PaStiX : A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions *

Pascal Hénon, Pierre Ramet, and Jean Roman

LaBRI, UMR CNRS 5800, Université Bordeaux I & 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 an efficient static scheduling of block computations for a parallel $L.D.L^t$ factorization of sparse symmetric positive definite systems based on a combination of 1D and 2D block distributions. Our solver uses a supernodal fan-in approach and is fully driven by this scheduling. We give an overview of the algorithm and present performance results and comparisons with PSPASES on an IBM-SP2 with 120 MHz Power2SC nodes for a collection of 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. In order to achieve efficient parallel sparse factorization, three pre-processing phases are commonly 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, 10]; the partition 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

* 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.

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, one can deduce the weighted elimination quotient graph that describes all dependencies between column blocks, as well as the supernodal elimination tree.

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

The authors presented in [8] a preliminary version describing a mapping and scheduling algorithm for 1D distribution of column blocks. This paper extends this previous work by presenting and analysing a mapping and scheduling algorithm based on a combination of 1D and 2D block distributions. This algorithm computes an efficient static scheduling [7, 11] of the block computations for a parallel solver based on a supernodal fan-in approach [3, 12, 13, 14] 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.

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. According to these results, our PaStiX software appears to be a good competitor with the current reference software PSPASES [9]. Then, we conclude with remarks on the benefits of this study and on our future work.

2 Parallel solver and block mapping

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.

Let us now consider a parallel supernodal version of sparse LDL^t factorization with total local aggregation: all non-local block contributions are aggregated locally in block structures. This scheme is close to the Fan-In algorithm [4] as processors communicate using only aggregated update blocks. If memory is a critical issue, an aggregated update block can be sent with partial aggregation to free memory space; this is close to the Fan-Both scheme [2]. So, a block j in column block k will receive an aggregated update block only from every processor in set $Procs(L_{jk}) := \{map(j,i) \mid i \in BStruct(L_{k*})\ and\ j \in BStruct(L_{*i})\}$, where the $map(,)$ operator is the 2D block mapping function. These aggregated update blocks, denoted in the following AUB_{jk}, are stored by processors in $Procs(L_{jk})$, and 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.

The proposed algorithm can yield 1D or 2D block distributions. The block computations as well as the emissions and receptions of aggregated update block are fully ordered with a compatible priority. Block computations, or tasks, can be classified in four types :

- `COMP1D(k)` : update and compute contributions for column block k;
- `FACTOR(k)` : factorize the diagonal block k;
- `BDIV(j,k)` : update the off-diagonal block j on column block k ($k < j \leq N$);
- `BMOD(i,j,k)` : compute the contribution for block i in column block k facing column block j ($k < j \leq i \leq N$).

For a given k, $1 \leq k \leq N$, we define :
- N_p : total number of local tasks for processor p;
- $K_p[n]$: n^{th} local task for processor p;
- for the column block k, symbol $\star$ 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 :

For $n = 1$ to N_p **Do**
  Switch (Type($K_p[n]$)) **Do**
    `COMP1D(k)`: Receive all $AUB_{[k]k}$ for $A_{[k]k}$
      Factor A_{kk} into $L_{kk}D_kL_{kk}^t$
      Solve $L_{kk}F_\star^t = A_{\star k}^t$ and $D_kL_{\star k}^t = F_\star^t$
      For $j \in BStruct(L_{*k})$ **Do**
        Compute $C_{[j]} = L_{[j]k}F_j^t$
        If $map([j], j) == p$ **Then** $A_{[j]j} = A_{[j]j} - C_{[j]}$
        Else $AUB_{[j]j} = AUB_{[j]j} + C_{[j]}$, if ready send $AUB_{[j]j}$ to $map([j], j)$
    `FACTOR(k)`: Receive all AUB_{kk} for A_{kk}
      Factor A_{kk} into $L_{kk}D_kL_{kk}^t$ and send $L_{kk}D_k$ to $map([k], k)$
    `BDIV(j,k)`: Receive $L_{kk}D_k$ and all AUB_{jk} for A_{jk}
      Solve $L_{kk}F_j^t = A_{jk}^t$ and $D_kL_{jk}^t = F_j^t$ and send F_j^t to $map([j], k)$
    `BMOD(i,j,k)`: Receive F_j^t
      Compute $C_i = L_{ik}F_j^t$
      If $map(i, j) == p$ **Then** $A_{ij} = A_{ij} - C_i$
      Else $AUB_{ij} = AUB_{ij} + C_i$, if ready send AUB_{ij} to $map(i, j)$

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 the precedence constraints imposed by the factorization algorithm; the block elimination tree structure must be used there.

The existing approaches for block partitioning and mapping rise several problems, which can be divided into two categories: on one hand, problems due to the measure of workload, and on the other hand those due to the run-time scheduling of block computations in the solver. To be efficient, solver algorithms are block-oriented to take advantage of BLAS subroutines, which efficiencies are far from being linear in terms of number of operations. Moreover, workload encompasses block computations but does not take into account the other phenomena that occur in parallel fan-in factorization, such as extra-workload generated by the fan-in approach and idle-waits due to the latency generated by message passing.

The obtaining of high performances at run-time requires to compute efficient solutions to several scheduling problems that define the orders in which:
- to process tasks that are locally ready what is crucial for minimizing idle time;
- to send and to process the reception of aggregate update blocks (due to fan-in) and blocks used by `BDIV` or `BMOD` tasks, which determines what block will be ready next for local computation.
To tackle these problems, 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.

To make our scheme very reliable, we estimate the workload and message passing latency by using a BLAS and communication network time model, which is automatically calibrated on the target architecture. Unlike usual algorithms, our partitioning and mapping strategy is divided in two distinct phases.

The partitioning phase is based on a recursive top-down strategy over the block elimination tree. Pothen and Sun presented such a strategy in [11]. For each supernode, starting by the root, we assign it to a set of candidate processors Q for its mapping. Given the number of such candidate processors and the cost (that takes into account BLAS effect) of the supernode, we choose a 1D or 2D distribution strategy; the mapping and scheduling phase will use this choice to distribute this supernode. Then, recursively, each subtree is assigned to a subset of Q proportionally to its workload. Consequently, this strategy leads to a 2D distribution for the uppermost supernodes in the block elimination tree, and to a 1D for the others. Moreover, we allow a candidate processor to be in two sets of candidate processors for two subtrees having the same father in the elimination tree. The mapping phase will choose the better proportion to use such a processor in the two sets. By this way we avoid any problem of rounding to integral numbers. Furthermore, the column blocks corresponding to large supernodes are split using the blocking size suitable to achieve BLAS efficiency.

Once the partitioning phase is over, the task graph is built, such that each task is associated with the same set of candidate processors as the one of the supernode from which the task depends. The scheduling phase maps each task onto one of these processors; it uses a greedy algorithm that consists in mapping each task as it comes during the simulation of the parallel factorization. Thus, for each processor, we define a timer that will hold the current elapsed computation time, and a ready task heap that will contain at a time all tasks that are not yet mapped, that have received all of their contributions, and for which the processor is candidate. The algorithm then starts by mapping the leaves of the elimination tree (which have only one candidate processor). When a task is mapped onto a processor, the mapper: updates the timer of this processor according to our BLAS model; computes the time at which a contribution from this task is computed; puts into the heaps of their candidate processors all tasks all of the contributions of which have been computed.

After a task has been mapped, the next task to be mapped is selected by taking the first task of each ready tasks heap, and by choosing the one that comes from the lowest node in the elimination tree. Then, we compute for each of its candidate processors the time at which it will have completed the task if it is mapped onto it, thanks to: the processor timer; the time at which all contributions to this task have been computed (taking into account the fan-in overcost); the communication cost modeling that gives the time to send the contributions. The task is mapped onto the candidate processor that will be able to compute it the soonest.

After this phase has ended, the computations of each task are ordered with respect to the rank at which the task have been mapped. Thus, for each processor p, we obtain a vector K_p of the N_p local task numbers fully ordered by priority, and the solver described at figure 1 will be fully driven by this scheduling order.

3 Numerical Experiments

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

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

Name	Columns	$\mathrm{NNZ_A}$	$\mathrm{NNZ_{L(Scotch)}}$	OPC(Scotch)	$\mathrm{NNZ_{L(MeTiS)}}$	OPC(MeTiS)
B5TUER	162610	3873534	2.542e+07	1.531e+10	2.404e+07	1.237e+10
BMWCRA1	148770	5247616	6.597e+07	5.702e+10	6.981e+07	6.124e+10
MT1	97578	4827996	3.115e+07	2.109e+10	3.455e+07	2.269e+10
OILPAN	73752	1761718	8.912e+06	2.985e+09	9.065e+06	2.751e+09
QUER	59122	1403689	9.119e+06	3.281e+09	9.586e+06	3.448e+09
SHIP001	34920	2304655	1.428e+07	9.034e+09	1.481e+07	9.462e+09
SHIP003	121728	3982153	5.873e+07	8.008e+10	5.910e+07	7.587e+10
SHIPSEC5	179860	4966618	5.650e+07	6.952e+10	5.256e+07	5.509e+10
THREAD	29736	2220156	2.404e+07	3.884e+10	2.430e+07	3.583e+10
X104	108384	5029620	2.634e+07	1.713e+10	2.728e+07	1.412e+10

We compare factorization times performed in double precision real by our PaStiX software and by PSPASES version 1.0.3 based on a multifrontal approach [9]. PaStiX makes use of version 3.4 of the Scotch static mapping and sparse ordering software package developed at LaBRI. PSPASES is set to make use of MeTiS version 4.0 as its default ordering library. In both cases, blocking size is set to 64 and the IBM ESSL library is used.

These parallel experiments were run on an IBM SP2 at CINES (Montpellier, France), which nodes are 120 MHz Power2SC thin nodes (480 MFlops peak performance) having 256 MBytes of physical memory each. Table 2 reports the performances of our parallel block factorization for our distribution strategy and the one of the PSPASES software.

Table 2. Factorization performance results (time in seconds and Gigaflops) on the IBM SP2. For each matrix, the first line gives the PaStiX results and the second line the PSPASES ones.

Name	Number of processors					
	2	4	8	16	32	64
B5TUER	**29.54** (0.52)	**15.52** (0.99)	**8.86** (1.73)	**5.25** (2.91)	**3.96** (3.87)	**2.91** (5.25)
	-	-	14.04 (0.88)	7.32 (1.69)	3.91 (3.16)	2.58 (4.79)
BMWCRA1	-	-	**30.97** (1.84)	**17.28** (3.30)	**9.89** (5.76)	**6.94** (8.21)
	-	-	-	-	24.87 (2.49)	13.48 (4.61)
MT1	**37.92** (0.56)	**20.35** (1.04)	**11.29** (1.87)	**6.65** (3.17)	**4.33** (4.87)	**3.51** (6.01)
	-	-	-	10.71 (2.12)	5.70 (3.98)	3.59 (6.32)
OILPAN	**7.28** (0.41)	**3.81** (0.78)	**2.15** (1.39)	**1.39** (2.14)	**1.00** (3.00)	**0.87** (3.42)
	-	5.23 (0.53)	2.79 (0.99)	1.73 (1.59)	1.25 (2.20)	0.93 (2.96)
QUER	**8.35** (0.39)	**4.46** (0.74)	**2.57** (1.28)	**1.67** (1.96)	**1.16** (2.83)	**0.93** (3.53)
	23.80 (0.14)	13.11 (0.26)	3.22 (1.07)	2.01 (1.72)	1.30 (2.65)	0.96 (3.59)
SHIP001	**20.98** (0.43)	**10.91** (0.83)	**6.07** (1.49)	**3.63** (2.49)	**2.43** (3.72)	**1.96** (4.60)
	-	15.32 (0.62)	8.11 (1.17)	4.48 (2.11)	2.98 (3.17)	2.14 (4.42)
SHIP003	-	**109.78** (0.73)	**45.83** (1.75)	**25.75** (3.11)	**16.60** (4.83)	**12.03** (6.66)
	-	-	-	-	21.28 (3.57)	14.08 (5.39)
SHIPSEC5	-	**79.12** (0.88)	**35.68** (1.95)	**20.51** (3.39)	**13.99** (4.97)	**11.58** (6.00)
	-	-	-	-	21.80 (2.52)	11.81 (4.66)
THREAD	**78.04** (0.50)	**41.14** (0.94)	**22.85** (1.70)	**13.50** (2.88)	**10.42** (3.73)	**6.70** (5.80)
	-	-	-	21.02 (1.70)	11.24 (3.19)	6.41 (5.59)
X104	**31.53** (0.54)	**19.69** (0.87)	**9.98** (1.72)	**7.49** (2.29)	**5.09** (3.37)	**3.85** (4.44)
	-	-	-	8.32 (1.70)	4.92 (2.87)	3.18 (4.44)

An important remark to help comparisons is that PSPASES uses a Cholesky (LL^T) factorization, intrinsicaly more BLAS efficient and cache-friendly than the LDL^T one used by PaStiX. For instance, for a dense 1024x1024 matrix on one Power2SC node, the ESSL LL^T factorization time is 1.07s whereas the ESSL LDL^T factorization time is 1.27s.

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.

Good scalability is achieved for all of the test problems, at least for moderately sized parallel machines; the measured performances vary between 3.42 and 8.21 Gigaflops on 64 nodes. Results show that PaStiX compares very favorably to PSPASES and achieves better solving times in almost all cases up to 32 processors. The comparison on larger cases still appears favorable to PaStiX almost up to 64 processors. On the other smaller cases, performance is quite equivalent on 64 processors when scalability limit is reached.

4 Conclusion and Perspectives

In this paper, we have presented an efficient combination of 1D and 2D distributions and the induced static scheduling of the block computations for a parallel sparse direct solver. This work is still in progress; we are currently working on a more efficient criterion to switch between the 1D and 2D distributions, to improve scalability. We are also developing a modified version of our strategy to take into account architectures based on SMP nodes.

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] P. Hénon, P. Ramet, and J. Roman. A Mapping and Scheduling Algorithm for Parallel Sparse Fan-In Numerical Factorization. In *EuroPAR'99, LNCS 1685*, pages 1059–1067. Springer Verlag, 1999.

[9] M. Joshi, G. Karypis, V. Kumar, A. Gupta, and Gustavson F. PSPASES : Scalable Parallel Direct Solver Library for Sparse Symmetric Positive Definite Linear Systems. Technical report, University of Minnesota and IBM Thomas J. Watson Research Center, May 1999.

[10] 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. Extended version to be published in Concurrency: Practice and Experience.

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

[12] 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.

[13] 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.

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