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. SIAM Journal on Matrix Analysis and Applications, 2017, SIAM Journal on Matrix Analysis and Applications, 38 (1), pp.226 - 248. 10.1137/16M1062454 . hal-01485507v2

HAL Id: hal-01485507
https://inria.hal.science/hal-01485507v2

Submitted on 5 Apr 2017

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.

SIAM J. Matrix Anal. Appl.
Vol. 38, No. 1, pp. 226–248
© 2017 Society for Industrial and Applied Mathematics

REORDERING STRATEGY FOR BLOCKING OPTIMIZATION IN SPARSE LINEAR SOLVERS*

GREGOIRE PICHON[†], MATHIEU FAVERGE[‡], PIERRE RAMET[†], AND JEAN ROMAN[†]

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 computation has become more important than ever. In this paper, we present a reordering strategy that increases this block granularity. This strategy relies on 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 20% on GPUs.

Key words. sparse block linear solver, nested dissection, sparse matrix ordering, heterogeneous architectures

AMS subject classifications. 05C50, 65F05, 65F50, 68Q25

DOI. 10.1137/16M1062454

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 be considered as either 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 two- (2D) and three- (3D) dimensional finite element or finite volume meshes. The main issue with such factorizations is the fill-in—zero entries becoming nonzero—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 ordering of the unknowns that aims at minimizing the fill-in to limit the memory overhead and floating point operations (Flops) required to complete the factorization. The problem is then transformed into $(PAP^t)(Px) = Pb$,

*Received by the editors February 22, 2016; accepted for publication (in revised form) by P. A. Knight December 22, 2016; published electronically March 23, 2017.
http://www.siam.org/journals/simax/38-1/M106245.html
Funding: This work was funded by Bordeaux INP and the DGA under a DGA/Inria grant.
[†]Bordeaux INP, CNRS (Labri UMR 5800), Inria, University of Bordeaux, Talence, France (gregoire.pichon@inria.fr, pierre.ramet@inria.fr, jean.roman@inria.fr).
[‡]Bordeaux INP, CNRS (Labri UMR 5800), Inria, University of Bordeaux, Talence, France, and ICL, University of Tennessee, Knoxville, TN 37996 (mathieu.faverge@inria.fr).

where P is an orthogonal permutation matrix. A wide array of literature exists on solutions to graph reordering problems, the nested dissection recursive algorithm introduced by George [10] being the most commonly used solution for sparse direct factorization.

The second preprocessing step of sparse direct solvers is block-symbolic factorization [6]. 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 [9], on those blocks instead of scalar operations. For this purpose, extra fill-in, and by extension 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 and vector units, providing 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 constant the amount of fill-in generated during the factorization. So the amount of memory and computation used 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 [1, 19], increasing the performance of BLAS kernels.

Section 2 gives a brief background on block-symbolic factorization for sparse direct solvers and introduces the problem when classical reordering techniques are used. Section 3 states the problem, describes our reordering strategy, and gives an upper bound of its theoretical cost for the class of graphs with bounded degree, that is, graphs from real-life 2D or 3D numerical simulations. The quality of the symbolic structure and its impact on the performance of a sparse direct solver, here PaStiX [14], are 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 fill-in characterization theorem (see Theorem 2.1) to minimize the fill-in, zeros becoming nonzeros during factorization, and the second one predicts the block structure that will facilitate efficient numerical factorization and solve. Steps 3 and 4 perform the actual computation.

THEOREM 2.1 (fill-in characterization theorem from [27]). *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 nonzero 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 [3] and the minimum local fill (MF) algorithm [23, 31]. 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 [10] 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.

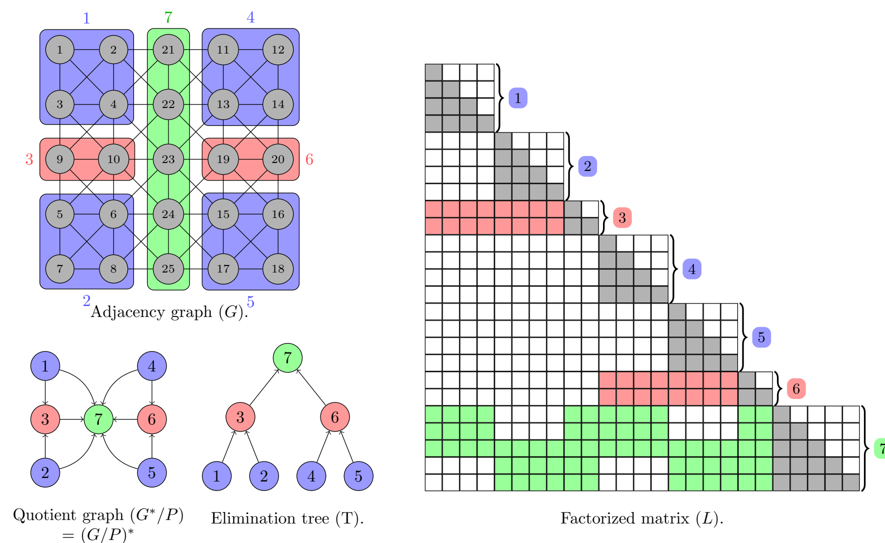


FIG. 2.1. *Nested dissection and block-data structure for L. (Color available online.)*

The top-left part of Figure 2.1 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 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 nonseparated subgraphs (by using the algorithm introduced in [21, 22]). This partitioning and ordering operation is usually performed through an external tool such as METIS [17] or SCOTCH [26].

Given this supernodal partition, one can compute the block-symbolic data struc-

ture of the factorized matrix, as presented on the right part of Figure 2.1. 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 [9]. 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 [6]. 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 ancestors. Finally, before distributing the column blocks on the processors, the biggest column blocks corresponding to the topmost supernodes in the tree are split in order to exploit the parallelism inside the dense computations [14].

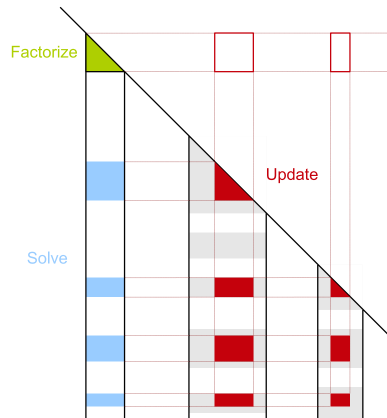


Fig. 2.2. *Steps to factorize the matrix: symmetric positive definite case. (Color available online.)*

The first two steps of a direct solver are preprocessing stages, independent of 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 2.2 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 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 2.2). 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 2.2). The temporary result is then scattered and added to the target column block. The last option, similar to what is done in a multifrontal solver, consists of a single matrix-matrix multiplication that is followed by a 2D scatter of the updates. In the last two options, 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 compact 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. Intranode 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 2.3. 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 at lower levels.

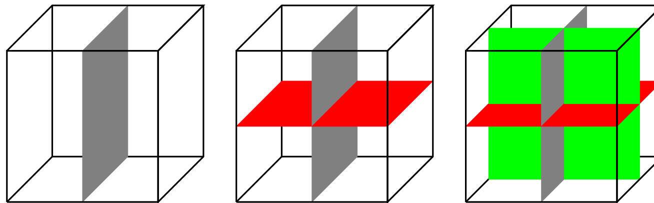
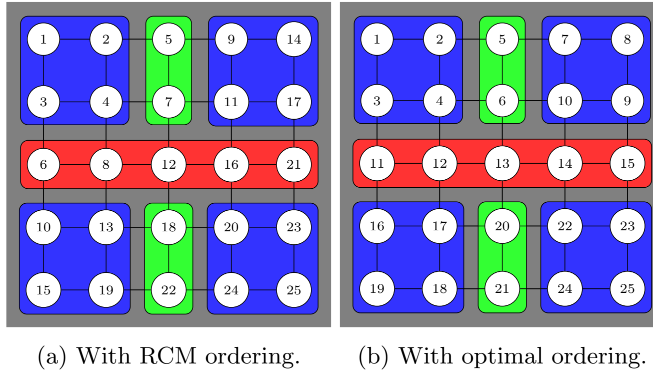


FIG. 2.3. *Three levels of nested dissection on a regular cube. (Color available online.)*

Inside each separator, vertices have to be ordered as well, and it is common to use techniques such as the reverse Cuthill–McKee [11] (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 2.1, this 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 complete knowledge of the local graph and of its outer interactions will lead to better quality in terms of the number of off-diagonal blocks.

Figure 2.4 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



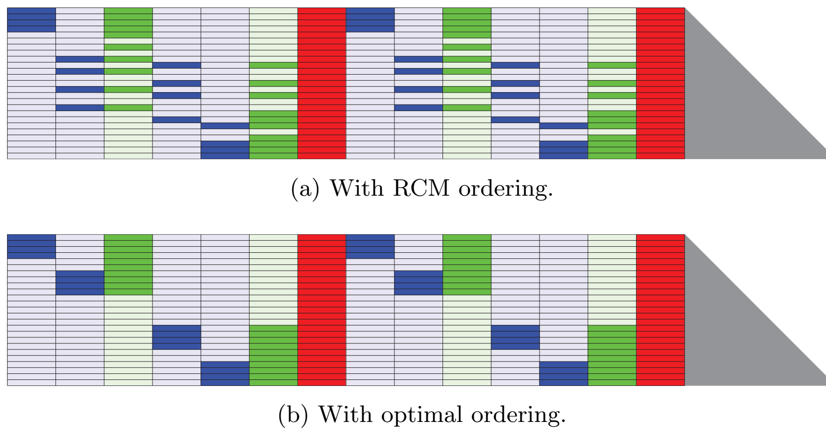
(a) With RCM ordering. (b) With optimal ordering.

Fig. 2.4. *Projection of contributing supernodes and ordering on the first separator (gray in Figure* 2.3*). (Color available online.)*

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 (see Figure 2.4(b)), computed by hand, as opposed to an RCM algorithm (see Figure 2.4(a)). This ordering is considered optimal as it minimizes the number of off-diagonal blocks to one per column. 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 the 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 block-symbolic factorization for these two orderings in Figures 2.5(a) and 2.5(b). The ordering proposed in Figure 2.4(b) is optimal in terms of number of off-diagonal blocks as long as it is impossible to exhibit a block-symbolic structure with less than 14 off-diagonal blocks: there is no more than one off-diagonal block per column block.

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. However, it often happens that the separators found by generic tools such as Metis or Scotch are disconnected graphs, making this previous statement incorrect as long as it is impossible to recover and follow the spatial neighborhood of the vertices (issued from the associated mesh).

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 2.6 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.



(a) With RCM ordering.

(b) With optimal ordering.

Fig. 2.5. *Off-diagonal blocks contributing to the first separator in Figure* 2.3.

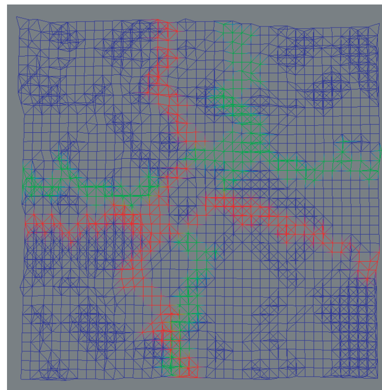


Fig. 2.6. *Projection of contributing supernodes on the first separator of a* 3*D Laplacian of size* $40 \times 40 \times 40$*, using* Scotch.

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, shown in Figure 2.6, 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.

2.3. Related works. Studying the structure of off-diagonal blocks was used in different contexts. In [12], 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 were applied to 2D graphs and successfully reduced 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 [15], the authors introduced reordering techniques in the context of both supernodal and multifrontal solvers (see the HSL library [30]). The objective is to create larger off-diagonal blocks to enhance data locality and reduce factorization time in the MA87 supernodal solver. Each diagonal block is reordered according to its set of children. Given a child and one of its ancestors, section 2.2 showed that the set of rows from the ancestor connected to the child should be ordered contiguously to create a single off-diagonal block and avoid scattering operations. HSL reordering strategy starts by sorting children according to their contribution size to the studied supernode, in other words with the number of rows that connects the two nodes. Then the rows connected to the larger child are numbered continuously to create a single off-diagonal block coming from this child, and the process is repeated on the remaining rows until all of them are reordered. This strategy has led to performance gains in a multithreaded context [15]. However, from construction, this algorithm gives priority to the largest branch of the elimination tree, neglecting the other ones. In practice, we observe that leaves from both sides of the elimination tree might be connected to the same unknowns of their ancestors. Thus, we can expect that an algebraic view of the problem will provide better results by using a global view of the connections of one supernode to all its descendants in the elimination tree.

In [29], a reordering strategy is proposed for the multifrontal solver MUMPS [2]. The objective is to provide a row ordering and the associated mapping on a set of processors to minimize the total volume of communications. The strategy studied minimizes communication between a parent and its set of children. As opposed to HSL and our solutions, this algorithm, designed for multifrontal solvers only, dynamically reorders the rows at each level of the elimination tree. It considers only interactions from children to a direct parent to minimize scattering operations at each level when updating the frontal matrix. This way rows from a parent node can have different orderings at each level of the elimination tree, or even between different child branches. Thus, it is beyond the scope of the global ordering proposed in this paper for a supernodal solver.

3. Improving the blocking size.

As presented in section 2.2, the RCM algorithm, widely used to order supernodes, generates many extra off-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 main idea is to consider the set of contributions for each row of a supernode, before using a distance metric to minimize the creation of off-diagonal blocks when permuting rows.

3.1. Problem modeling.

The strategy 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 recomputing 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 at 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

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

row^ℓ_{ik} 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. Otherwise, it is equal to 0, meaning that no nonzero element connects the two in the initial matrix, or no fill-in will create that connection. Let's now define for each vertex the binary vector $B^\ell_i = (row^\ell_{ik})_{k\in[\![1,\ell-1]\!]}$. We can then define w^ℓ_i, the weight of a row i, as in (2), which represents the number of supernodes contributing to that row i, and the distance between two rows i and j, $d^\ell_{i,j}$, as in (3). This is known as the Hamming distance [13] 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, $d^\ell_{i,j}$ represents the number of off-diagonal blocks 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:

$$(2)\qquad w^\ell_i = \sum_{k=1}^{\ell-1} row^\ell_{ik},$$

$$(3)\qquad d^\ell_{i,j} = d(B^\ell_i, B^\ell_j) = \sum_{k=1}^{\ell-1} row^\ell_{ik} \oplus row^\ell_{jk},$$

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

$$(4)\qquad odb^\ell = \frac{1}{2}\left(w^\ell_1 + \sum_{i=1}^{|C_\ell|-1} d^\ell_{i,i+1} + w^\ell_{|C_\ell|} \right),$$

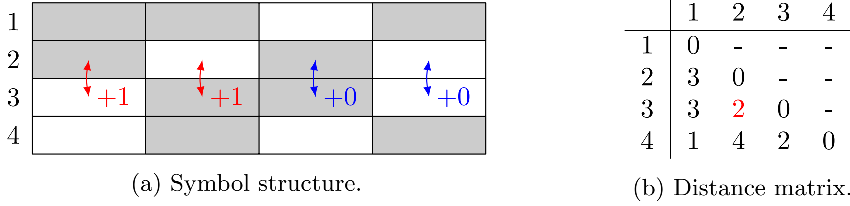
where the Hamming weights of the first and last rows of the supernode C_ℓ correspond, respectively, to the number of blocks in the first row and in the last one, and the distances between two consecutive rows give the evolution in the number of blocks when traveling through them.

Figure 3.1 illustrates the computation of the number of off-diagonal blocks with (4) on an example of four rows. The computation of the distance from the second to the third row is illustrated on the left: there are two differences, making it a distance of 2. Figure 3.1(b) summarizes the distances between each couple of rows in this example. With this information and the weight of the first and last rows, respectively, 3 and 2, one can compute the number of off-diagonal blocks, odb, from the formula: $\frac{1}{2}(w_1 + d_{1,2} + d_{2,3} + d_{3,4} + w_4) = \frac{1}{2}(3 + 3 + 2 + 2 + 2) = 6$.

Thus, 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 NP-hard.

3.2. Proposed heuristic. We first propose to introduce an extra virtual vertex, S_0, for which B_0 is the null set. Thus, we have

$$(5)\qquad \forall i \in [\![1, |C_\ell|]\!], \quad d^\ell_{0,i} = d^\ell_{i,0} = w_i.$$



(a) Symbol structure.

	1	2	3	4
1	0	-	-	-
2	3	0	-	-
3	3	2	0	-
4	1	4	2	0

(b) Distance matrix.

FIG. 3.1. *Example of a symbolic structure and its associated distance matrix. Computation of the distance between rows* 2 *and* 3.

The problem can now be transformed in a traveling salesman problem [4] (TSP):

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

which is also an NP-Hard problem, but for which multiple heuristics have been proposed in the literature [16], as opposed to shortest Hamiltonian path problem. Furthermore, our problem presents properties that make it suitable for better heuristics and theoretical models that bound the maximum distance to the optimal solution. First, our problem is symmetric, since

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

and second, it respects the triangular inequality

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

This sets our problem as a 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 3.1.

To keep a global complexity below that of the numerical factorization, we explain in section 3.3 that the complexity of the TSP algorithm has to remain equal to or lower than $\Theta(p^2)$, where p is the number of vertices in the cycle. Thus, it prevents advanced algorithms such as the Christofides algorithm [7] 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 guarantees a maximal distance to the optimal of at most 2 [28]. A quality comparison of our algorithm over 3D Laplacian matrices and real matrices against the CONCORDE [32] TSP solver that returns optimal solutions has shown that our nearest insertion algorithm provides results within less than 10% from the optimal.

TABLE 3.1
Complexity and quality of different TSP algorithms.

Algorithm	Complexity for p nodes	Quality (with respect to 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

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
```

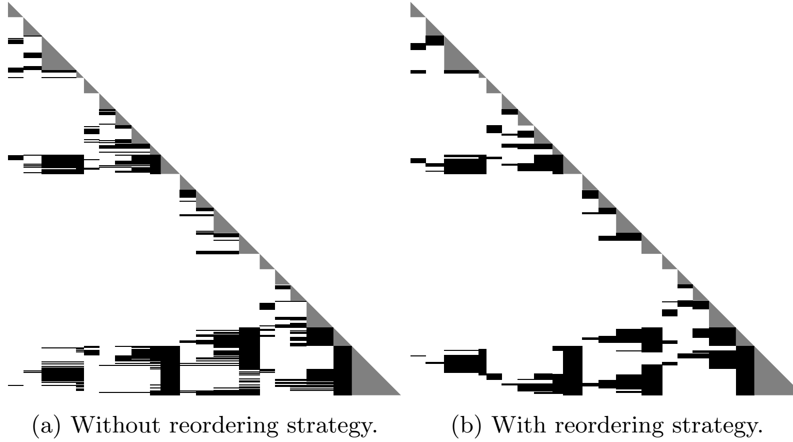
Our final algorithm is then decomposed in three stages, presented in Algorithm 1, that are applied to each separator of the nested dissection. Note that it is not applied on the leaves of the elimination tree since they will not receive contributions from other supernodes. The first step is to compute the B_i^ℓ vectors for each row i of the current separator. Then it computes the distance 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 (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, which provides a compressed storage of the information similar to the compressed 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 are in $\Theta(n)$. Note that we store the contributing supernode numbers in 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_{ij}^ℓ between rows i and j from C_ℓ, we take advantage of the sorted sets B_i^ℓ and B_j^ℓ to realize this computation in $\Theta(|B_i^\ell| + |B_j^\ell|)$ 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.

Figure 3.2 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 3.2(a), our reordering algorithm has not been applied, and supernode ordering results only



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

FIG. 3.2. *Block-symbolic factorization of* $8 \times 8 \times 8$ *Laplacian initially reordered with* SCOTCH.

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 3.2(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. For this study we consider graphs issued from finite element meshes coming from real-life simulations of 2D or 3D physical problems. From a theoretical point of view, the majority of those graphs have a bounded degree and are specific cases of bounded-density graphs [25]. 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 or, more generally, for overlap graphs [24]. In d dimensions, 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 [20] 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 [6]). *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 [6]. In that 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 an* n^σ*-separation theorem, the reordering algorithm complexity is bounded by* $\Theta(n^{\sigma+1})$*.*

Proof of Theorem 3.3. The main cost of the reordering algorithm is issued from the distance matrix computation. As presented in section 3.2, we compute a distance matrix 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 2.5(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

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

For graphs of bounded degree, this result leads to the following:

- 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 [6], we can compute the overall complexity as a recursive function depending on the complexity on one supernode. This 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 usually gives 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.

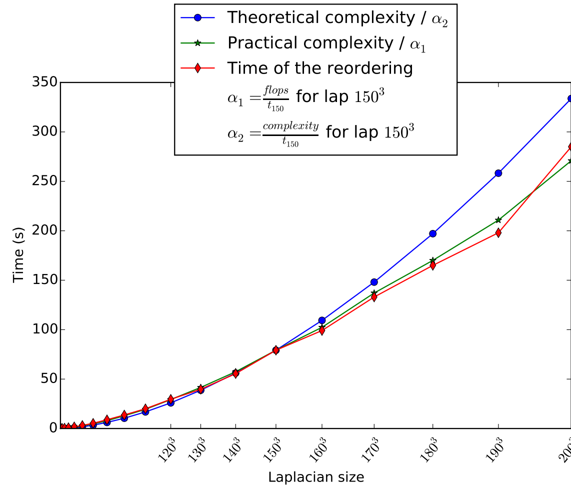


Fig. 3.3. *Comparison of the time of reordering against theoretical and practical complexities on 3D Laplacians. (Color available online.)*

Figure 3.3 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 (diamonds) represents 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 readability, so we can confirm that the trends of both curves are identical to a constant factor. Finally, in green (stars) 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 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 such as Metis and Scotch and has an impact on the global fill-in of the matrix. As presented before, the reordering stage takes part of the 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.

Multilevel: 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 2D 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-level* 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 to or larger than the *high-level* distance by construction of the elimination tree. Then we compute the *low-level* 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. This 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 exceeds a threshold parameter. This solution helps to quickly disregard the rows that are "far away" from each other. This limits the complexity of the distance computation, reducing the overall practical complexity. In most cases, a small value such as 10 can already provide good quality improvement. However, it depends on the graph properties and the average number of differences between rows. Unfortunately, if this heuristic is used alone, this improvement is not always guaranteed and it might lead to a degradation in quality. In association with the previous multilevel heuristic, the results are always improved, as we will see in the following section.

4. Experimental 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. We compare here three different strategies to the original ordering provided by the Scotch library. Two are based on our strategy, namely, TSP, with the full distance computation or with the multilevel approximation. The third, namely, HSL, is the one implemented in the HSL library for the MA87 supernodal solver (`optimize_locality` routine from MA87). Note that those three reordering strategies are applied to the partition found by Scotch, and they do not modify the fill-in.

4.1. Context.

We used a set of large matrices arising from real-life applications originating from the University of Florida's [8] 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.

We utilized the *Plafrim*[1] supercomputer for our experiments. For the performance experiments on a heterogeneous environment, we used the `mirage` cluster, where nodes are composed of two Intel Westmere Xeon X5650 hexa-core CPUs running at 2.67 GHz with 36 GB of memory, and enhanced by three NVIDIA GPUs, M2070. We used Intel MKL 2016.0.0 for the BLAS kernels on the CPUs, and we used the NVIDIA CUDA 7.5 development kit to compile the GPU kernels. For the scalability experiments in a multithreaded context, we used the `miriel` cluster. Each node is equipped with two Intel Xeon E5-2680 v3 12-cores running at 2.50 GHz with 128 GB of memory. The same version of the Intel MKL is used.

The PaStiX version used for our experiments is the one implemented on top of the Parsec [5] runtime system and presented in [19].

For the initial ordering step, we used Scotch 5.1.11 with the configurable strategy string from PaStiX to set the minimal size of nonseparated subgraphs, *cmin*, to be 20 as in [18]. We also set the *frat* parameter to 0.08, meaning that column 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 and the computational time of the three reordering algorithms, the two versions of our TSP and HSL, compared to the original ordering computed by Scotch that is known to be in $\Theta(n \times log(n))$. Note that sequential implementation is used for all algorithms, except in the subsection "Parallelism."

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 optimum is never reached and varies a lot from one matrix to another, so we stayed with the global number of off-diagonal blocks and its evolution compared to the original solution given by Scotch.

Quality. Figure 4.1 presents the quality of reordering strategies 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. Three metrics are represented: one with HSL reordering, one for our multilevel heuristic, and finally one for the full distance computation heuristic. We can see that our algorithm reduces the number of off-diagonal blocks in all test cases. In the 3D problems, our reordering strategy improves the metric by 50–60%, while in the 2D problems, the improvement is of 20–30%. Furthermore, we can observe that the multilevel 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. The HSL heuristic improves the initial ordering on average by approximately 20%, and up to 40%, but is outperformed by our TSP heuristic regardless of the multilevel distances approximation on all cases.

In addition, we observed that for matrices not issued from meshes with an un-

[1] https://plafrim.bordeaux.inria.fr

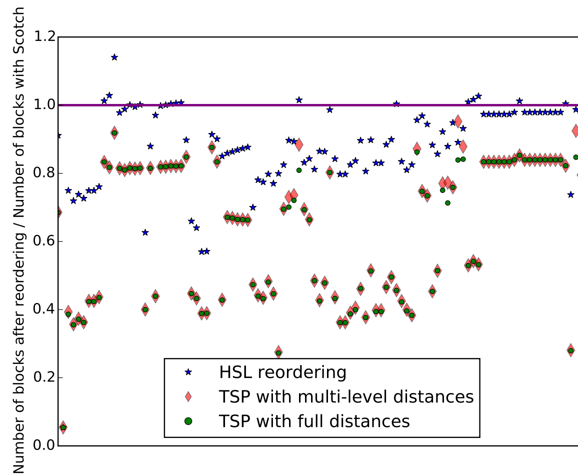


Fig. 4.1. *Impact of the heuristic used on the ratio of off-diagonal blocks over those produced by the initial* Scotch *ordering on the Florida set of matrices. The lower the better. (Color available online.)*

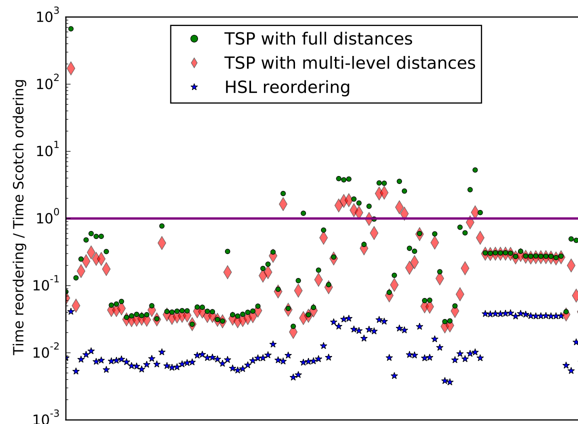


Fig. 4.2. *Time of the sequential reordering step with respect to the initial* Scotch *ordering on the University of Florida set of matrices. The lower the better. (Color available online.)*

balanced elimination tree, and not presented here, using the multilevel heuristic can deteriorate the solution. Indeed, in this case, the multilevel heuristic is unable to distinguish close interactions from far interactions with only the first levels, leading to incorrect choices.

Time. Figure 4.2 presents the cost of the three sequential reordering strategies with respect to the cost of the initial ordering performed by Scotch. 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 of our TSP heuristic is 2 to 10 times faster than Scotch. Thus, adding the reordering step creates a sequential overhead of no more than 10–50% 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 diminished by the multilevel heuristic, which reduces the time of the reordering step to the same order as SCOTCH. We observe that the multilevel heuristic is always beneficial to the computational time. For the second matrix—an optimization problem with a huge density on the left of the figures—we can observe a quality gain of more than 95%, while the cost is more than 600 times larger than the ordering time. This problem illustrates the limitation of our heuristic using a global view compared to the local heuristic of the HSL algorithm, which is still faster than the SCOTCH ordering but gives only 20% improvement. This problem is typically not suited for sparse linear solvers, due to its large number of nonzeros as well as its consequent fill-in.

TABLE 4.1
Number of off-diagonal blocks and reordering times on a CEA matrix with 10 *million unknowns.*

Strategy	Number of blocks		Time (s)	
	Full	Multilevel	Full	Multilevel
No reordering	9760700		360	
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
Reordering / Stop= ∞	3891825	3892522	64.8	47.7

Stopping criteria. Table 4.1 shows the impact of the stopping criteria on a large test case issued from a 10-million-unknown 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 39–42% 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 multilevel 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, while the HSL heuristic reduces them on average only by a fifth. 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. In our experiments, we reach a practical complexity close to the 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. While HSL reordering overhead might be much smaller than our heuristic, we hope that the difference in the quality gain, as well as the fact that our strategy improves all children instead of giving advantage to the largest one, will benefit the factorization step by a larger factor.

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

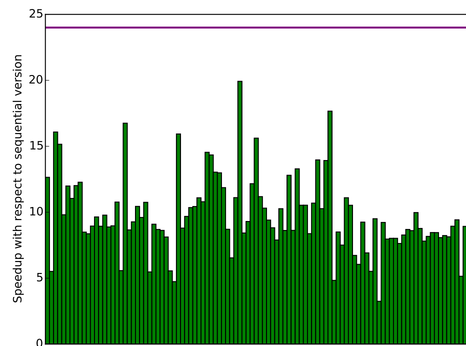


FIG. 4.3. *Speedup of the reordering step (full distance heuristic) with* 24 *threads on the full set of matrices.*

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 4.3 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 multithreaded 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 1 s. In a few cases, the speedup 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 the reordering strategies. For these experiments, we extracted 6 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. We recall that this number is stable with all reordering heuristics.

Figure 4.4 presents the performance on a single `mirage` node, for three algorithms based on the original ordering from Scotch. The first one leaves the Scotch ordering untouched. The HSL heuristic is applied on the second one, and finally the third one includes the TSP heuristic with full distances. For each matrix and each ordering, scalability of the numerical factorization is presented with all 12 cores of the architecture enhanced by 0 to 3 GPUs. All results are an average performance over five runs.

To explain the different performance gains, we rely on Table 4.2, which presents the average number of rows in the off-diagonal blocks with and without reordering, and not the total number of blocks to give some insight into the size of the updates. 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 beneficial to enlarge blocks when the original solution provides small data blocks.

For multithreaded runs, both reordering strategies give a slight benefit up to 7% 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

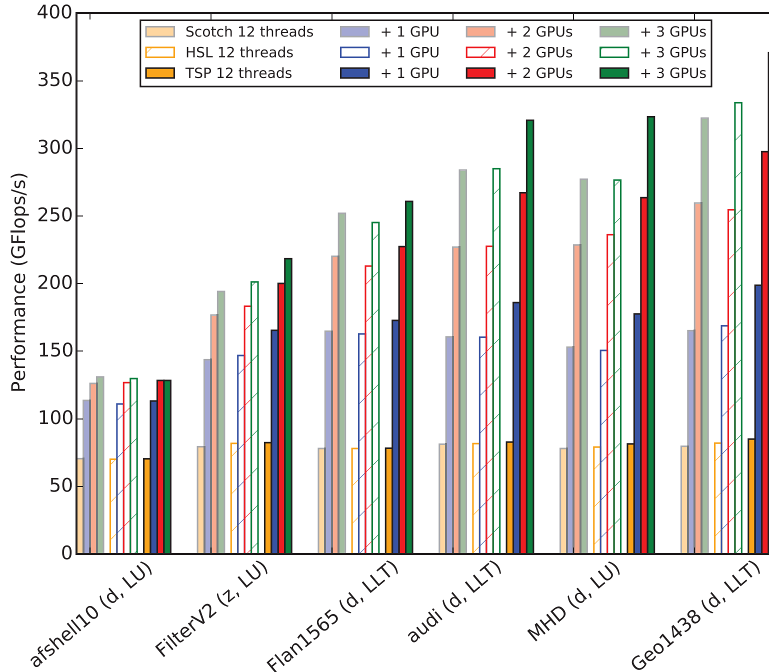


FIG. 4.4. *Performance impact of the reordering algorithms on the* PaStiX *solver on top of the* Parsec *runtime with* 1 *node of the hybrid mirage architecture.*

TABLE 4.2
Impact of the reordering strategies on the number of rows per off-diagonal block and on the timings. The timing shows the overhead of the parallel TSP and the gain it provides on the factorization step using the `mirage` *architecture.*

Matrix	Avg. number of rows of off-diagonal block			TSP times	
	Scotch	HSL	TSP	Overhead	Gain
afshell10	46.05	45.52	54.09	0.133 s	0 s
FilterV2	8.794	11.33	19.91	0.164 s	1.23 s
Flan1565	29.13	32.33	62.40	0.644 s	0.52 s
audi	17.94	20.57	41.76	0.748 s	2.08 s
MHD	16.86	17.04	27.64	1.16 s	4.42 s
Geo1438	18.79	23.17	49.74	1.78 s	7.48 s

already runs at up to 67% of the theoretical peak of the node (128.16 GFlop/s). This is also true for HSL reordering. In general, when the solver exploits GPUs, the benefit is more important and can reach up to 20%.

In Figure 4.4, we can see that with the `afshell10` matrix, extracted from a 2D application, reordering strategies have a low impact on the performance, and the accelerators are also not helpful for this lower computation case. For the `Flan1565` matrix, the gain is not important for both reordering strategies because the original off-diagonal block height is already large enough for efficiency. On other problems, issued from 3D applications, we observe significant gains from 10–20% which reflect the block height increase of 1.5 to 2.5 presented in Table 4.2.

If we compare this with the HSL reordering strategy, we can see that our reordering is helpful in slightly improving the performance of the solver. Hence, choosing

between both strategies depends on the number of factorizations that are performed.

Table 4.2 also presents our TSP reordering strategy overhead with the parallel implementation and the resulting gain on the numerical factorization time using the `mirage` architecture with the three GPUs. Those numbers reflect that it is interesting to use our reordering strategy in many cases with a small overhead that is immediately recovered by the performance improvement of the numerical factorization. As long as several problems presenting the same structure are solved, this small overhead is again diminished. Similarly, if GPUs are used, the gain during the factorization is higher, completely hiding the overhead of the reordering. The cost of the HSL strategy being really small with respect to SCOTCH, it is always recommended to apply it for a single factorization or for homogeneous computations. However, if GPUs are involved, HSL reordering impact on the performance of the numerical factorization is really slight, and it goes from a slight slowdown to a slight speedup (around 3%). This validates the use of more complex heuristics than the proposed TSP.

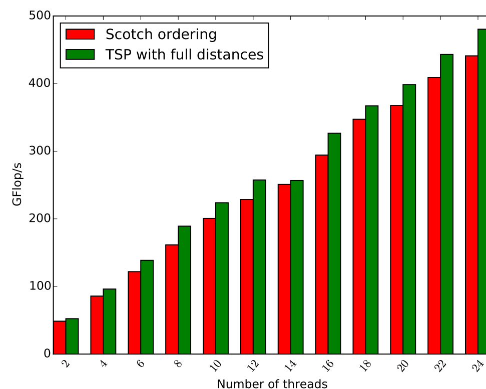


FIG. 4.5. *Scalability on the CEA* 10*-million-unknown matrix with* 24 *threads.*

Figure 4.5 presents a scalability study on one `miriel` node with 24 threads, with and without our reordering stage on the 10-million-unknown 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 rises 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 reducing the number of tasks when using a runtime system. The resulting gain can be up to 20% on heterogeneous architectures enhanced by 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. It works particularly well for graphs issued from finite element meshes of 3D problems. In the context of 2D graphs, partitioner tools can be sufficient, as long as separators are close to 1D structures and can easily be

ordered by following the neighborhood. For other problems, the strategy enhances the number of off-diagonal blocks, but might be costly on graphs where vertices have large degrees.

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. We compared it with HSL reordering, which targets the same objective of reducing the overall number of off-diagonal blocks. While the TSP heuristic is often more expensive, the quality is always improved, leading to better performance. In the context of multiple factorizations, or when using GPUs, the TSP overhead is recovered by performance improvement, while it may be better to use HSL for the other cases.

For future work, we plan to study the impact of our reordering strategy in a multifrontal context with the MUMPS [2] solver and compare it with the solution studied in [29], which performs the permutation during the factorization. The main difference with the static ordering heuristics studied in this paper is that the MUMPS heuristic is applied dynamically at each level of the elimination tree. Such a reordering 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.

Acknowledgment. Experiments presented in this paper were carried out using the PLAFRIM experimental platform.

REFERENCES

[1] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, *Multifrontal QR factorization for multicore architectures over runtime systems*, in Euro-Par 2013 Parallel Processing, Lecture Notes in Comput. Sci. 8097, F. Wolf, B. Mohr, and D. Mey, eds., Springer, Berlin, Heidelberg, 2013, pp. 521–532, https://doi.org/10.1007/978-3-642-40047-6_53.

[2] 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, https://doi.org/10.1007/978-0-387-09766-4_204.

[3] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 886–905, https://doi.org/10.1137/S0895479894278952.

[4] 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 University Press, Princeton, NJ, 2007.

[5] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, *PaRSEC: Exploiting heterogeneity to enhance scalability*, Comput. Sci. Engrg., 15 (2013), pp. 36–45.

[6] P. Charrier and J. Roman, *Algorithmic study and complexity bounds for a nested dissection solver*, Numer. Math., 55 (1989), pp. 463–476.

[7] N. Christofides, *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem*, tech. report, DTIC Document, 1976.

[8] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), 1, https://doi.org/10.1145/2049662.2049663.
[9] J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 1–17, https://doi.org/10.1145/77626.79170.
[10] A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363, https://doi.org/10.1137/0710032.
[11] A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite*, Prentice–Hall Professional Technical Reference, 1981.
[12] A. George and D. R. McIntyre, *On the application of the minimum degree algorithm to finite element systems*, SIAM J. Numer. Anal., 15 (1978), pp. 90–112, https://doi.org/10.1137/0715006.
[13] R. W. Hamming, *Error detecting and error correcting codes*, Bell System Tech. J., 26 (1950), pp. 147–160.
[14] P. Hénon, P. Ramet, and J. Roman, *PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems*, Parallel Comput., 28 (2002), pp. 301–321.
[15] J. D. Hogg, J. K. Reid, and J. A. Scott, *Design of a multicore sparse Cholesky factorization using DAGs*, SIAM J. Sci. Comput., 32 (2010), pp. 3627–3649, https://doi.org/10.1137/090757216.
[16] D. S. Johnson and L. A. McGeoch, *The traveling salesman problem: A case study in local optimization*, in Local Search in Combinatorial Optimization, Wiley, Chichester, UK, 1997, pp. 215–310.
[17] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392, https://doi.org/10.1137/S1064827595287997.
[18] X. Lacoste, *Scheduling and Memory Optimizations for Sparse Direct Solver on Multi-core/Multi-gpu Cluster Systems*, Ph.D. thesis, Université Bordeaux, Talence, France, 2015.
[19] 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 Proceedings of the 2014 IEEE International Parallel Distributed Processing Symposium Workshops (IPDPSW), Phoenix, AZ, IEEE, 2014, pp. 29–38.
[20] R. J. Lipton and R. E. Tarjan, *A separator theorem for planar graphs*, SIAM J. Appl. Math., 36 (1979), pp. 177–189, https://doi.org/10.1137/0136016.
[21] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172, https://doi.org/10.1137/0611010.
[22] J. W. H. Liu, E. G. Ng, and B. W. Peyton, *On finding supernodes for sparse matrix computations*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 242–252, https://doi.org/10.1137/0614019.
[23] R. Luce and E. G. Ng, *On the minimum FLOPs problem in the sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., 35 (2014), pp. 1–21, https://doi.org/10.1137/130912438.
[24] G. L. Miller, S.-H. Teng, and S. A. Vavasis, *A unified geometric approach to graph separators*, in Proceedings of the 31st Annual Symposium on Foundations of Computer Science, IEEE, 1991, pp. 538–547.
[25] G. L. Miller and S. A. Vavasis, *Density graphs and separators*, in Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, ACM, New York, 1991, pp. 331–336.
[26] F. Pellegrini, *Scotch and libScotch* 5.1 *User's Guide*, 2008.
[27] D. J. Rose and R. E. Tarjan, *Algorithmic aspects of vertex elimination on directed graphs*, SIAM J. Appl. Math., 34 (1978), pp. 176–197, https://doi.org/10.1137/0134014.
[28] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II, *An analysis of several heuristics for the traveling salesman problem*, SIAM J. Comput., 6 (1977), pp. 563–581, https://doi.org/10.1137/0206041.
[29] W. M. Sid-Lakhdar, *Scaling the Solution of Large Sparse Linear Systems Using Multifrontal Methods on Hybrid Shared-Distributed Memory Architectures*, Ph.D. thesis, École Normale Supérieure de Lyon, 2014.
[30] STFC (Science and Technology Facilities Council), *The HSL Mathematical Software Library. A Collection of Fortran Codes for Large Scale Scientific Computation*, http://www.hsl.rl.ac.uk/.
[31] W. F. Tinney and J. W. Walker, *Direct solutions of sparse network equations by optimally ordered triangular factorization*, Proc. IEEE, 55 (1967), pp. 1801–1809.
[32] University of Waterloo, *Concorde TSP Solver*, http://www.math.uwaterloo.ca/tsp/concorde.html.