entries are computed. This entails accessing some parts of L and U multiple times in different blocks according to the entries computed in the corresponding blocks. In [2], we consider the combinatorial problem of partitioning the requested entries into blocks to minimize the overall cost in an out-of-core environment although we also present some issues related to the in-core case; more detail about the in-core case can be found in [14]. In this paper, we are concerned with how to compute such blocks of entries efficiently in parallel in an in-core setting.

We can extend the concept of tree pruning to apply to these blocks of right-hand sides. In this case the pruned tree will be effectively the union of all pruned trees corresponding to each right-hand side in the block.

3 Parallel and sparsity issues

3.1 The difficulty of computing blocks of 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, 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, we would like to run parallel instances of the linear solver in parallel, each instance using the whole set of processors 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. A potential workaround would be to replicate the factors on all processors, 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; 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 are 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 neither exploited 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, similar to that proposed in [2, 15]:

- Sparsity is exploited by blocks: when processing a block of right-hand sides, the tree is pruned specifically for this block, as mentioned in Section 2.2.
- 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). A large blocksize will be beneficial for using BLAS operations but may involve more arithmetic operations and require more storage.

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 unique processor), 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 processors; one of these processors is called the *master processor* and is in charge of organizing the tasks corresponding to the node. In our initial computational setting, the right-hand sides are processed following an order which tends to put together nodes which are close in the assembly tree, e.g. a postorder. As a consequence, few processors (probably only one) will be active in the lower part of tree when processing a block of right-hand sides. A natural way of involving more processors is to interleave the right-hand sides so that every processor will be active when a block of entries is computed. An algorithm to do this is described in Algorithm 1. We propose some modifications and some alternative strategies later, in particular for the management of parallel nodes (nodes that are processed by more than one process).

Algorithm 1 Interleaving algorithm.

Input: old_rhs: preordered list of requested entries (preordering designed to reduce operations)
node-to-master processor mapping
Output: new_rhs: the interleaved list of entries

The current processor 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 processor
  if an entry has been found **then**
    add the entry to new_rhs
  end if
  change current processor (cyclicly)
end while

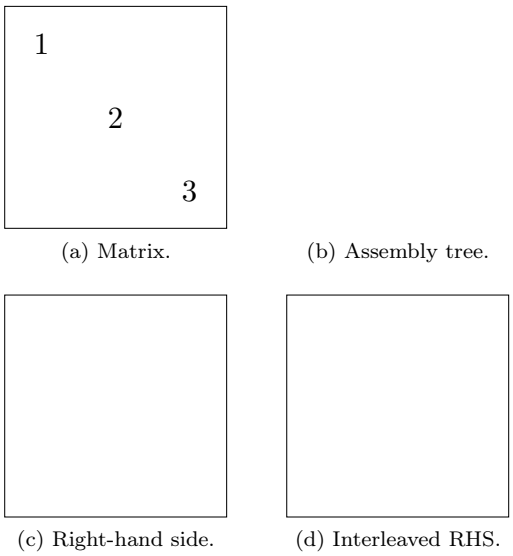
We illustrate the problems raised by this approach on the following (archetypal) example, illustrated in Figure 2.

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 postordering (which is straightforward here). Let us examine different situations:

1 processor: on this example the postorder is optimal with respect to the number of operations.

2 processors: nodes 1 and 2 are mapped on processors 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 processor, P_0, is active. Similarly, when processing the second block, only P_1 is active at the bottom of the tree.
- **With interleaving:** right-hand sides are permuted such that each block contains $N/6$ 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). However, all the $N/3$ columns of the block must be operated on by each node (we do not take advantage of sparsity in the right-hand sides), so the operation count is multiplied by 2 and so is the speed (two processors are active at the same time). Thus, there is no speed-up.



(a) Matrix. (b) Assembly tree.

(c) Right-hand side. (d) Interleaved RHS.

Figure 2: Archetypal example.

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

4 Improving the current approach

Before sketching our overall algorithm for the parallel solution of multiple inverse entries, we first describe the key idea of our strategy. We start with the case where only diagonal entries are computed; we describe the general case in Section 5.

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 and minimum granularity in order to achieve performance, two nodes that are close in the tree are likely to be mapped on the same processor, significantly limiting parallelism within a B-sized block without interleaving. Interleaving tends to cancel the benefits of a good permutation for sequential computation; it increases the number of accesses/operations because it puts together 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 processors 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.

The way we do this is to work at each node only on the right-hand sides from the B right-hand sides currently being processed that are active at that node. By saying that a variable i is active at a node, we mean that the node lies on the path between the node associated with i and the root node. Thus 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 is possible if the sparsity of an individual right-hand side is not exploited (note that it only would be exploited if $B = 1$). Thus, at the leaf nodes, the block of computations will normally be quite small corresponding only to entries present at

that node. However, note that because of tree pruning, there will be at least one right-hand side being 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. One very important feature is that, 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 position of the first and last entries need 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

We give in Algorithm 2 an overall sketch of our algorithm before examining 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: Reordered list of requested diagonal entries

1. Reorder the set of entries for example using a permutation aimed at reducing operations.
2. Obtain an interleaving algorithm for parallelism (perhaps Algorithm 1).
3. Split the list of requested entries into blocks of size B.
4. Reorder each block by a postordering. This will later ensure that when we merge blocks as we process the right-hand sides, we can maintain contiguity and efficient use of the BLAS.
5. Perform the computation for each B-sized block of right-hand sides:
a. For each block, prune the tree.
b. For each node, the block of right-hand sides to be operated upon is determined by the entries from the B-sized block that are involved at that node.
c. For nodes above the leaf nodes, the set of so-called active columns is determined by merging sets from their children and the computational unit will increase until it is of size B at the root (except that the last block of right-hand sides may have fewer entries).

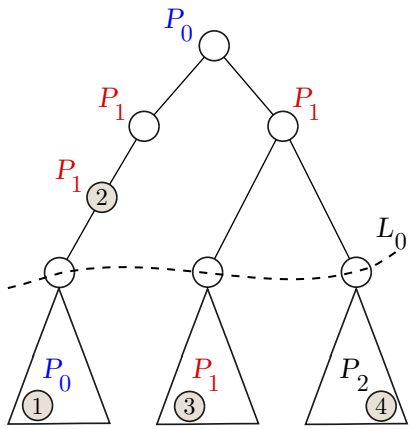
4.3 Interleaving algorithm

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-processor mapping is done, which can strongly influence the behaviour of the interleaving algorithm. We illustrate this idea in Figure 3, where four entries are requested. With a blocksize 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 another (the computation of the block involves two processors but they are not active at the same time).

We suggest the following strategy: perform a pass of interleaving first on the lower part of the tree (sequential subtrees at layer L_0), then on the upper part. On the example of Figure 3, the partitioning becomes {{1,3},{4,2}}, which provides better parallelism since two processors are active *at the same time* within each block.

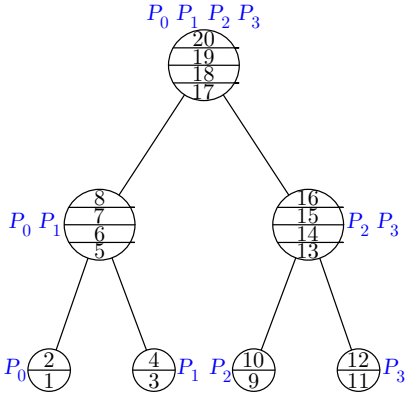
We now address the issue of managing the parallel nodes when interleaving the requested entries over the processors. We note that we did not consider these nodes in Algorithm 1. Our strategy is to store the load on each processor (that is the number of entries selected on the processor), and, when a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



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

Interleaving:
| 1 3 4 2 |

Figure 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 |

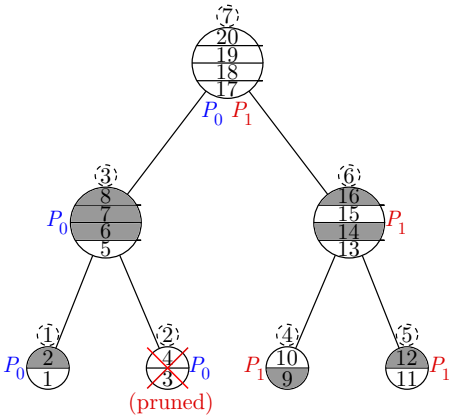
Figure 4: Adapting the interleaving procedure to manage parallel nodes. The 3 top level nodes are mapped on several processors and the interleaving procedure enforces some load balancing. In this example, all the diagonal entries of the inverse are requested.

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

This strategy is expected to provide a fair load balancing. Note that, in this case, a complete mapping has to be provided; that is, for each node, the master and the list of processors 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. In the first step of the algorithm, a tree pruning procedure provides the solve phase with a list of "pruned leaves" (these are the leaves that are present in the pruned tree), and dense computations are performed on the nodes of the pruned tree with the whole block of size B. However, it would be too expensive to compute with this whole block at each node and so we perform operations only on the subblock of entries associated with that node. Therefore, for each node of the tree, we have to specify the subblock of right-hand sides on which the computations will be performed. We use Algorithm 3; blocks can be computed as follows. We work on requested nodes only, that is nodes of the pruned assembly tree containing a variable which is a row or column subscript of a requested entry (for the forward and backward step respectively). The subblock of right-hand sides that we will deal



Column indices of requested entries in postorder:
| 2 6 7 8 9 12 14 16 |
Interleaving of entries, postordering of each block:
| 2 6 9 12 | 7 8 14 16 |
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]

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

with at this node is initialized for each entry of the inverse corresponding to a column index of the nodal matrix.

Algorithm 3 Exploiting sparsity by subblocks of a B-sized block.

We process the pruned tree in topological order. For the leaf nodes, the subblock of columns of the right-hand sides is defined by those entries of the B-sized block that correspond to fully summed variables at the node. For other nodes of the tree, the subblock consists of the right-hand sides associated with the node of the assembly tree together with the columns from its children (that are already computed because of the topological ordering).

Since each B-sized block of right-hand sides is postordered (step 4 of Algorithm 2), the variables of each requested node that are a column index (or row index in the backward case) of a requested entry correspond to consecutive columns of the right-hand side; therefore, each set is an interval. The nodes are processed in a topological order so that the subblock for each node is 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.

By recursion (starting at the leaves), since each block of right-hand sides is postordered, the sets of the children at every node of the pruned tree are consecutive intervals, and thus the union is an interval. Therefore, the block of right-hand sides associated with every node of the pruned tree is an interval of columns from the B-sized block.

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 7 nodes for a matrix of order 20. This tree is mapped onto two processors. 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.

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 want to know, for each node of the tree, which columns of the B-sized block it will process. We illustrate this idea on the first block (entries 2, 6, 9, 12):

1. We first compute the subblocks at the leaf nodes:
   - Entry 2 (first column of the block of right-hand sides) belongs to node 1, hence node 1 has to process the first column of the B-sized block; the interval that is provided to node 1 as output of Algorithm 3 is thus [1 1].

- Node 2 is pruned therefore it does not take part in computations; the set of columns that it has to process is empty.
- 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.
- Node 5 has to process the fourth column of the B-sized block, because that column corresponds to entry 12, which is active at node 5.

2. The other nodes are then processed in topological order: each node has then to include the columns processed by its children, hence we have to augment the set for each node with the sets from its children.

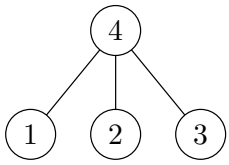
   - Node 3 processes the second column because the corresponding entry (6) is active at that node. Node 3 takes part in the same computations as its children, for instance the first column, because of node 1. Therefore, the set of columns that node 3 processes is $[1\ 1] \cup [2\ 2] = [1\ 2]$.
   - Node 6 takes part in the computation of entries 9 and 12, because of nodes 4 and 5 respectively.
   - Node 7 has an empty set, which is updated by the sets of 3 and 6, and thus performs computations on the full B-sized block.

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 doing totally 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 particularly if we wish to exploit parallel (multithreaded) BLAS on nodes that are assigned to more than one process.

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 the sets of columns to be processed at each node will be contiguous, i.e. intervals. In this case, in order to keep an efficient implementation, 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. As discussed in the previous section, the way the columns are ordered may also influence the number of padded zeros. This is not the purpose of our work in this paper, for which we assume that a "good" ordering (see [2]) of the columns is provided at 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 elimination tree in Figure 6, with $B = 3$. This means that, in the forward phase, $Lx = [e_1\ e_2\ e_3]$ is to be 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 1). In our strategy, node 1 is provided with the interval $[1, 3]$, that bounds the necessary set $\{1, 3\}$; some operations are thus performed on a padded zero introduced in column 2 (which of course does not belong to node 1).



(a) Elimination tree.

1:	X	0	X
2:	X	X	0
3:	0	X	X
4:	0	0	0

(b) Structure of the solution.

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

(c) Intervals for the backward phase.

Figure 6: Example of elimination 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 to the backward phase (c) are shown for $B = 3$ in the case where the permutation is the natural order $\{1, 2, 3\}$.

6 Experiments

We carried our experiments on the Hyperion Altix ICE 8200 system at the Centre Interuniversitaire de Calcul de Toulouse (CICT). Each node of the machine has two quad-core 2.8 GHz Intel Xeon 5560 processors and 32 GB of main memory. Our experimental set of matrices is described in Table 1. They all correspond to large problems arising from industrial or academic applications.

Matrix name	Order N	Entries (millions)	Factors (GB)	Description, origin, and type
NICE20MC$^{(*)}$	715923	28.1	8.3	Seismic processing; BRGM lab; double real
AUDI$^{(*)}$	943695	39.3	9.9	Automotive crankshaft model; Parasol collection; double real
bone010$^{(*)}$	986703	36.3	8.6	3D trabecular bone; Mathematical Research Institute of Oberwolfach; double real
CONESHL$^{(*)}$	1262212	43.0	5.5	3D finite element, SAMCEF code; SAMTECH; double real
Hook_1498$^{(*)}$	1498023	31.2	12.3	3D model of a steel hook; Padova University; double real
CAS4R_LR15	2423135	19.6	4.5	3D electromagnetism; EADS Innovation Works; single complex

Table 1: Set of matrices used for the experiments. Those marked with $^{(*)}$ are publicly available at either **gridtlse.org** or in the University of Florida Sparse Matrix Collection.

6.1 Sequential case

Exploiting sparsity within blocks enables us to decrease the computational cost, and doing this is interesting not only in a parallel setting but also for serial execution. In this section, we report on experiments with six matrices in a sequential context. In all the cases, 10% of the diagonal entries of the inverse are computed using the sparse inverse functionality in MUMPS; the block size is $B = 1024$. Experiments were performed on one node (i.e. 8 cores) of the abovementioned 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 to be performed and gives some improvement in computation time. We see that the time improvement is a superlinear function of the number of operations. We believe that this is because many of the operations we gain correspond to nodes located at the bottom of the tree, where the GFlops/s rates are much smaller than in the top of the tree. Note that when exploiting

Matrix	Operations ($\times 10^{12}$)		Time (s)	
	w/o ES within blocks	w/ ES within blocks	w/o ES within blocks	w/ ES within blocks
AUDI	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
Hook_1498	111.2	104.9	2807	2141
CAS4R_LR15	15.0	$12.9^{(*)}$	1144	582

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.

sparsity within the blocks, we reach a speed of 49 GFlops/s, which exceeds 50% of the peak of `DGEMM` on this system. This is a very good speed 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 medium size problem (11-pt stencil discretization of a $200 \times 200 \times 20$ domain), using one node (eight cores) of the Hyperion system described above, with single-threaded BLAS and $B = 512$. We show the time for the solution phase, the operation count and the average number of processes active at the same time during the computation, for one, four and eight MPI processes (one per core). Firstly, 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 is 1.2 (1667/1366) on four processes): this is because the average number of processes that are active at the same time is close to one. If the interleaving procedure is activated but sparsity is not exploited within blocks, the operation count increases significantly (e.g. it is multiplied by 4 on eight processes); this prevents good speed-ups, even though the average number of active processes increases. When sparsity is exploited within blocks, the operation count and thus the gains are significantly better: almost 4 on eight processes.

Procs	Strategy		Time (seconds)	Operations ($\times 10^{12}$)	Active procs
1	-		1667	16.4	1
4	IL off	ES off	1366	16.4	1.20
	IL on	ES off	2028	42.4	3.92
		ES on	659	15.3	
8	IL off	ES off	1241	16.4	1.10
	IL on	ES off	1508	64.7	7.76
		ES on	418	15.3	

Table 3: Computation of a random 10% of the diagonal entries of the inverse of a matrix corresponding to an 11-pt stencil discretization of a $200 \times 200 \times 20$ domain, using one node of the Hyperion system and $B = 512$. The different strategies are compared: "IL" stands for the interleaving strategy and "ES" for exploiting sparsity within blocks of right-hand sides.

We report in Table 4 on experiments on six different matrices, using four nodes of the Hyperion system. 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 than the baseline algorithm: on matrix NICE20MC, the time is reduced by almost a factor 6 and the overall speed-up on 32 cores is quite interesting especially as we are considering the solution phase. Additionally, this is particularly impressive given that sparsity is exploited at many different levels.

Matrix	Processors		
	1	32	
		Baseline Time(s)	IL+ES Time(s)
AUDI	1143	380	75
NICE20MC	945	245	43
bone010	922	327	70
CONESHL	803	293	48
Hook_1498	1860	720	173
CAS4R_LR15	882	482	116

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

6.2.2 Influence of the blocksize

We show in Table 5 the influence of the block size B for the 11-pt stencil discretization of a $200 \times 200 \times 20$ domain. 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 512 and then 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 eight 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. If we compare the best sequential time with the best parallel time, the speed-up is almost 4 on 8 processes.

Procs	Strategy		Block size			
			64	128	512	1024
1	-		1518	1432	1667	2002
8	IL on	ES on	555	466	418	379

Table 5: Influence of the block size: a random 10% of the diagonal entries of the inverse of a matrix corresponding to an 11-pt stencil discretization of a $200 \times 200 \times 20$ domain are computed. The influence of the block size B on the time for the solution phase (indicated in seconds) is illustrated both for sequential and parallel executions, with interleaving ("IL") and exploiting sparsity within blocks ("ES"), on one node of the Hyperion system.

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 combined effect of these two strategies leads to a significantly improved sparse triangular solution phase for both sequential and parallel environments.

Finally, exploiting sparsity within blocks of right-hand sides has a larger scope than 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.

Acknowledgements

This work was granted access to the HPC resources of CALMIP under the allocation 2012-P0989.

References

[1] P. R. Amestoy, I. S. Duff, J. Koster and J.-Y. L'Excellent, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM Journal of Matrix Analysis and Applications, Vol 23, No 1, pp 15–41 (2001).

[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 Journal on Scientific Computing, Vol 34, No 4, pp A1975–A1999, 2012.

[3] P. R. Amestoy and A. Guermouche and J.-Y. L'Excellent and S. Pralet, *Hybrid scheduling for the parallel solution of linear systems*, Parallel Computing, Vol 32 (2), pp 136–156 (2006).

[4] Å. Björck, *Numerical methods for least squares problems*, Society for Industrial Mathematics, 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*, The Astrophysical Journal, 635 (2005), pp. 1103–1115.

[6] Y. Campbell, T. Davis, *Computing the sparse inverse subset: an inverse multifrontal approach*, Technical report, 1995.

[7] Y. Campbell, T. Davis, *A parallel implementation of the block-partitioned inverse multifrontal Zsparse algorithm*, Technical report, 1995.

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

[9] G. Geist and E. Ng, *Task scheduling for parallel sparse Cholesky factorization, International Journal of Parallel Programming*, 18 (1989), pp. 291–314.

[10] L. Lin, J. Lu, L. Ying, R. Car, W. E, *Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems*, Communications in Mathematical Sciences, 2009.

[11] L. Lin, C. Yang, J. Lu, L. Ying, W. E, *A fast parallel algorithm for selected inversion of structured sparse matrices with application to 2D electronic structure calculations*, SIAM Journal on Scientific Computing, Vol 33, No 3, 1329–1351, 2011.

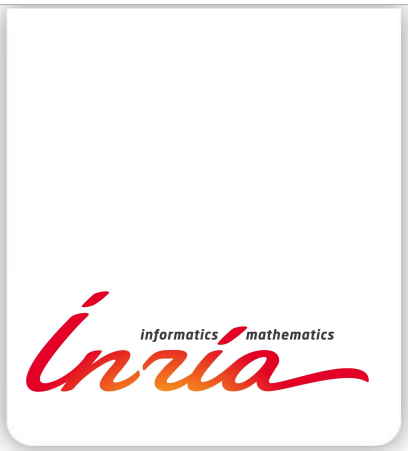
[12] M. Luisier, A. Schenk, W. Fichtner, and G. Klimeck, *Atomistic simulation of nanowires in the $sp^3d^5s^*$ tight-binding formalism: From boundary conditions to strain calculations*, Physical Review B, 74 (2006), p. 205323.

[13] J. Rommes and W. H. A. Schilders, *Efficient methods for large resistor networks*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol 29, No 1, pp 28–39 (2010).

[14] F.-H. Rouet, *Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides*, PhD 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*, PhD 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 8th PICA Conference, Minneapolis, Minnesota, 1973.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399