# A Mapping and Scheduling Algorithm for Parallel Sparse Fan-In Numerical Factorization ⋆

Pascal Hénon, Pierre Ramet, and Jean Roman

LaBRI, UMR CNRS 5800, Université Bordeaux 1 & ENSERB
351, cours de la Libération, F-33405 Talence, France
{henon|ramet|roman}@labri.u-bordeaux.fr

**Abstract.** We present and analyze a general algorithm which computes efficient static schedulings of block computations for parallel sparse linear factorization. Our solver, based on a supernodal fan-in approach, is fully driven by this scheduling. We give an overview of the algorithms and present performance results on a 16-node IBM-SP2 with 66 MHz Power2 thin nodes for a collection of grid and irregular problems.

## 1 Introduction

Solving large sparse symmetric positive definite systems $Ax = b$ of linear equations is a crucial and time-consuming step, arising in many scientific and engineering applications. Consequently, many parallel formulations for sparse matrix factorization have been studied and implemented; one can refer to [6] for a complete survey on high performance sparse factorization.

In this paper, we focus on the block partitioning and scheduling problem for sparse $LDL^t$ factorization without pivoting on parallel MIMD architectures with distributed memory; we use $LDL^T$ factorization in order to solve sparse systems with complex coefficients. More precisely, we present and analyze a general algorithm which computes an efficient static scheduling [7, 10] of the block computations for a parallel solver based on a supernodal fan-in approach [3, 11, 12, 13] such that the parallel solver is fully driven by this scheduling. This can be done by very precisely taking into account the computation costs of BLAS 3 primitives, the communication cost and the cost of local aggregations due to the fan-in strategy. Using this block partitioning and scheduling algorithm, combined with an ordering of the unknowns based on a tight coupling of Nested Dissection and Approximate Minimum Degree strategies [9], our parallel fan-in supernodal solver is almost as efficient on irregular problems as it is for regular 2D or 3D grid problems.

In order to achieve efficient parallel sparse factorization, three pre-processing phases are required:

• The *ordering* phase, which computes a symmetric permutation of the initial matrix $A$ such that factorization will exhibit as much concurrency as possible while incurring low fill-in. In this work, we use a tight coupling of the Nested Dissection and Approximate Minimum Degree algorithms [1, 8, 9]; the partition

⋆ This work is supported by the *Commissariat à l'Énergie Atomique* CEA/CESTA under contract No. 7V1555AC, and by the GDR ARP (iHPerf group) of the CNRS.

of the original graph into supernodes is achieved by merging the partition of separators computed by the Nested Dissection algorithm and the supernodes amalgamated for each subgraph ordered by Halo Approximate Minimum Degree.

• The *block symbolic factorization* phase, which determines the block data structure of the factored matrix $L$ associated with the partition resulting from the ordering phase. This structure consists of $N$ column blocks, each of them containing a dense symmetric diagonal block and a set of dense rectangular off-diagonal blocks. One can efficiently perform such a block symbolic factorization in quasi-linear space and time complexities [5]. From the block structure of L, we can deduce the weighted elimination quotient graph that describes all dependencies between blocks, as well as the supernodal elimination tree.

• The *block repartitioning and scheduling* phase, which refines the partition, by splitting large supernodes in order to exploit concurrency within dense block computations, and maps it onto the processors of the target architecture.

The rest of the paper is organized as follows. Section 2 presents our algorithmic framework for parallel sparse symmetric factorization, describes our block repartitioning and scheduling algorithm, and outlines the supernodal fan-in solver driven by this precomputed scheduling. Section 3 provides numerical experiments on an IBM SP2 for a large class of sparse matrices, including performance results and analysis. Then, we conclude with remarks on the benefits of this study and on our future work.

## 2 Block mapping and parallel solver

Let us consider the block data structure of the factored matrix $L$ computed by the block symbolic factorization. Let us remind that a column block holds one dense diagonal block and some dense off-diagonal blocks. From this block data structure, we can introduce the boolean function $\mathit{off\_diag}(k,j)$, $1 \leq k < j \leq N$, that returns *true* if and only if there exists an off-diagonal block in column block $k$ facing column block $j$. Then, we can define the two sets $BStruct(L_{k*}) := \{i < k \mid \mathit{off\_diag}(i,k) = \mathit{true}\}$ and $BStruct(L_{*k}) := \{j > k \mid \mathit{off\_diag}(k,j) = \mathit{true}\}$. Thus, $BStruct(L_{k*})$ is the set of column blocks that update column block $k$, and $BStruct(L_{*k})$ is the set of column blocks updated by column block $k$.

We consider now a parallel supernodal version of sparse $LDL^t$ factorization with total local aggregation: all non-local column block contributions are aggregated locally in block structures. This scheme is close to the Fan-In algorithm [4] as processors communicate using only aggregate update column blocks. It is important to note that column block $k$ will receive an aggregated update column block only from every processor in set $Procs(L_{k*}) := \{map(i) \mid i \in BStruct(L_{k*})\}$, where the $map()$ operator is the block mapping function. These structures, denoted in the following $AUCB_k$, can be built from the block symbolic factorization. The pseudo-code of $LDL^t$ factorization can be expressed in terms of dense block computations (see figure 1); these computations are performed, as much as possible, on compacted sets of blocks, for BLAS efficiency. Synchronizations are enforced thanks to $r_k$, which counts the number of contributions remaining for column block $k$. While $r_k \neq 0$, we need to receive contributions for column block $k$ (lines 2–4). Then the diagonal block is factored and the off-diagonal blocks are updated (lines 6–8). Partial results (block $F$) are stored, to be re-used in the contribution computations. All of the contributions associated with column block $k$ can now be computed; a contribution can be added locally (line 12), or in an aggregate update column block (line 14). An aggregate update column block is sent when all of its local contributions have been added (line 15). To limit memory consumption due to the storage of aggregates, these structures are allocated only when needed and are deallocated just after being sent. If memory is a critical issue, an aggregate update column block can be sent with partial aggregation to free memory space; this is close to the Fan-Both scheme [2].

**Notations for a given $k$, $1 \leq k \leq N$ :**

- $r_k$ : number of contributions yet to subtract to column block $k$,
- $r_k^o$ : initial value of $r_k$,
- $s_k$ : number of contributions yet to add to $AUCB_k$,
- $s_k^o$ : initial value of $s_k$,
- symbol $*$ means $\forall j \in BStruct(L_{*k})$,
- let $j \geq k$; sequence $[j]$ means $\forall i \in BStruct(L_{*k}) \cup \{k\}$ with $i \geq j$ .

On processor p :
1. **For** $k \in [1, N]$ such that $map(k) == p$ **Do**
2.     **While** $r_k \neq 0$ **Do** % receive_update step
3.         receive $AUCB_k$;
4.         $A_{[k]k} = A_{[k]k} - AUCB_k; \quad r_k \leftarrow r_k - s_k^o;$
5.     % compute_send step
6.     Factor $A_{kk}$ into $L_{kk} D_k L_{kk}^t$;
7.     Solve $L_{kk} F^t = A_{*k}^t$;
8.     Solve $D_k L_{*k}^t = F^t$;
9.     **For** $j \in BStruct(L_{*k})$ **Do**
10.         compute $C = L_{[j]k} F_j^t$;
11.         **If** $map(j) == p$ **Then**
12.             $A_{[j]j} = A_{[j]j} - C; \quad r_j \leftarrow r_j - 1;$
13.         **Else** % aggregate step
14.             $AUCB_j = AUCB_j + C; \quad s_j \leftarrow s_j - 1;$
15.             **If** $s_j = 0$ **Then** send $AUCB_j$ to $map(j)$;

**Fig. 1.** Outline of the parallel factorization algorithm.

Before running the general parallel algorithm we presented above, we must perform a step consisting in the partitioning and mapping of the blocks of the symbolic matrix onto the set of processors. The partitioning and mapping phase aims at computing a static regulation that balances workload and enforces precedence constraints imposed by the factorization algorithm; the block elimination tree structure must be used there. Different levels of parallelism can be exhibited. The first level is induced by the sparsity of the matrix and corresponds to independent branches in the block elimination tree. The second level is induced by dense computation on large full blocks. The third level exploits instruction-level parallelism at the processor level by using BLAS subroutines. Algorithms like "subtree to subcube" or "proportional mapping" propose a partitioning and mapping strategy that balances workload based on a recursive traversal of the block elimination tree. Then, given the new partition and the mapping of blocks, the solver is responsible for scheduling block computations and communications.

This existing approach of a block partitioning and mapping strategy rises some problems. Those problems can be divided into two categories: on one hand problems due to the measure of workload and on the other hand those due to run-time scheduling of block computations in the solver. The measure of workloads is very rough because only numbers of operations are taken into account. Indeed, to be efficient, the solver algorithm is block-oriented, in order to use BLAS subroutines, whose efficiencies are very far from being linear in terms of numbers of operations. Moreover, the workload deals with block computations but does not take into account all the others phenomena in the parallel fan-in factorization: extra-workload generated by the fan-in approach and idle-waits due to the latency generated by message passing. The second kind of problem is run-time scheduling of concurrent block computations and communication in the solver is limited to rough criteria.

We tackle these problems by using an inspector-executor-like approach. In our scheme, the partitioning and mapping computations stand for the inspector phase, and the parallel solving computations for the executor phase. Thus, the partitioning and mapping step generates a fully ordered schedule used in parallel solving computations. This schedule aims at statically regulating all of the issues that are classically managed at run-time. These issues are:

- the order in which column blocks that have a contribution counter $r_k$ equal to zero are processed can influence the efficiency of computations (line 1);
- the order in which to send aggregate update column blocks (line 3) and to process the received ones (line 15) determines what block will be ready next for local computation;
- the way to switch between `receive_update` and `compute_send` steps (line 2).

To make our scheme very reliable, we estimate the workload and message passing latency by using a BLAS and communication network time model. To be efficient, this model has to be calibrated on the target architecture. The proposed scheduling and mapping algorithm can yield 1D or 2D block distributions. At the time being, we only handle 1D distributions, and thus in all the following we will restrict our discussion to the 1D case. However, a 2D extension of our work is in progress, since it is well known that, for large numbers of processors, 2D distributions achieve better scalability.

Unlike usual algorithms, our partitioning and distributing strategy is divided in two distinct phases. The partition phase splits column blocks associated with large supernodes, and builds for each column block a set of candidate processors for its mapping. The scheduling phase optimally maps each column block onto one of these processors. The partitioning algorithm is based on a recursive top-down strategy over the block elimination tree. Pothen and Sun presented such a strategy in [10]. It starts by splitting the root and assigning it to the set of candidate processors $Q$. Then, recursively, each subtree is assigned to a subset of $Q$ proportionally to its workload. It is important to notice that since a candidate processor is only a suggestion for the mapping and scheduling phase, we avoid any problem of rounding to integral numbers of processors by allowing a candidate processor to be in two sets of candidate processors for two subtrees having the same father in the block elimination tree.

Once the partitioning phase has built a new partition and the set of candidate processors for each column block, the task of electing an owner processor for each column block falls to the mapping phase. The idea that leads the mapping and scheduling strategy is to simulate parallel factorization as mapping comes along. Thus, we define for each processor a timer that will stand for the current time in computation of the processor and a ready block heap that contains at a time all blocks not yet mapped that have received their contribution and for which the processor is a candidate. Then we start mapping the leaves of the elimination tree (one candidate processor). When a block is mapped on a processor, three actions are processed:

- update the processor timer for local computation thanks to our BLAS model;
- compute the time a contribution from this block to another is computed;
- put a block in the ready block heaps of its candidate processors if all its contributions are computed.

After a block has been mapped we choose the next one to be mapped: we get the processor with lower timer and with a not empty ready block heap. Then we get the block in its ready block heap that is the sooner available. This one is the next block we are to map. Thus we are able to compute for each candidate processor the time it will have finished to treat this block if it is mapped on. This is possible thanks to:

- the processor timer;
- the date at which all contributions to this block have been computed;
- the communication cost modelization that gives the time needed to exchange the contributions.

The block is mapped on the candidate processor that will be able to compute it the sooner. Then we can loop on another block until the last.

The block computations are ordered in respect to the rank at which the column block have been mapped. Thus, for each processor $p$, we obtain a vector $K_p$ of the $N_p$ local column block numbers fully ordered by priority. Ordering column block computation is not sufficient to assure that the solver will respect all the predictions made in the mapping phase: emission and reception of aggregate update column blocks have to be ordered too.

The factorization algorithm is modified (see figure 2) to account for these new extensions. The emissions are performed on the set $AUCB_{ready}$ containing the ready aggregate update column blocks to send to $map(j)$ and whose priority is higher than the one of the first non ready aggregate update column block.

On processor p :

1. **For** $i = 1$ **To** $N_p$ **Do**
    $k = K_p(i)$
    [...]
3-4. subtract the $AUCB_k$ received
    [...]
15. **If** $s_j = 0$ **Then** send $AUCB_{ready}$ to $map(j)$;

**Fig. 2.** Outline of the new parallel factorization algorithm.

## 3 Numerical Experiments

In this section, we describe experiments performed on a collection of sparse matrices; the values of the metrics in table 1 come from scalar column symbolic factorization.

These parallel experiments were run on an IBM SP2 with 66 MHz Power2 thin nodes (260 MFlops peak performance) having 128 MBytes of physical memory and 512 MBytes of virtual memory each, and switch interruptions are enabled with default delay to perform non-blocking communications efficiently.

Table 2 reports the efficiency of our parallel block factorization for our distribution strategy. Profiled executions confirm that BLAS level 3 block computations ($GEMM$ and $TRSM$) represent more than 75% of total BLAS computations. A multi-variable polynomial regression has been used to build an analytical model of these routines. This model and the experimental values obtained for communication startup and bandwidth are used by the partitioning and scheduling algorithm. It is important to note that the number of operations actually performed during factorization is greater than the OPC value because of amalgamation and block computations. A good scalability is achieved for all of the test problems, at least for moderately sized parallel machines, and even with 1D block mapping.

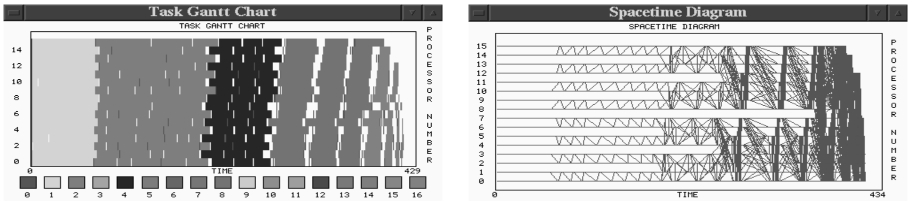


**Fig. 3.** Measured Gantt chart & Spacetime graph for CUBE39.

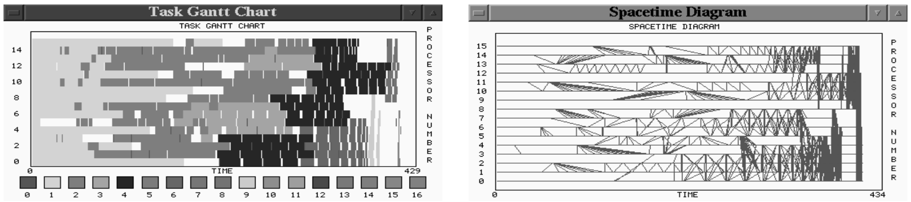


**Fig. 4.** Measured Gantt chart & Spacetime graph for CRANKSEG1.

**Table 1.** Description of our test problems. $NNZ_A$ is the number of off-diagonal terms in the triangular part of matrix $A$, $NNZ_L$ is the number of off-diagonal terms in the factored matrix $L$ and $OPC$ is the number of operations required.

| Name | Columns | $NNZ_A$ | $NNZ_L$ | OPC | Description |
|---|---|---|---|---|---|
| GRID511 | 261121 | 1041420 | 1.202166e+07 | 2.565341e+09 | Regular 2D mesh |
| CUBE39 | 59319 | 730778 | 2.210534e+07 | 2.240674e+10 | Regular 3D cube |
| BCSSTK30 | 28924 | 1007284 | 4.309003e+06 | 1.188915e+09 | Boeing-Harwell |
| BCSSTK32 | 44609 | 985046 | 5.239146e+06 | 1.162900e+09 | Boeing-Harwell |
| COL15 | 10428 | 701292 | 3.371109e+06 | 1.495460e+09 | 3D Cologne Chalenge |
| COL30 | 20373 | 1394292 | 7.849464e+06 | 4.359803e+09 | 3D Cologne Chalenge |
| COL75 | 50208 | 3473292 | 2.248613e+07 | 1.532035e+10 | 3D Cologne Chalenge |
| CRANKSEG1 | 52804 | 5280703 | 3.142730e+07 | 3.007141e+10 | PARASOL |
| OILPAN | 73752 | 1761718 | 8.912337e+06 | 2.984944e+09 | Boeing-Harwell |
| TOOTH | 78136 | 452591 | 1.031143e+07 | 6.267094e+09 | Boeing-Harwell |
| B5TUER | 162610 | 3873534 | 2.541937e+07 | 1.530774e+10 | PARASOL |
| BBMAT | 38744 | 1274141 | 1.716094e+07 | 1.250040e+10 | PARASOL |
| INVEXT | 30412 | 906915 | 7.256566e+06 | 3.766788e+09 | PARASOL |
| MIXTANK | 29957 | 982542 | 9.280247e+06 | 7.316933e+09 | PARASOL |
| MT1 | 97578 | 4827996 | 3.114873e+07 | 2.109265e+10 | PARASOL |
| QUER | 59122 | 1403689 | 9.118592e+06 | 3.280680e+09 | PARASOL |
| SHIP001 | 34920 | 2304655 | 1.427916e+07 | 9.033767e+09 | PARASOL |
| X104 | 108384 | 5029620 | 2.634047e+07 | 1.712902e+10 | PARASOL |

**Table 2.** Factorization performance results (time in seconds and **Gigaflops**) on the IBM SP2.

| Name | Number of processors | | | |
|---|---|---|---|---|
| | 2 | 4 | 8 | 16 |
| *GRID*511 | 14.45 (**0.18**) | 7.45 (**0.34**) | 4.24 (**0.61**) | 2.70 (**0.95**) |
| *CUBE*39 | - | 42.86 (**0.52**) | 24.40 (**0.92**) | 15.55 (**1.44**) |
| *BCSSTK*30 | 6.43 (**0.18**) | 3.60 (**0.33**) | 2.27 (**0.52**) | 1.83 (**0.65**) |
| *BCSSTK*32 | 7.28 (**0.16**) | 3.60 (**0.33**) | 2.13 (**0.55**) | 1.52 (**0.77**) |
| *COL*15 | 6.93 (**0.22**) | 3.78 (**0.40**) | 2.37 (**0.63**) | 1.84 (**0.81**) |
| *COL*30 | 18.13 (**0.24**) | 9.81 (**0.44**) | 5.68 (**0.77**) | 3.72 (**1.17**) |
| *COL*75 | 157.54 (**0.10**) | 31.24 (**0.49**) | 17.14 (**0.89**) | 9.99 (**1.53**) |
| *CRANKSEG*1 | - | 58.14 (**0.52**) | 32.11 (**0.94**) | 19.29 (**1.56**) |
| *OILPAN* | 14.03 (**0.21**) | 7.61 (**0.39**) | 4.41 (**0.68**) | 3.12 (**0.96**) |
| *TOOTH* | 32.97 (**0.19**) | 18.05 (**0.35**) | 10.17 (**0.62**) | 6.49 (**0.97**) |
| *B*5*TUER* | - | 31.79 (**0.48**) | 17.58 (**0.87**) | 10.77 (**1.42**) |
| *BBMAT* | 61.54 (**0.20**) | 30.21 (**0.41**) | 17.07 (**0.73**) | 11.17 (**1.12**) |
| *INVEXT* | 20.37 (**0.18**) | 10.99 (**0.34**) | 5.88 (**0.64**) | 4.10 (**0.92**) |
| *MIXTANK* | 30.31 (**0.24**) | 18.05 (**0.41**) | 10.03 (**0.73**) | 6.75 (**1.08**) |
| *MT*1 | - | 43.77 (**0.48**) | 24.72 (**0.85**) | 15.07 (**1.40**) |
| *QUER* | 14.92 (**0.22**) | 8.05 (**0.41**) | 4.69 (**0.70**) | 3.32 (**0.99**) |
| *SHIP*001 | 38.46 (**0.23**) | 19.88 (**0.45**) | 10.75 (**0.84**) | 7.45 (**1.21**) |
| *X*104 | - | 38.21 (**0.45**) | 21.24 (**0.81**) | 13.97 (**1.23**) |

On both moderate and large size grid and irregular problems, the measured performance vary between 0.65 and 1.56 Gigaflops on 16 nodes, using double precision arithmetic, thereby obtaining about 35% of the processor peak theoretical performance. Our scheduling leads to efficient parallel execution, with very good communication locality, and minimizes idle-time (see figures 3, 4).

## 4 Conclusion and Perspectives

In this paper, we have presented an efficient static scheduling of the block computations of a parallel sparse direct solver. This work is still in progress; we are currently developing a 2D mapping strategy for the uppermost column blocks of the block elimination tree, and studying an adaptive computation of optimal packet sizes to improve computation/communication overlap in asynchronous communications.

## References


[1] P. Amestoy, T. Davis, and I. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. and Appl.*, 17:886–905, 1996.

[2] C. Ashcraft. The fan-both family of column-based distributed Cholesky factorization algorithms. *Graph Theory and Sparse Matrix Computation, IMA, Springer-Verlag*, 56:159–190, 1993.

[3] C. Ashcraft, S. C. Eisenstat, and J. W.-H. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SIAM J. Sci. Stat. Comput.*, 11(3):593–599, 1990.

[4] C. Ashcraft, S. C. Eisenstat, J. W.-H. Liu, and A. Sherman. A comparison of three column based distributed sparse factorization schemes. In *Proc. Fifth SIAM Conf. on Parallel Processing for Scientific Computing*, 1991.

[5] P. Charrier and J. Roman. Algorithmique et calculs de complexité pour un solveur de type dissections emboîtées. *Numerische Mathematik*, 55:463–476, 1989.

[6] I. S. Duff. Sparse numerical linear algebra: direct methods and preconditioning. Technical Report TR/PA/96/22, CERFACS, 1996.

[7] G. A. Geist and E. Ng. Task scheduling for parallel sparse Cholesky factorization. *Internat. J. Parallel Programming*, 18(4):291–314, 1989.

[8] F. Pellegrini and J. Roman. Sparse matrix ordering with SCOTCH. In *Proceedings of HPCN'97, Vienna, LNCS 1225*, pages 370–378, April 1997.

[9] F. Pellegrini, J. Roman, and P. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. In *Proceedings of IRREGULAR'99, Puerto Rico, LNCS 1586*, pages 986–995, April 1999.

[10] A. Pothen and C. Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM J. Sci. Comput.*, 14(5):1253–1257, September 1993.

[11] E. Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. *SIAM J. Sci. Comput.*, 17(3):699–713, May 1996.

[12] E. Rothberg and A. Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. *SIAM J. Sci. Comput.*, 15(6):1413–1439, November 1994.

[13] E. Rothberg and R. Schreiber. Improved load distribution in parallel sparse Cholesky factorization. In *Proceedings of Supercomputing'94*, pages 783–792. IEEE, 1994.