HAL
open science

Reordering strategy for blocking optimization in sparse linear solvers

Grégoire Pichon, Mathieu Faverge, Pierre Ramet, Jean Roman

To cite this version:

Grégoire Pichon, Mathieu Faverge, Pierre Ramet, Jean Roman. Reordering strategy for blocking optimization in sparse linear solvers. [Research Report] RR-8860, Inria Bordeaux Sud-Ouest; LaBRI - Laboratoire Bordelais de Recherche en Informatique; Bordeaux INP; Université de Bordeaux. 2016, pp.26. hal-01276746v1

HAL Id: hal-01276746

https://inria.hal.science/hal-01276746v1

Submitted on 19 Feb 2016 (v1), last revised 13 Oct 2016 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Reordering strategy for blocking optimization in sparse linear solvers

Grégoire Pichon, Mathieu Faverge, Pierre Ramet, Jean Roman

RESEARCH REPORT
N° 8860
February 2016
Project-Team HiePACS

ISSN 0249-6399 ISRN INRIA/RR--8860--FR+ENG

Reordering strategy for blocking optimization in sparse linear solvers

Grégoire Pichon[*†‡§], Mathieu Faverge[*†‡§ ¶], Pierre Ramet [‡†*§],

Jean Roman [†*‡§]

Project-Team HiePACS

Research Report n° 8860 — February 2016 — 26 pages

Abstract: Solving sparse linear systems is a problem that arises in many scientific applications, and sparse direct solvers are a time consuming and key kernel for those applications and for more advanced solvers such as hybrid direct-iterative solvers. For this reason, optimizing their performance on modern architectures is critical. The preprocessing steps of sparse direct solvers, ordering and block-symbolic factorization, are two major steps that lead to a reduced amount of computation and memory and to a better task granularity to reach a good level of performance when using BLAS kernels. With the advent of GPUs, the granularity of the block computations became more important than ever. In this paper, we present a reordering strategy that increases this block granularity. This strategy relies on the block-symbolic factorization to refine the ordering produced by tools such as METIS or SCOTCH, but it does not impact the number of operations required to solve the problem. We integrate this algorithm in the PASTIX solver and show an important reduction of the number of off-diagonal blocks on a large spectrum of matrices. This improvement leads to an increase in efficiency of up to 10% on CPUs and up to 40% on GPUs.

Key-words: Sparse linear solver, nested dissection, sparse matrix ordering, heterogeneous architectures

* Bordeaux INP, Talence, France
† Inria Bordeaux - Sud-Ouest, Talence, France
‡ University of Bordeaux, Talence, France
§ CNRS (Labri UMR 5800), Talence, France
¶ University of Tennesse, ICL, Knoxville, USA

RESEARCH CENTRE BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Stratégie de renumérotation pour optimiser la granularité des calculs dans la résolution des systèmes linéaires creux

Résumé : De nombreuses applications scientifiques recquièrent La résolution de large systèmes linéaires creux qui est généralement l'étape la plus comnsommatrice de ressources, que ce soit en temps de calculs ou mémoire. Il est donc primordial d'optimiser les bibliothèques de résolution de ces problèmes sur les architectures modernes. Nous présentons dans ce documents une technique de renumérotation des inconnues qui permet d'élargir la granularité des calculs afin de mieux exploiter les accélérateurs, comme les GPUs, dans ces bibliothèques. Cet algorithme s'appuie sur les renumérotations calculées par des outils comme Metis ou Scotch sans changer le nombre d'opérations de la factorisation numérique. Nous présentons les résultats de l'intégration de cette stratégie dans la bibliothèque PaStiX qui améliore les temps de factorisations de 10% sur les CPUs, et jusqu'à 40% lorsque des GPUs sont utilisés.

Mots-clés : algèbre linéaire creuse, dissection emboitée, renumérotation, architectures hétérogènes

1 Introduction

Many scientific applications, such as electromagnetism, astrophysics, and computational fluid dynamics, use numerical models that require solving linear systems of the form $Ax = b$. In those problems, the matrix A can either be considered as dense (almost no zero entries) or sparse (mostly zero entries). Due to multiple structural and numerical differences that appear in those problems, many different solutions exist to solve them. In this paper, we focus on problems leading to sparse systems with a symmetric pattern and more specifically on direct methods which factorize the matrix A in LL^t, LDL^t or LU, with L, D and U respectively unit lower triangular, diagonal, and upper triangular according to the problem numerical properties. Those sparse matrices appear mostly when discretizing Partial Differential Equations (PDEs) on 2D and 3D finite element or finite volume meshes. The main issue with such factorizations is the fill-in – zero entries becoming non-zero – that appears in the factorized form of A during the execution of the algorithm. If not correctly considered, the fill-in can transform the sparse matrix into a dense one which might not fit in memory. In this context, sparse direct solvers rely on two important preprocessing steps to reduce this fill-in and control where it appears.

The first one finds a suitable unknown ordering that aims at minimizing the fill-in to limit the memory overhead and the floating point operations (Flop) required to complete the factorization. The problem is then transformed into $(PAP^t)(Px) = Pb$ where P is an orthogonal permutation matrix. A wide array of literature exists on solutions to graph reordering problems; the most commonly used for sparse direct factorization being the nested dissection recursive algorithm introduced by George [1].

The second preprocessing step of sparse direct solvers is the block-symbolic factorization [2]. This step analytically computes the block-structure of the factorized matrix from the reordering step and from a supernode partition of the unknowns. It allows the solver to create the data structure that will hold the final matrix instead of allocating it at runtime. The goal of this step is also to block the data in order to efficiently apply matrix-matrix operations, also known as BLAS Level 3 [3], on those blocks instead of scalar operations. For this purpose, extra fill-in, and by extent extra computations, might be added in order to reduce the time to solution. However, the size of those blocks might not reach the sufficient size to extract all the performance from the BLAS kernels.

Modern architectures, whether based on CPUs, GPUs, or Intel Xeon Phi may be efficient with a performance close to the theoretical peak. This can be achieved only if the data size is large enough to take advantage of caches, vector units, and provides a larger ratio of computation per byte. Accelerators such as GPUs or Intel Xeon Phi require even larger blocking sizes than the ones for CPUs due to their particular architectural features.

In order to provide more suited block sizes to kernel operations, we propose in this paper an algorithm that reorders the unknowns of the problem to increase the average size of the off-diagonal blocks in block-symbolic factorization structures. The major feature of this solution is that, based on an existing nested dissection ordering for a given problem, our solution will keep stable the amount of fill-in generated during the factorization. So, the amount of memory and computation to store and compute the factorized matrix is invariant. The consequence of this increased average size is that the number of off-diagonal blocks is largely reduced, diminishing the memory overhead of the data structures used by the solver and the number of tasks required to compute the solution in task-based implementations [4, 5], increasing the performance of BLAS kernels.

Section 2 gives a brief background on the block-symbolic factorization for sparse direct solvers and introduces the problem when classical reordering techniques are used. Section 3 states the problem and describes our reordering strategy and his theoretical cost. The quality of the

symbolic structure and its impact on the performance of a sparse direct solver, here PaStiX [6], is studied in section 4. Finally, we conclude and present some future opportunities for this work.

2 Background

This section provides some background on sparse direct solvers and the associated preprocessing steps. A detailed example showing the impact of the ordering on the block-structure for the factorized matrix is also presented.

2.1 Sparse direct solvers

The common approach to sparse direct solvers is composed of four main steps: 1) ordering and computation of a supernodal partition for the unknowns; 2) block-symbolic factorization; 3) numerical block-factorization; and 4) triangular system solve. The first step exploits the property of the Characterization theorem (see theorem 2.1) to minimize the fill-in, zeros becoming non-zeros during factorization, and the second one predicts the block-structure that will facilitate efficient numerical factorization and solve. The last two steps perform the actual computation and can benefit multiple times from the preprocessing steps as long as the original matrix structure does not change.

Theorem 2.1. *(Characterization Theorem) Given an $n \times n$ sparse matrix A, and its adjacency graph $G = (V, E)$, any entry $a_{i,j} = 0$ from A will become a non-null entry in the factorized matrix if and only if there is a path in G from vertex i to vertex j that only goes through vertices with a lower index than i and j.*

Among the ordering techniques known to efficiently reduce the matrix fill-in are the Approximate Minimum Degree (AMD) algorithm [7] and the Minimum Local Fill (MF) algorithm [8,9]. However, these orderings fail to expose a lot of parallelism in the block computation during the factorization. In order to both reduce fill-in and exhibit parallelism, an ordering algorithm based on nested dissection [1] has been introduced and is now the most widely used in sparse direct solvers. This class of algorithms works on the symmetric undirected graph associated with the matrix and recursively partitions the graph to expose independent subproblems that can be solved in parallel while reducing the fill-in of the matrix.

Top-left part of figure 1 shows the adjacency graph of a 2D symmetric 5×5 grid with a possible 2-level partitioning of the graph. The goal of the nested dissection method is to recursively partition the graph $G = (V, E)$ into $A \cup B \cup C$ such that no edge directly connects a vertex from A to a vertex of B, and conversely such that C is as small as possible. C is called the separator and corresponds to a supernode. This separation of A and B combined with theorem 2.1 guarantees that if all vertices of C are numbered with larger numbers than those of A and B, no fill-in appears between a vertex from A and a vertex from B. This partitioning and ordering process is then recursively applied on A and B, until a small enough size is reached for the subgraphs. Then, local ordering heuristics like AMD are used on these remaining subgraphs. A global supernode partition of the unknowns is obtained by merging the set of supernodes from the nested dissection process (all the separators) and the set of supernodes achieved from the reordered non-separated subgraphs (by using the algorithm introduced in [10, 11]). This partitioning and ordering operation is usually performed through an external tool such as Scotch [12] or Metis [13].

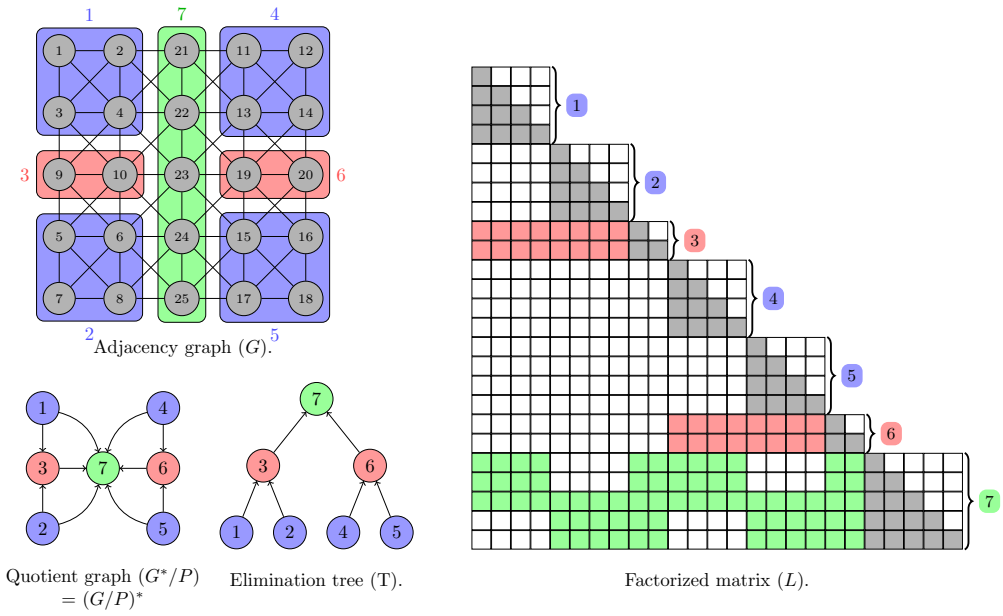


Figure 1: Nested dissection and block-data structure for L.

Given this supernodal partition, one can compute the block-symbolic data structure of the factorized matrix, as presented on the right part of figure 1. The goal is to predict the block-data structure of the final L matrix for the numerical factorization and to gather information in blocks that will enable the use of efficient kernels as BLAS Level 3 operations [3]. This block-data structure is composed of N column blocks, one for each supernode of the partition, with a dense diagonal block (in gray in the figure) and with several dense off-diagonal blocks (in green or in red in the figure) corresponding to interactions between supernodes; some additional fill-in is accepted to form dense blocks in order to be more CPU-efficient. The block-symbolic factorization computes this block-data structure with $\Theta(N)$ space and time complexities [2]. From this structure, one can deduce the quotient graph which describes all the interactions between supernodes during the factorization (for example, supernode 1 will contribute to supernodes 3 and 7), and the elimination tree which describes the amount of parallelism in the computations as a supernode will contribute only to supernodes belonging to its ascendance in the tree. Finally, before distributing the column blocks on the processors, the biggest column blocks corresponding to the top most supernodes in the tree are split in order to exploit the parallelism inside the dense computations [6].

The first two steps of a direct solver are preprocessing stages, independent from numerical values. Note that those steps can be computed once to solve the same problem several times with different numerical values. Steps 3 and 4 are numerical. Figure 2 presents how the elimination of a column block is divided into three stages:

1. Factorization of the dense diagonal block;
2. Application of an in place Solve on the off-diagonal blocks;
3. Update of the underlying matrix.

Usually, the solve stage (stage 2) is done through one or multiple calls to BLAS kernels

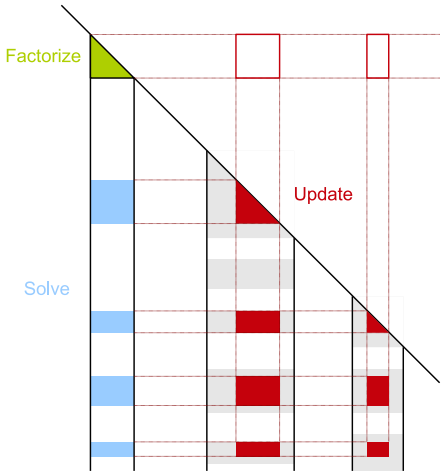


Figure 2: Steps to factorize the matrix - symmetric definite positive case.

according to the data distribution used by the solver. In the PaStiX solver, a 1D distribution is used and all blocks are stored contiguously in dense storage in order to perform the solve stage in only one BLAS call. The update stage (stage 3) can be done in multiple ways. The first option is to do it similarly to dense factorization with one matrix-matrix multiply per couple of off-diagonal blocks (red updates in figure 2). However, the granularity of the tasks in sparse solvers is often so small before reaching the top levels of the elimination tree that it is inefficient. The most adopted solution for supernodal methods is to compute a matrix-matrix multiply for each column block that requires updates, meaning one per blue off-diagonal block (only the first two are represented in figure 2). The temporary result is then scattered and added to the target column block. The last solution, similar to what is done in a multifrontal solver, is to perform only one matrix-matrix multiplication and do a 2D scatter of the updates. In both last solutions, if the updates are too discontinuous and spread all over the updated submatrix, this can lead to memory bound updates while the operation is originally compute bound. It is then interesting to consider an ordering solution, compatible with the nested dissection method, that will limit the number of off-diagonal blocks to have more compacted updates. It will also reduce the memory bound aspect of the update operation, which is the most time consuming for the factorization and solve steps.

2.2 Intra-node reordering

Let us now illustrate the problem of current ordering solutions and how to overcome this problem. For this purpose, we consider a regular 3D cube of n^3 vertices presented in figure 3. We apply the nested dissection process to this cube. Naturally, the first separator, in gray, is a plane of n^2 vertices cutting the cube into two halves of balanced parts. Then, by recursively applying the nested dissection process, we partition the two-halves' subparts with the two red separators, and again dissect the resulting partitions by the four third-level green separators giving us eight final partitions. We know from this process that each separator will be ordered with higher indices than those in lower levels.

Inside each separator, vertices have to be ordered as well, and it is common to use techniques such as the Reverse Cuthill-McKee [14] (RCM) algorithm in order to have an internal separator ordering "as continuous as possible" to limit the number of off-diagonal blocks in the associated

column block. This strategy works with only the local graph induced by the separator. It starts from a peripheral vertex and orders, consecutively, vertices at distance 1, then at distance 2, and so on, giving indices in reverse order. It is close to a Breadth-First Search (BFS) algorithm. However, such an algorithm uses only interactions within a supernode, without taking into account contributing supernodes. On the quotient graph of figure 1, it means that this will reorder unknowns inside a node of this graph without considering interactions with other nodes of this graph. However, these interactions are the ones related to off-diagonal blocks in the factorized matrix. Therefore, it is important to note that the ordering inside a supernode can be rearranged to take into account interactions with vertices outside its local graph without changing the final fill-in of the L block-structure used by the solver. Then, we can expect that a complete knowledge of the local graph and of its outer interactions will lead to a better quality in terms of the number of off-diagonal blocks.

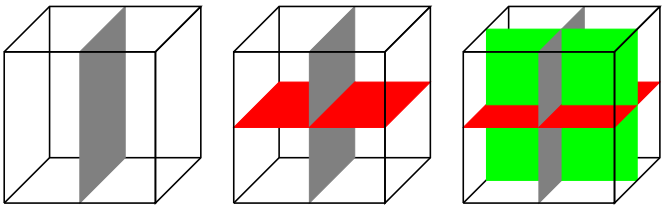
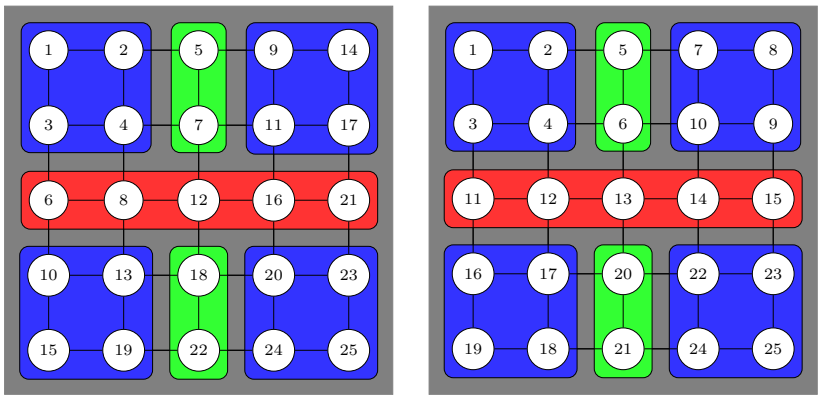


Figure 3: Three-levels of nested dissection on a regular cube.

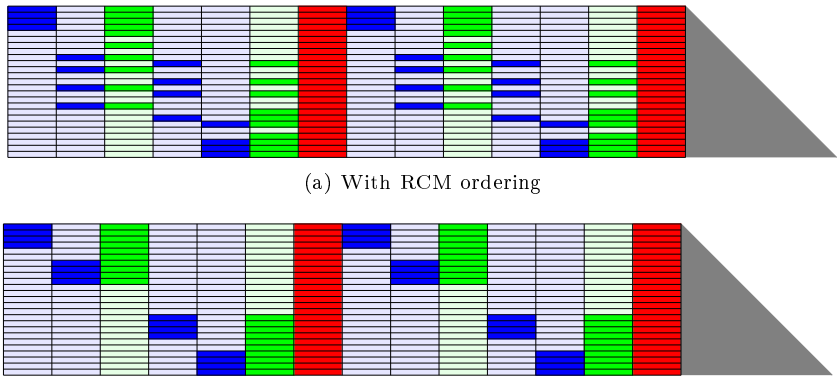


(a) With RCM ordering (b) With optimal ordering

Figure 4: Projection of contributing supernodes and ordering on the first separator (gray in figure 3).

Figure 4 presents the vertices of the gray separator from the 3D cube case with $n = 5$. The projection of contributing supernodes on this separator is shown. The blue parts are the vertices connected only to the leaves of the elimination tree. Thanks to the nested dissection process, the nodes of the gray separator have the largest numbers and their connections to other supernodes represent the off-diagonal contributions. Based on this, we propose an optimal ordering, in figure 4(b), computed by hand, as opposed to an RCM algorithm, in figure 4(a). One can note that RCM will not order, consecutively, vertices that will receive contributions from the same supernodes, leading to a substantially larger number of off-diagonal blocks than

the optimal solution. For instance, the four blue vertices in the top right of RCM ordering will create four different off-diagonal blocks. The general idea is that some projections will be cut by RCM following the neighborhood, while those vertices could have been ordered together to reduce the number of off-diagonal blocks. On the right, the optimal ordering tries to consider this rule by ordering vertices with similar connections in a contiguous manner. This leads to a smaller number of off-diagonal blocks as shown in the block-data structure computed by the block-symbolic factorization for these two orderings in figures 5(a) and 5(b).



(a) With RCM ordering

(b) With optimal ordering

Figure 5: Off-diagonal blocks contributing to the first separator in figure 3.

We have demonstrated with this simple example that RCM does not fulfill the correct objective in a more global view of the problem. This is especially true in the context of 3D graphs, where the separator is a 2D structure, receiving contributions from 3D structures on both sides. With 2D graphs, the separator is a 1D structure and in such a case, RCM will provide generally a good solution by following the neighborhood in the BFS algorithm. Moreover, it often happens that the separators found by generic tools such as Scotch or Metis are disconnected making this previous statement incorrect.

Note that if it is quite easy to manually compute the optimal ordering on our example, it is harder in practice. Indeed, given an initial partition $V = A \cup B \cup C$, nothing guarantees that subparts A and B will be partitioned in a similar fashion, and that the resulting projection will match. For instance, figure 6 presents the projection of level-1 (in red) and level-2 (in green) supernodes on the first separator of a $40 \times 40 \times 40$ laplacian partitioned with Scotch. One can note that there are crossed contributions, meaning that subparts A and B are partitioned differently.

In the next section, we propose a new reordering strategy that permutes the rows to compact the off-diagonal information. Note that such a reordering strategy will not impact the global fill-in as long as the diagonal blocks are considered as dense blocks. The first solution that appears on figure 6 would be to cluster vertices by common connections to nodes of the quotient graph. However, in most cases, that would result in clusters of $O(1)$ size that would still need to be ordered correctly, taking into account their level in the elimination tree of the connected supernodes. The solution we propose to remedy this problem relies on the computed block-data structure. Our objective is to express an algorithm providing the optimal solution before proposing a heuristic with a reasonable complexity.

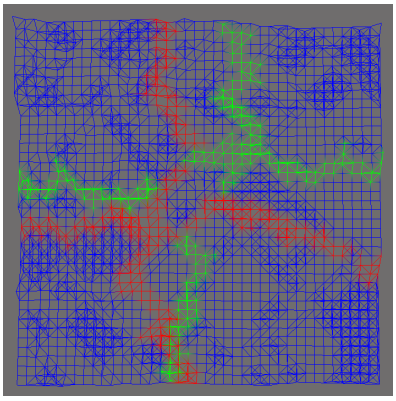


Figure 6: Projection of contributing supernodes on the first separator of a 3D laplacian of size $40 \times 40 \times 40$, using SCOTCH.

2.3 Related Works

Studying the structure of off-diagonal blocks was used in different contexts. In [15], the purpose was to reduce the overhead associated with each single off-diagonal block. The authors proposed a reordering strategy that refines the ordering provided by the minimum degree algorithm. Their experiments have been applied to 2D graphs and successfully reduce the number of off-diagonal blocks. However, the authors did not provide a theoretical study of their reordering algorithm and their solution did not apply in the context of 3D graphs. In [16], the authors introduce reordering in the context of a multifrontal solver. A front is ordered according to its two sons (in the nested dissection process), starting with the largest son, in order limit the scatter operation when updating the front with the son's contributions. A similar approach was studied in [17] for the MUMPS solver, to reorder blocks in order to reduce the communication volume induced by the fronts in distributed contexts.

3 Improving the blocking size

As presented in section 2.2, the RCM algorithm – widely used to order supernodes – generates many extra diagonal blocks by not considering supernode interactions, which leads to an increased number of less efficient block operations. In this section, we present an algorithm that intends to reorder supernodes thanks to a global view of the nested dissection partition. We expect that considering contributing supernodes will lead to a better quality — a smaller number of larger blocks. Our proposition is to consider the set of contributions for each line of a supernode, before using a distance metric to minimize the creation of off-diagonal blocks when permuting lines.

3.1 Problem modeling

The main idea is to rely on the block-symbolic factorization of L instead of the original graph of A. Indeed, it allows us to take into account fill-in elements that were computed thanks to the block-symbolic factorization process instead of re-computing those elements with the matrix graph. Let us consider the ℓth diagonal block C_ℓ of the factorized matrix that corresponds to a supernode, and the set of supernodes C_k with $k < \ell$ corresponding to the supernodes in lower levels of the elimination tree than C_ℓ. Note that we refer to N as the total number of diagonal

blocks appearing in the structure of the factorized matrix, as opposed to n for the total number of unknowns.

We define for each supernode C_ℓ:

$$row_{ik}^{\ell} = \begin{cases} 1 \text{ if vertex } i \text{ from } C_\ell \text{ is connected to } C_k \\ 0 \text{ otherwise} \end{cases}, k \in [\![1, \ell-1]\!], i \in [\![1, |C_\ell|]\!]. \tag{1}$$

row_{ik}^{ℓ} is then equal to 1 when the vertex i, or row i, of the supernode C_ℓ is connected to any vertex of the supernode k belonging to a lower level in the elimination tree. It is equal to 0, if not, meaning that no non-zero element connects the two in the initial matrix, or no fill-in will create that connection. Let's now define for each vertex the set $B_i^\ell = (row_{ik}^{\ell})_{k \in [\![1,\ell-1]\!]}$. We can then define w_i^ℓ, the weight of a row i, as in equation (2), that represents the number of supernodes contributing to that row i, and the distance between two rows i and j, $d_{i,j}^\ell$, as in equation (3). It is known as the Hamming distance [18] between two binary vectors, and allows for measuring the number of off-diagonal blocks induced by the succession of two rows i and j. Indeed, $d_{i,j}^\ell$ represents the number of off-diagonal that belongs to only one of the two rows, which can be seen as the number of blocks that end at row i or start at row j.

$$w_i^\ell = \sum_{k=1}^{\ell-1} row_{ik}^{\ell}, \tag{2}$$

$$d_{i,j}^\ell = d(B_i^\ell, B_j^\ell) = \sum_{k=1}^{\ell-1} row_{ik}^{\ell} \oplus row_{jk}^{\ell}, \tag{3}$$

where $\oplus$ is the exclusive `or` operation.

Thus, the total number of off-diagonal blocks, odb^ℓ, contributing to the diagonal block C_ℓ can be defined as:

$$odb^\ell = \frac{1}{2}(w_1^\ell + \sum_{i=1}^{|C_\ell|-1} d_{i,i+1}^\ell + w_{|C_\ell|}^\ell), \tag{4}$$

where the Hamming weights of the first and last row of the supernode C_ℓ correspond respectively to the number of blocks in the first row and in the last one, and the distances between two consecutive rows gives the evolution in the number of blocks when traveling through them.

Then, to reduce the total number of off-diagonal blocks in the final structure, the goal is to minimize this metric odb^ℓ for each supernode, by computing a minimal path visiting each node, with a constraint on the first and the last node. This problem is known as the Shortest Hamiltonian Path Problem, and is an NP-hard problem.

3.2 Proposed heuristic

We first propose to introduce an extra fictive vertex, S_0, for which B_0 is the null set. Thus, we have:

$$\forall i \in [\![1, |C_\ell|]\!],\ d_{0,i} = d_{i,0} = w_i, \tag{5}$$

The problem can now be transformed in a Traveller Salesman Problem [19] (TSP):

$$\sum_{i=0}^{|C_\ell|} d_{i,(i+1)}^\ell, \tag{6}$$

which is also an NP-Hard problem, but for which multiple heuristics have been proposed in the literature [20], as opposed to Shortest Hamiltonian Path Problem. Furthermore, our problem presents properties that make it compatible for better heuristics and theoretical models that guarantee the maximum distance to the optimal solution. Firstly, our problem is symmetric as:

$$d_{ij}^{\ell} = d_{ji}^{\ell}, \forall (i,j) \in [\![1, |C_\ell|]\!]^2, \tag{7}$$

and secondly, respects the triangular inequality:

$$d_{ij}^{\ell} \leq d_{ik}^{\ell} + d_{kj}^{\ell}, \forall (i,j,k) \in [\![1, |C_\ell|]\!]^3. \tag{8}$$

This sets our problem as an Euclidean TSP, and so heuristics for these specific cases can be used. Different TSP heuristics that can be used to solve this problem, with their respective cost and quality with respect to the optimal, are presented in table 1.

To keep a global complexity below that of the numerical factorization, we explain in the following section 3.3 that the complexity of the TSP algorithm has to remain equal or lower to $\Theta(p^2)$, where p is the number of vertices in the cycle. Thus, it prevents advanced algorithms such as the Christofides algorithm [21] from being used. Furthermore, as p might reach several hundred or more, the use of Nearest neighbor or Clarke and Wright heuristics might provide low quality results. From the remaining options, we decided to use the nearest insertion method which is a quadratic algorithm and guaranties a maximal distance to the optimal of 2 [22]. A quality comparison of our algorithm over 3D Laplacian matrices and real matrices against the Concorde [23] TSP solver that returns optimal solutions have shown that our nearest insertion algorithm provides results in less than 10% from the optimal.

Algorithm	Complexity for p nodes	Quality (wrt optimal)
Nearest neighbor	$\Theta(p^2)$	$\frac{1}{2}(1 + log(p))$
Nearest insertion	$\Theta(p^2)$	2
Clarke and Wright	$\Theta(p^2\ log(p))$	$\Theta(log(p))$
Cheapest insertion	$\Theta(p^2\ log(p))$	2
Minimum spanning tree	$\Theta(p^2)$	2
Christofides	$\Theta(p^3)$	1.5

Table 1: Complexity and quality of different TSP algorithms.

Our final algorithm is then decomposed in three stages presented in algorithm 1 that are applied to each separator of the nested dissection. Note that it is not applied on the leaves of the elimination tree since they will not receive contributions from other supernodes. The first step is to compute the B_i^ℓ vectors for each row i of the current separator. Then, it computes the distances matrix of the separator: $D_\ell = (d_{i,j})_{(i,j)\in[\![0,|C_\ell|]\!]^2}$. Finally, the TSP algorithm is executed using this matrix to produce the local ordering of the supernode that minimizes the equation (6).

The first stage of this algorithm builds the vector B_i^ℓ for each row i. In fact, to minimize the storage, only contributing supernodes ($row_{ik}^\ell = 1$) are stored for B_i^ℓ. In order to do so, we rely on the structure of the block-symbolic factorization that provides a compressed storage of the information similar to compress sparse row (CSR) format. Given a supernode, one can easily access the off-diagonal blocks contributing to this supernode, and due to the sparse property, the number of these blocks is much smaller than $(\ell - 1)$. The accumulated operations for all the supernodes in the matrix is in $\Theta(n)$. Note that we store the contributing supernode numbers in

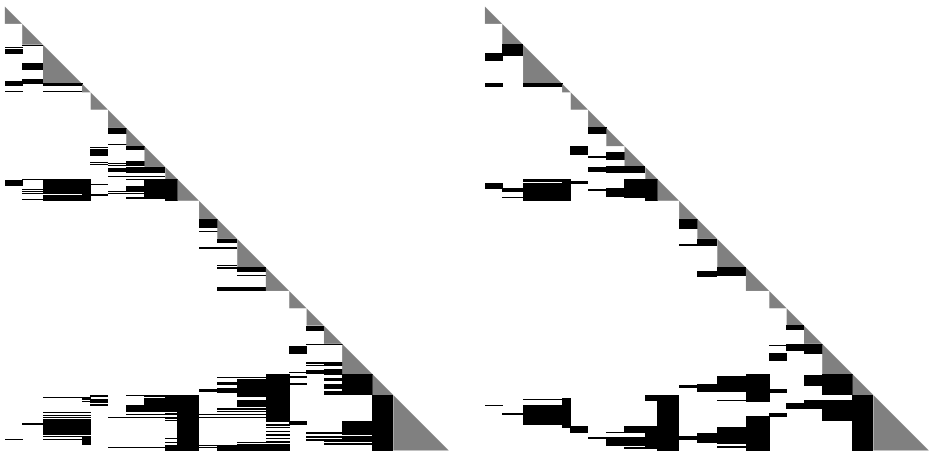
Algorithm 1 Reordering algorithm

```
for each supernode C_ℓ in the elimination tree do
    for each row i in the supernode C_ℓ do
        for each contributing node k ∈ [[1, ℓ − 1]] do
            Set row^ℓ_ik to 1                                   ▷ Build the structure B^ℓ_i
        end for
    end for
    for each row i in the supernode C_ℓ do
        for each row j in the supernode C_ℓ do
            Compute the distance between rows i and j           ▷ Compute the distances
        end for
    end for
    Cycle^ℓ = {S_0, 1}
    for i ∈ [[2, |C_ℓ|]] do
        Insert row i in Cycle^ℓ such that (6) is minimized      ▷ Order rows
    end for
    Split Cycle^ℓ at S_0
end for
```

an ordered fashion for faster computation of the distances. Furthermore, the memory overhead of this operation is limited by the fact that each supernode is treated independently.

The second stage computes the distance matrix. When computing the distance d^{ℓ}_{ij} between rows i and j from C_ℓ, we take advantage of the sorted sets B^{ℓ}_i and B^{ℓ}_j to realize this computation in $\Theta(|B^{\ell}_i| + |B^{\ell}_j|)$ operations.

The third stage executes the nearest insertion heuristics to solve the TSP problems on the vertices of the supernode based on the previously computed distance matrix. As stated previously, this step is computed in $\Theta(|C_\ell|^2)$ operations. It is known that the solution given is not optimal but will be at a distance 2 of the optimal in the worst case.



(a) Without reordering strategy (b) With reordering strategy

Figure 7: Block-symbolic factorization of $8 \times 8 \times 8$ Laplacian initially reordered with Scotch.

Figure 7 presents the block-symbolic factorization of a 3D Laplacian of size $8 \times 8 \times 8$ reordered

with the SCOTCH nested dissection algorithm. In figure 7(a), our reordering algorithm has not been applied, and supernodes ordering results only from the local RCM applied by SCOTCH. One can notice that some rows can be easily aggregated to reduce the number of off-diagonal blocks. In figure 7(b), our algorithm has to reorder unknowns within each supernode. The final structure exhibits more compact blocks that are larger. Note that the fill-in of the matrix has not changed due to the dense storage of the diagonal blocks. Our algorithm does not impact the fill-in outside those diagonal blocks.

3.3 Complexity study

In real-life applications, a large part of the graphs are issued from finite element mesh representations of physical 2D or 3D problems. From a theoretical point of view, the majority of those graphs have a bounded degree and are specific cases of bounded-density graphs [24]. In this section, we provide a complexity study of our reordering algorithm in the context of a nested dissection partitioning strategy for this class of graphs.

Good separators can be built for bounded-density graphs [24] or more generally for overlap graphs [25]. In d-dimension, such n-node graphs have separators whose size grows as $\Theta(n^{(d-1)/d})$. In this study, we consider the general framework of separator theorems introduced by Lipton and Tarjan [26] for which we will have $\sigma = \frac{d-1}{d}$.

Definition 3.1. *A class φ of graphs satisfies an n^σ-separator theorem, $\frac{1}{2} \leq \sigma < 1$, if there are constants $\frac{1}{2} \leq \alpha < 1, \beta > 0$ for which any n-vertex graph in φ has the following property: the vertices of G can be partitioned into three sets A, B, and C such that:*

- *no vertex in A is adjacent to any vertex in B,*
- *$|A| \leq \alpha n$, $|B| \leq \alpha n$, and*
- *$C \leq \beta n^\sigma$ where C is the separator of G.*

Theorem 3.2. *(From [2]) The number of off-diagonal rows in the block data structure for the factorized matrix L is at most $\Theta(n)$.*

This result comes from [2]. In this paper, the authors demonstrated that the number of off-diagonal blocks is at most $\Theta(n)$ and this was achieved by proving that this upper bound is in fact true for the total number of rows inside the off-diagonal blocks, leading to theorem 3.2. Using this theorem, we demonstrate theorem 3.3.

Theorem 3.3. *(Reordering Complexity) For a graph of bounded degree satisfying a n^σ-separation theorem, the reordering algorithm complexity is bounded by $\Theta(n^{\sigma+1})$.*

Proof

The main cost of the reordering algorithm is issued from the distance matrix computation. As presented in section 3.2, we compute a distance matric for each supernode. This matrix is of size $|C_\ell|$, and each element of the matrix, D_ℓ, is the distance between two rows of the supernode. The overall complexity is then given by:

$$\mathcal{C} = \sum_{\ell=1}^{N} \sum_{i=1}^{|C_\ell|} (row_{ik}^{\ell})_{k \in [\![1,\ell-1]\!]} \times (|C_\ell| - 1). \tag{9}$$

More precisely, for a supernode C_ℓ, the complexity is given by the number of off-diagonal rows that contribute to it multiplied by the number of comparisons: $(|C_\ell| - 1)$. For instance, given figure 5(a), one can note that the complexity will be proportional to the colored surface (blue, green, and red blocks), where $row_{ik}^\ell = 1$, as well as in the number of rows. Using the compressed sparse information (colored blocks) only – instead of the dense matrix – is important for reaching a reasonable theoretical complexity, as long as this number of off-diagonal blocks is bounded in the context of finite element graphs.

Given theorem 3.2, we know that the number of off-diagonal contributing rows in the complete matrix L is in $\Theta(n)$. In addition, the largest separator is asymptotically smaller than the maximum size of the first separator that is $\Theta(n^\sigma)$. The complexity is then bounded by:

$$\mathcal{C} \leq \underbrace{\max_{1\leq\ell\leq N}(|C_\ell| - 1)}_{\Theta(n^\sigma)} \times \underbrace{\sum_{\ell=1}^{N}(\sum_{i=1}^{|C_\ell|}(row_{ik}^\ell)_{k\in[\![1,\ell-1]\!]})}_{\Theta(n)} = \Theta(n^{\sigma+1}).$$

For graphs of bounded degree, this result leads to:

- for the graph family admitting an $n^{\frac{1}{2}}$-separation theorem (2D meshes), the reordering cost is bounded by $\Theta(n\sqrt{n})$, and is – at worst – as costly as the numerical factorization;
- for the graph family admitting an $n^{\frac{2}{3}}$-separation theorem (3D meshes), the reordering cost is bounded by $\Theta(n^{\frac{5}{3}})$, and is cheaper than the numerical factorization, which grows as $\Theta(n^2)$.

Analysis

Note that this complexity is – as said before – larger than the complexity of the TSP nearest insertion heuristic. For a subgraph of size p respecting the p^σ-separation theorem, this heuristic complexity is in $\Theta(p^{2\sigma})$. Using [2], we can compute the overall complexity as a recursive function depending on the complexity on one supernode. It leads to an overall complexity in $\Theta(n\ log(n))$ for 2D graphs and $\Theta(n^{\frac{4}{3}})$ for 3D graphs, and is then less expensive than the complexity of computing the distance matrix.

The reordering is as costly as the numerical factorization for 2D meshes, but RCM is usually giving a good ordering on 2D graphs, as long as the separators are contiguous lines. For the 3D cases, the reordering strategy is cheaper than the numerical factorization. Thus, this reordering strategy is interesting for any graph with $\frac{1}{2} < \sigma < 1$, including graphs with a structure between 2D and 3D meshes. This algorithm can easily be parallelized since each supernode is an independent subproblem, and the distance matrix computation can also be computed in parallel. Thus, the sequential cost of this reordering step can be lowered and should be negligible compared to the numerical factorization.

Figure 8 presents the complexity study on 3D Laplacian matrices. We computed the practical complexity of our reordering algorithm with respect to the upper bound we demonstrated. The red curve presents the sequential time taken by our reordering algorithm. It is compared to the theoretical complexity demonstrated previously, but scaled to match on the middle point (size 150^3) to ease the read and check, so we can confirm that the trends of both curves are identical to a constant factor. Finally, in green we also plotted the practical complexity: total number of comparisons performed during our reordering algorithm, to see if the theoretical complexity was of the same order. This curve is also scaled to match on the middle point. One can note that

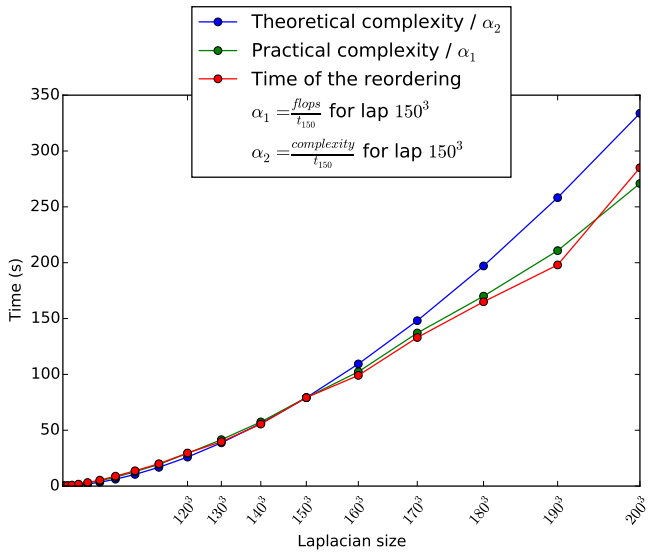


Figure 8: Comparison of the time of reordering against theoretical and practical complexities on 3D Laplacians.

the three curves are quite close, which confirms that we found a good upper bound complexity for a large set of sizes.

Note that this complexity seems to be significant with respect to the factorization complexity. Nevertheless, the nature of operations (simple comparisons between integers) is much cheaper than the numerical factorization operations. In addition, if we use the partitioner to obtain large enough supernodes, it will reduce by a notable factor the complexity of our algorithm, as long as we operate on a column block and not on each element contributing to each row. This parameter can be set in ordering tools as Scotch and Metis, and has an impact on the global fill-in of the matrix. As presented before, the reordering stage takes part of preprocessing steps, and can be used for many numerical steps and it enhances both factorization and solve steps.

3.4 Strategies to reduce computational cost

As we have seen, the total complexity of the reordering step can still reach the complexity of the numerical factorization. We now introduce heuristics to reduce the computational cost of our reordering algorithm.

Multi-Level: partial computation based on the elimination tree

As presented in section 2, the obtained partition allows us to decompose contributing supernodes according to the elimination tree. With the characterization theorem, we know that when we consider one row, if this row receives a contribution from a supernode in the lowest levels, then it will receive contributions from all its descendants to this node. This helps us divide our distance computation into a two dimension distance $d_{i,j} = d_{i,j}^{high} + d_{i,j}^{low}$ to reduce its cost. Given a $split_{level}$ parameter, we first compute the *high-levels* distance, $d_{i,j}^{high}$, by considering only the contributions from the supernodes in the $split_{level}$ levels directly below the studied supernode. This distance gives us a minimum of the distance between two rows. Indeed, if considering all supernodes,

the distance will be necessarily equal or larger to the *high-levels* distance by construction of the elimination tree. Then, we only compute the *low-levels* distance, $d_{i,j}^{low}$, only if the first one is equal to 0.

In practice, we observed that for a graph of bounded degree, not especially regular, a ratio of 3 to 5 between the number of lower and upper supernodes largely reduces the number of complete distances computed while conserving a good quality in the results. The $split_{level}$ parameter is then adjusted to match this ratio according to the part of the elimination tree considered. It is important to notice that it is impossible to consider the distances level by level, since the goal here is to group together the rows which are connected to the same set of leaves in the elimination tree. It means that they will receive contributions from nodes on identical paths in this tree. The partial distances consider only the beginning of those paths and not their potential *reconnection* further down the tree. That is why it is important to take multiple levels at once to keep a good quality.

Stopping criteria: partial computation based on distances

The second idea we used to reduce the cost of our reordering techniques is to stop the computation of a distance if it overcomes a threshold parameter. This solution helps to quickly disregard the rows that are "far-away" from each other. The resulting practical complexity is divided by an interesting factor, since a small value such as 10 can already give good quality improvement. Unfortunately, if this heuristic is used alone, this improvement is not always guaranteed and in some cases it can lead to a quality worsening. In association with the previous multi-level heuristic, the results are always improved, as we will see in the following section.

4 Experiment study

In this section, we present experiments with our reordering strategy, both in terms of quality (number of off-diagonal blocks) and impact on the performance of numerical factorization.

4.1 Context

We used a set of large matrices arising from real-life applications originating from the University of Florida's [27] Sparse Matrix Collection. For that experiment, we took all matrices from this collection with a size between 500.000 and 10.000.000. From this large set, we extracted matrices that are applicants for solving linear systems. Thus, we remove matrices originating from the Web and from DNA problems. This final set is composed of 104 matrices, sorted by families. We also conduct some experiments with a matrix of 10^7 unknowns, taken from a CEA simulation, an industrial partner in the context of the PaStiX project. All the matrices in this set are listed in the table **??** in the same order as they appear in the experiments.

We utilized the `Curie` heterogeneous computing platform from *TGCC*[1] to conduct our performance measurements. `Curie`'s nodes are composed of two Intel `Westmere` quadcore CPUs running at 2.66 GHz with 24 GB of memory, and enhanced by two Nvidia GPUs, `M2090 T20A`. We used Intel MKL `14.0.3.174` for the BLAS kernels on the CPUs, and we used the Nvidia Cuda `5.5.22` development kit to compile the GPU kernels.

For the scalability experiments in a multi-threaded context, we used the `miriel` cluster from the *Plafrim*[2] supercomputer. Each node is equipped with two Intel `Xeon E5-2680 v3` 12-cores running at 2.50 GHz and 128 GB of memory. The same version of the Intel MKL is used.

[1] `http://www-hpc.cea.fr/fr/complexe/tgcc-curie.htm`
[2] `https://plafrim.bordeaux.inria.fr`

The PaStiX version used for our experiments is the one implemented on top of the StarPU [28] runtime system presented in [4].

For the initial ordering step, we used Scotch 6.0 with the configurable strategy string from PaStiX to set the minimal size of the leaves, *cmin*, to 20 as in [29]. We also set the $frat$ parameter to 0.08, meaning that columns aggregation is allowed by Scotch as long as the fill-in introduced does not exceed 8% of the original matrix. It is important to use such parameters to increase the width of the column blocks and reach a good level of performance using accelerators. Even if it increases the fill-in, the final performance gain is usually more important and makes the memory overhead induced by the extra fill-in acceptable.

4.2 Reordering quality and time

First, we study the quality of the reordering algorithm we proposed, and the time it takes to compute compared to the original ordering computed by Scotch that is known to be in $\Theta(n \times log(n))$. For the quality criteria, the metric we use is the number of off-diagonal blocks in the matrix. We always use Scotch to provide the initial partition and ordering of the matrix, thus the number of off-diagonal blocks only reflects the impact of the reordering strategy. Another related metric we could use is the number of off-diagonal blocks per column block. In ideal cases, it would be, respectively, 4 and 6 for 2D and 3D meshes. However, since the partition computed by scotch is not based on the geometry, this optimal is never reached and varies a lot from one matrix to another, so we stayed with the global number of off-diagonal and its evolution compared to the original solution given by Scotch.

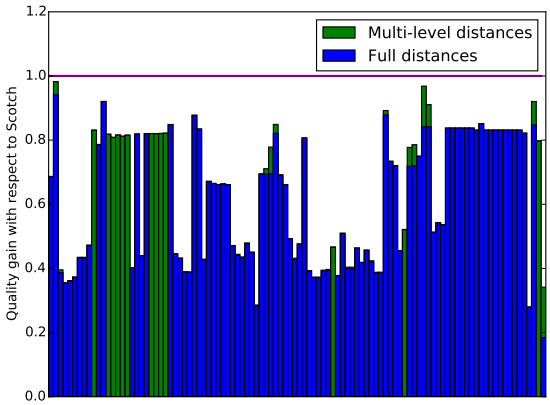


Figure 9: Impact of the distance heuristic on the ratio of off-diagonal blocks in our reordering algorithm over those produced by the initial Scotch ordering on the Florida set of matrices.

Quality

Figure 9 presents the quality of our reordering strategy in terms of the number of off-diagonal blocks with respect to the Scotch ordering. We recall that Scotch uses RCM to order unknowns within each supernode. The color on top gives the heuristic to compute the distances with the best result: green for the multi-level heuristic, and blue for the full distance computation. We can see that our algorithm reduces the number of off-diagonal blocks on all test cases. On the 3D problems, our reordering strategy improves the metric by 50 to 60%, while on the 2D problems, the improvement is of 20 to 30%. Furthermore, we can observe that the multi-level

heuristic does not significantly impact the quality of the ordering. It only reduces it by a few percent in 11 cases over the 104 matrices tested, while giving the same quality in all other cases.

In addition, we observed that for matrices not issued from meshes, with an unbalanced elimination tree, using the multi-level heuristic can deteriorate the solution. Indeed, in this case, the multi-level heuristic is unable to distinguish close interactions from far interactions with only the first levels, leading to incorrect choices.

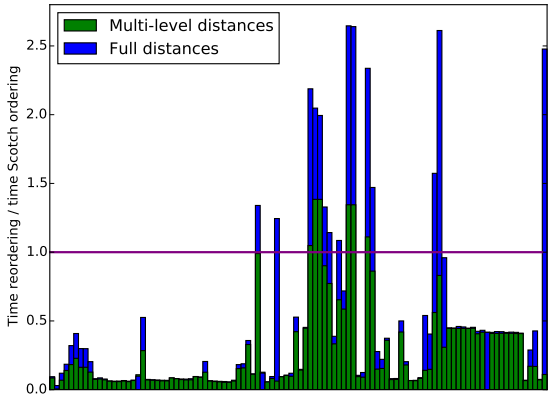


Figure 10: Time of the sequential reordering step with respect to the initial SCOTCH ordering on the Florida set of matrices.

Time

Figure 10 presents the sequential reordering cost with respect to the initial ordering performed with SCOTCH with both distance heuristics: full distance computation in blue, and multi-level computation in green. The reordering is in fact an extra step in the preprocessing stage of sparse direct solvers. One can note that despite the higher theoretical complexity, the reordering step is faster than SCOTCH, creating an overhead of 10% to 50% in most cases. However, on specific matrix structures, with a lot of connections to the last supernode, the reordering operation can be twice as expensive as SCOTCH. In those cases, the overhead is largely reduced by the multi-level heuristic, which reduces the time of the reordering step to the same order as SCOTCH. We observe that the multi-level heuristic is always beneficial to the computational time.

Strategy	Number of blocks		Time (s)	
	Full	Multi-level	Full	Multi-level
No reordering	9760700		360	
Reordering / Stop= ∞	3891825	3892522	64.8	47.7
Reordering / Stop= 10	4100616	4095986	33.2	**31.1**
Reordering / Stop= 20	3896248	3897179	42.6	38.5
Reordering / Stop= 30	**3891210**	3891262	50.7	43.3
Reordering / Stop= 40	3891803	3891962	58.1	46.3

Table 2: Number of off-diagonal blocks and reordering times on the CEA 10 million unknowns matrix.

Stopping criteria

Table 2 shows the impact of the stopping criteria on a large test case issued from a 10 million unknowns matrix from the CEA. The first line presents the results without reordering and the time of the SCOTCH step. We compared this to the number of off-diagonal blocks and the time obtained with our reordering algorithm when using different heuristics. The `STOP` parameter refers to the criteria introduced in section 3.4 and defines after how many differences a distance computation must be stopped. One can notice that with all configurations the quality is within 39% to 42% of the original, which means that those heuristics have a low impact on the quality of the result. However, this can have a large impact on the time to solution, since a small `STOP` criterion combined with the multi-level heuristic can divide the computational time by more than 2.

In conclusion, we can say that for a large set of sparse matrices, we obtain a resulting number of off-diagonal blocks between two and three times smaller than the original SCOTCH RCM ordering. It is interesting as it should reduce by the same factor the overhead associated to the tasks management in runtimes, and should improve the kernel efficiency of the solver. Up to our experiments, we reach a practical complexity close to SCOTCH ordering process, leading to a preprocessing stage that is not too costly compared to the numerical factorization. Furthermore, it should accelerate the numerical factorization and solve steps to hide this extra cost when only one numerical step is made, and give some global improvement when multiple factorizations or solves are performed.

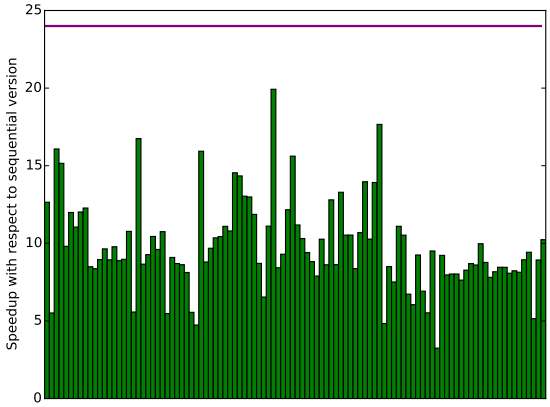


Figure 11: Speedup of the reordering step with 24 threads on the full set of matrices.

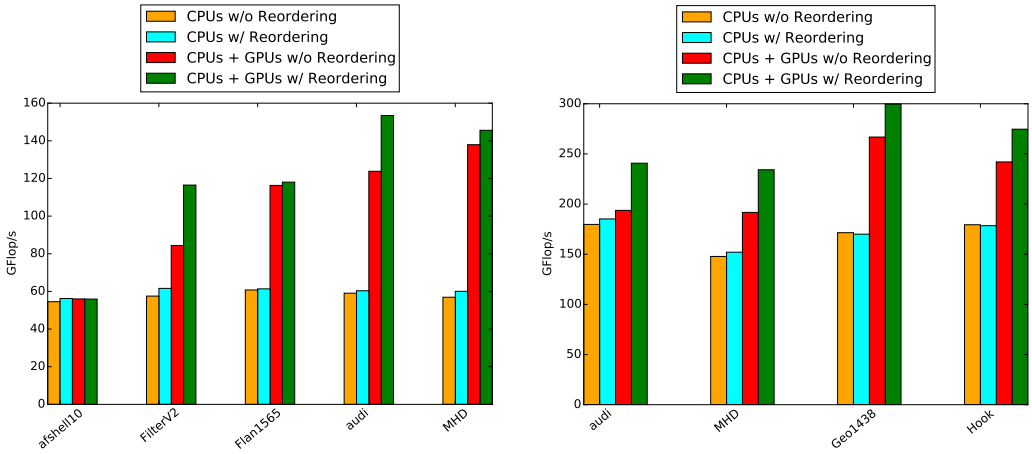
Parallelism

As previously stated, the reordering algorithm is largely parallel as each supernode can be reordered independently of the others. The first level of parallelism is a dynamic bin-packing that distributes supernodes in reverse order of their sizes. However, some supernodes are too large and take too long to be reordered compared to all others. They represent almost all the computational requirements. We then divided the set of supernodes into two parts. For the smaller set, we just reorder different supernodes in parallel, and for the larger set, we parallelize the distance matrix computation. Figure 11 shows the speedup obtained with 24 threads over the best sequential version on a `miriel` node. This simple parallelization accelerates the algorithm by 10 on average and helps to totally hide the cost of the reordering step in a multi-threaded

context, where ordering tools are hard to parallelize. Note that for many matrices, the parallel implementation of our reordering strategy has an execution time smaller than $1s$. In a few cases, the speed-up is still limited to 5 because the TSP problem on the largest supernode remains sequential and may represent a large part of the sequential execution.

4.3 Impact on supernodal method: PaStiX

In this section, we measure the performance gain brought by our reordering strategy. For these experiments, we extracted 7 matrices from the previous collection, and we use the number of operations (FLOPs) that a scalar algorithm would require to factorize the matrix with the ordering returned by Scotch to compute the performance of our solver.



(a) On 1 node (b) On 4 nodes

Figure 12: Performance impact of the reordering algorithm on the PaStiX solver on top of the StarPU runtime with 1 and 4 nodes of the hybrid Curie architecture.

Figure 12(a) presents the performance on a single `Curie` node with the matrices that fit into the memory of one node. `CPU` runs correspond to the use of PaStiX with 8 threads. In the context of `CPUs+GPUs` runs, as we use the StarPU version, 2 cores are dedicated to the management of the GPUs, and the 6 remaining cores are used for computations on the CPUs. Figure 12(b) presents the same performance study on the largest cases in the context of distributed architectures with 4 nodes. `afshell10`, `FilterV2`, and `Flan1565` have been removed because they are too small to exploit parallelism in a distributed context. To explain the different performance gains, we rely on table 3, which presents the average off-diagonal block heights with and without reordering. The block width is defined by the Scotch partition and is the same for all experiments on each matrix. This number is important, as it is especially interesting to enlarge blocks when the original solution provides small data blocks.

For both shared and distributed multi-threaded runs, the reordering gives a slight benefit up to 5% on the performance. Indeed, on the selected matrices, the original off-diagonal block height is already large enough to get a good CPU efficiency since the original solver already runs at 70% of the theoretical peak of the node (85.12 GFlop/s). In general, when the solver exploits GPUs, the benefit is more important and can reach 40% for the `FilterV2` matrix.

In figure 12(a), we can see that with the `afshell10` matrix, extracted from a 2D application,

Matrix	w/o reordering	w/ reordering
afshell10	45.072117	45.671155
FilterV2	8.759254	19.395490
Flan1565	27.773377	55.786989
audi	16.882257	37.218162
MHD	16.277483	26.169013
Geo1438	17.640077	43.460251
Hook	14.617559	27.632099

Table 3: Average off-diagonal block heights with or without the reordering algorithm.

the reordering has a low impact on the performance, and the accelerators are also not helpful for this lower computation case. On other problems, issued from 3D applications, we observe different behaviors even if the table 3 always presents an interesting gain in terms of block size. For the `Flan1565` matrix, there is no gain because the original off-diagonal block size is large enough for efficiency. The gain appears in memory, for the tasks management. On the other hand, one can observe a large performance gain on the `FilterV2` matrix, because the original block size is really small. For the `audi` and `MHD` matrices, SCOTCH ordering provides a similar average off-diagonal block size, as well as a comparable performance. As our reordering step better improves the blocking size of the `audi` matrix, the resulting performance gain – with respect to the original PASTIX implementation – is higher.

On distributed architectures, figure 12(b), one can note that it is difficult to get a significant speedup from the use of accelerators when there are already a large number of CPUs (32). However, the reordering always improves the overall performance on heterogeneous runs by approximately 15%, proving that increasing the off-diagonal block sizes allows for taking advantage of the accelerators.

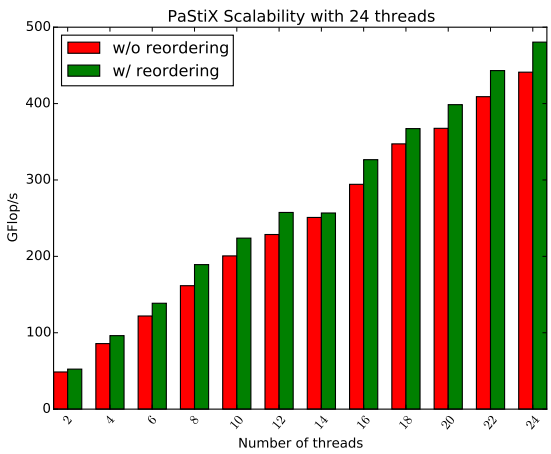


Figure 13: Scalability on the CEA 10 million unknowns matrix with 24 threads.

Figure 13 presents a scalability study on one `miriel` node with 24 threads, with and without our reordering stage on the 10 million unknowns matrix from the CEA. This matrix, despite being a large 3D problem, presents a really small average block size of less than 5 when no reordering is applied. The reordering algorithm raises up to 12.5, explaining the larger average

gain of $8 - 10\%$ that is observed. In both cases, we notice that the solver manages to scale correctly over the 24 threads, and even a little better when the reordering is applied. A slight drop in the performance on 14 threads is explained by the overflow on the second socket.

5 Conclusion

We presented a new reordering strategy, that – according to our experiments – succeeds in reducing the number of off-diagonal blocks in the block-symbolic factorization. It allows one to significantly improve the performance of GPU kernels, and the BLAS CPU kernels in smaller ratios, as well as reduces the number of tasks when using a runtime system. The resulting gain can be up to 40%, especially when tests are performed on NVIDIA Fermi architectures. Such an improvement is significant, as long as it is difficult to reach a good level of performance with sparse algebra on accelerators. This gain can be observed on both the factorization and the solve steps.

Furthermore, we proposed a parallel implementation of our reordering strategy, leading to a computational cost that is really low with respect to the numerical factorization and that is counterbalanced by the gain on the factorization. In addition, if multiple factorizations are applied on the same structure, this benefits the multiple factorization and solve steps at no extra cost. We proved that such a preprocessing stage is cheap in the context of 3D graphs of bounded degree, and showed that it works well for a large set of matrices.

For future work, we plan to study the impact of our reordering strategy in a multifrontal context with the MUMPS [30] solver and compare it with the solution studied in [17] that performs the permutation during the factorization. This technique is also important in the objective of integrating variable size batched operations currently under development for the modern GPU architectures. Finally, one of the most important perspectives is to exploit this result to guide matrix compression methods in diagonal blocks for using hierarchical matrices in sparse direct solvers. Indeed, considering the diagonal block by itself for compression without external contributions leads to incorrect compression schemes. Using the reordering algorithms to guide the compression helps to gather contributions corresponding to similar far or close interactions.

Acknowledgments

This material is based upon work supported by Bordeaux INP and the DGA under a DGA/Inria grant. Experiments presented in this paper were carried out using the PLAFRIM, GENCI and TGCC/Curie experimental platforms.

References

[1] A. George, "Nested dissection of a regular finite element mesh," *SIAM Journal on Numerical Analysis*, vol. 10, no. 2, pp. 345–363, 1973.

[2] P. Charrier and J. Roman, "Algorithmic study and complexity bounds for a nested dissection solver," *Numerische Mathematik*, vol. 55, no. 4, pp. 463–476, Jul. 1989.

[3] J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990.

[4] X. Lacoste, M. Faverge, P. Ramet, S. Thibault, and G. Bosilca, "Taking Advantage of Hybrid Systems for Sparse Direct Solvers via Task-Based Runtimes," in *Parallel Distributed*

Processing Symposium Workshops (IPDPSW), 2014 IEEE International. Phoenix, United States: IEEE, May 2014, pp. 29–38.

[5] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Multifrontal QR factorization for multicore architectures over runtime systems," in *Euro-Par 2013 Parallel Processing*, ser. Lecture Notes in Computer Science, F. Wolf, B. Mohr, and D. Mey, Eds. Springer Berlin Heidelberg, 2013, vol. 8097, pp. 521–532.

[6] P. Hénon, P. Ramet, and J. Roman, "PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems," *Parallel Computing*, vol. 28, no. 2, pp. 301–321, Jan. 2002.

[7] P. R. Amestoy, T. A. Davis, and I. S. Duff, "An approximate minimum degree ordering algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 4, pp. 886–905, 1996.

[8] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *Proceedings of the IEEE*, vol. 55, no. 11, pp. 1801–1809, Nov. 1967.

[9] R. Luce and E. G. Ng, "On the minimum flops problem in the sparse cholesky factorization," *SIAM Jouranl on Matrix Analysis and Applications*, vol. 35, no. 1, pp. 1–21, 2014.

[10] J. W. H. Liu, E. G. Ng, and B. W. Peyton, "On finding supernodes for sparse matrix computations," *SIAM Journal on Matrix Analysis and Applications*, vol. 14, no. 1, pp. 242–252, 1993.

[11] J. W. H. Liu, "The role of elimination trees in sparse factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 1, pp. 134–172, 1990.

[12] F. Pellegrini, "Scotch and libScotch 5.1 User's Guide," Aug. 2008, 127 pages.

[13] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[14] A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite.* Prentice Hall Professional Technical Reference, 1981.

[15] A. George and D. R. McIntyre, "On the application of the minimum degree algorithm to finite element systems," *SIAM Journal on Numerical Analysis*, vol. 15, no. 1, pp. 90–112, 1978.

[16] STFC, "HSL. A collection of Fortran codes for large scale scientific computation." http://www.hsl.rl.ac.uk/.

[17] S. L. M. Wissam, "Scaling the solution of large sparse linear systems using multifrontal methods on hybrid shared-distributed memory architectures," Ph.D. dissertation, Ecole normale supérieure de lyon - ENS LYON, Dec. 2014.

[18] R. W. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, vol. 26, no. 2, pp. 147–160, 1950.

[19] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics).* Princeton, NJ, USA: Princeton University Press, 2007.

[20] D. S. Johnson and L. A. Mcgeoch, *The Traveling Salesman Problem: A Case Study in Local Optimization*, 1997.

[21] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," DTIC Document, Tech. Rep., 1976.

[22] D. J. Rosenkrantz, R. E. Stearns, and P. M. L. II, "An analysis of several heuristics for the traveling salesman problem." *SIAM J. Comput.*, vol. 6, no. 3, pp. 563–581, 1977.

[23] U. of Waterloo, "Concorde TSP solver," http://www.math.uwaterloo.ca/tsp/concorde.html.

[24] G. L. Miller and S. A. Vavasis, "Density graphs and separators," in *Second Annual ACM-SIAM Symposium on Discrete Algorithms*, 1991, pp. 331–336.

[25] G. L. Miller, S.-H. Teng, and S. A. Vavasis, "A unified geometric approach to graph separators," in *Proc. 31st Annual Symposium on Foundations of Computer Science*, 1991, pp. 538–547.

[26] R. J. Lipton and R. E. Tarjan, "A separator theorem for planar graphs," *SIAM Journal on Applied Mathematics*, vol. 36, pp. 177–189, 1979.

[27] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[28] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Proceedings of the 15th International Euro-Par Conference, LNCS*, vol. 5704. Delft, The Netherlands: Springer, Aug. 2009, pp. 863–874.

[29] X. Lacoste, "Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu cluster systems," Ph.D. dissertation, Université Bordeaux, Talence, France, Feb. 2015.

[30] P. R. Amestoy, A. Buttari, I. S. Duff, A. Guermouche, J. L'Excellent, and B. Uçar, "MUMPS," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer, 2011, pp. 1232–1238.

Group	Name	n	nnz_A	real	2D/3D	spd	kind
AMD	G3_circuit	1585478	7660826	yes	no	yes	circuit simulation
Andrianov	lp1	534388	1643420	yes	no	no	optimization problem
ATandT	pre2	659033	5834044	yes	no	no	circuit simulation
Bodendiek	CurlCurl_2	806529	8921789	yes	yes	no	model reduction problem
Bodendiek	CurlCurl_3	1219574	13544618	yes	yes	no	model reduction problem
Bodendiek	CurlCurl_4	2380515	26515867	yes	yes	no	model reduction problem
Bourchtein	atmosmodd	1270432	8814880	yes	yes	no	fluid dynamics
Bourchtein	atmosmodj	1270432	8814880	yes	yes	no	fluid dynamics
Bourchtein	atmosmodl	1489752	10319760	yes	yes	no	fluid dynamics
CEMW	tmt_unsym	917825	4584801	yes	yes	no	electromagnetics problem
CEMW	tmt_sym	726713	5080961	yes	yes	yes	electromagnetics problem
Chevron	Chevron4	711450	6376412	no	no	no	other problem
DIMACS10	333SP	3712815	22217266	yes	no	no	undirected graph
DIMACS10	AS365	3799275	22736152	yes	no	no	undirected graph
DIMACS10	M6	3501776	21003872	yes	no	no	undirected graph
DIMACS10	NACA0015	1039183	6229636	yes	no	no	undirected graph
DIMACS10	NLR	4163763	24975952	yes	no	no	undirected graph
DIMACS10	adaptive	6815744	27248640	yes	no	no	undirected graph
DIMACS10	belgium_osm	1441295	3099940	yes	no	no	undirected graph
DIMACS10	channel-500x100x100-b050	4802000	85362744	yes	no	no	undirected graph
DIMACS10	delaunay_n19	524288	3145646	yes	no	no	undirected graph
DIMACS10	delaunay_n20	1048576	6291372	yes	no	no	undirected graph
DIMACS10	delaunay_n21	2097152	12582816	yes	no	no	undirected graph
DIMACS10	delaunay_n22	4194304	25165738	yes	no	no	undirected graph
DIMACS10	delaunay_n23	8388608	50331568	yes	no	no	undirected graph
DIMACS10	great-britain_osm	7733822	16313034	yes	no	no	undirected graph
DIMACS10	hugetrace-00000	4588484	13758266	yes	no	no	undirected graph
DIMACS10	hugetric-00000	5824554	17467046	yes	no	no	undirected graph
DIMACS10	hugetric-00010	6592765	19771708	yes	no	no	undirected graph
DIMACS10	hugetric-00020	7122792	21361554	yes	no	no	undirected graph
DIMACS10	italy_osm	6686493	14027956	yes	no	no	undirected graph
DIMACS10	netherlands_osm	2216688	4882476	yes	no	no	undirected graph
DIMACS10	packing-500x100x100-b050	2145852	34976486	yes	no	no	undirected graph
DIMACS10	rgg_n_2_19_s0	524288	6539532	yes	no	no	undirected random graph
DIMACS10	rgg_n_2_20_s0	1048576	13783240	yes	no	no	undirected random graph
DIMACS10	rgg_n_2_21_s0	2097152	28975990	yes	no	no	undirected random graph
DIMACS10	rgg_n_2_22_s0	4194304	60718396	yes	no	no	undirected random graph
DIMACS10	rgg_n_2_23_s0	8388608	127002786	yes	no	no	undirected random graph
DIMACS10	venturiLevel3	4026819	16108474	yes	no	no	undirected graph
Dziekonski	dielFilterV2clx	607232	25309272	no	yes	no	electromagnetics problem
Dziekonski	dielFilterV2real	1157456	48538952	yes	yes	no	electromagnetics problem
Dziekonski	dielFilterV3real	1102824	89306020	yes	yes	no	electromagnetics problem
Dziekonski	gsm_106857	589446	21758924	yes	yes	no	electromagnetics problem
Fluorem	HV15R	2017169	283073458	yes	yes	no	fluid dynamics
Freescale	Freescale1	3428755	17052626	yes	no	no	circuit simulation
Freescale	Freescale2	2999349	14313235	yes	no	no	circuit simulation matrix
Freescale	FullChip	2987012	26621983	yes	no	no	circuit simulation
Freescale	circuit5M	5558326	59524291	yes	no	no	circuit simulation
Freescale	circuit5M_dc	3523317	14865409	yes	no	no	circuit simulation
Freescale	memchip	2707524	13343948	yes	no	no	circuit simulation
GHS_psdef	apache2	715176	4817870	yes	yes	yes	structural problem
GHS_psdef	audikw_1	943695	77651847	yes	yes	yes	structural problem

GHS_psdef	inline_1	503712	36816170	yes	yes	yes	structural problem
GHS_psdef	ldoor	952203	42493817	yes	yes	yes	structural problem
Janna	Bump_2911	2911419	127729899	yes	yes	yes	2D/3D problem
Janna	Cube_Coup_dt0	2164760	124406070	yes	yes	no	structural problem
Janna	Cube_Coup_dt6	2164760	124406070	yes	yes	no	structural problem
Janna	Emilia_923	923136	40373538	yes	yes	yes	structural problem
Janna	Fault_639	638802	27245944	yes	yes	yes	structural problem
Janna	Flan_1565	1564794	114165372	yes	yes	yes	structural problem
Janna	Geo_1438	1437960	60236322	yes	yes	yes	structural problem
Janna	Hook_1498	1498023	59374451	yes	yes	yes	structural problem
Janna	Long_Coup_dt0	1470152	84422970	yes	yes	no	structural problem
Janna	Long_Coup_dt6	1470152	84422970	yes	yes	no	structural problem
Janna	ML_Geer	1504002	110686677	yes	yes	no	structural problem
Janna	PFlow_742	742793	37138461	yes	yes	yes	2D/3D problem
Janna	Queen_4147	4147110	316548962	yes	yes	yes	2D/3D problem
Janna	Serena	1391349	64131971	yes	yes	yes	structural problem
Janna	StocF-1465	1465137	21005389	yes	yes	yes	fluid dynamics
Janna	Transport	1602111	23487281	yes	yes	no	structural problem
Mazaheri	bundle_adj	513351	20207907	yes	yes	yes	computer vision proble
McRae	ecology1	1000000	4996000	yes	yes	no	2D/3D problem
McRae	ecology2	999999	4995991	yes	yes	yes	2D/3D problem
Oberwolfach	bone010	986703	47851783	yes	yes	yes	model reduction proble
Oberwolfach	boneS10	914898	40878708	yes	yes	yes	model reduction proble
Rajat	rajat29	643994	3760246	yes	no	no	circuit simulation
Rajat	rajat30	643994	6175244	yes	no	no	circuit simulation
Rajat	rajat31	4690002	20316253	yes	no	no	circuit simulation
Sandia	ASIC_680k	682862	2638997	yes	no	no	circuit simulation
Sandia	ASIC_680ks	682712	1693767	yes	no	no	circuit simulation
Schenk	nlpkkt120	3542400	95117792	yes	no	no	optimization problem
Schenk	nlpkkt160	8345600	225422112	yes	no	no	optimization problem
Schenk	nlpkkt80	1062400	28192672	yes	no	no	optimization problem
Schenk_AFE	af_0_k101	503625	17550675	yes	yes	yes	structural problem
Schenk_AFE	af_1_k101	503625	17550675	yes	yes	yes	structural problem
Schenk_AFE	af_2_k101	503625	17550675	yes	yes	yes	structural problem
Schenk_AFE	af_3_k101	503625	17550675	yes	yes	yes	structural problem
Schenk_AFE	af_4_k101	503625	17550675	yes	yes	yes	structural problem
Schenk_AFE	af_5_k101	503625	17550675	yes	yes	yes	structural problem
Schenk_AFE	af_shell1	504855	17562051	yes	yes	no	structural problem seque
Schenk_AFE	af_shell10	1508065	52259885	yes	yes	no	structural problem
Schenk_AFE	af_shell2	504855	17562051	yes	yes	no	structural problem
Schenk_AFE	af_shell3	504855	17562051	yes	yes	yes	structural problem
Schenk_AFE	af_shell4	504855	17562051	yes	yes	yes	structural problem
Schenk_AFE	af_shell5	504855	17579155	yes	yes	no	structural problem
Schenk_AFE	af_shell6	504855	17579155	yes	yes	no	structural problem
Schenk_AFE	af_shell7	504855	17579155	yes	yes	yes	structural problem
Schenk_AFE	af_shell8	504855	17579155	yes	yes	yes	structural problem
Schenk_AFE	af_shell9	504855	17588845	yes	yes	no	structural problem
Schmid	thermal2	1228045	8580313	yes	yes	yes	thermal problem
Sinclair	3Dspectralwave	680943	30290827	no	yes	no	materials problem
Williams	webbase-1M	1000005	3105536	yes	no	no	weighted directed grap
Wissgott	parabolic_fem	525825	3674625	yes	yes	yes	fluid dynamics
Zaoui	kkt_power	2063494	12771361	yes	no	no	optimization problem

Table 4: Matrices from Florida collection with $500000 \leq n \leq 10000000$

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

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

ISSN 0249-6399