Efficient algorithms for direct resolution of large sparse systems on clusters of SMP nodes

P. Hénon, P. Ramet and J. Roman
LaBRI, UMR CNRS 5800,
Université Bordeaux 1 & ENSEIRB,
INRIA Futurs ScAlApplix Project,
351, cours de la Libération,
F-33405 Talence, FRANCE

April 27, 2003

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 techniques for sparse matrix factorization have been studied and implemented; see [11] for a complete survey on high performance sparse factorization.

In this paper, we focus on the block partitioning and scheduling problem for high performance sparse LDL^T or LL^T factorization without pivoting on parallel architectures; in fact, our strategy is suitable for general distributed heterogeneous architectures whose computation and communication performances are predictable in advance.

In order to achieve efficient parallel sparse factorization, we perform three sequential pre-processing phases:

• The *ordering* phase, which computes a symmetric permutation of the initial matrix A such that factorization process 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, 27]. 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 factorized 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 [9]. 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 previous partition by splitting large supernodes in order to exploit concurrency within dense block computations, and which maps the resulting blocks onto the processors of the target architecture (see section 2.2).

There are two main approaches for numerical factorization algorithms: the *multifrontal* approach [2, 10, 12, 13, 15, 30], and the *supernodal* [16, 29, 31, 32] approach with *fan-in* or *fan-out* variations [5, 6, 7, 8]. Independently of these different methods, a static or dynamic scheduling of block computations can be used in a parallel framework. For homogeneous parallel architectures, it is useful to find an efficient static scheduling [14, 18, 28]. In this context, this scheduling can be induced by a fine cost computation/communication model.

The PSPASES solver [21] is based on a multifrontal approach without pivoting for symmetric positive definite systems. It uses METIS [22] for computing a fill-reducing ordering which is based on a multilevel nested dissection algorithm; when the graph is separated into p parts, a multiple mininum degree (MMD [25]) is then used. A "subtree to subcube" algorithm is applied to build a static mapping before the numerical factorization. In [4] the performances of MUMPS [3] and SuperLU [23, 24] are compared for nonsymmetric problems. MUMPS uses a multifrontal approach with dynamic pivoting for stability while SuperLU is based on a supernodal technique with static pivoting. The standard ordering used by MUMPS is the approximate minimum degree (AMD [1]) ordering, while SuperLU uses the

multiple minimum degree (MMD [25]) ordering. In both cases, a pivot order is defined by the symbolic factorization stage, but numerical considerations might prevent strict adherence to this order during numerical factorization. MUMPS can choose pivots off of the diagonal: the modulus of the prospective pivot is compared with the largest modulus of an entry in the column and the pivot is only accepted if this modulus is greater than a threshold value. In the SuperLU approach, a static pivoting strategy is used and kept rigorously to the pivot sequence chosen in the symbolic analysis.

In this paper, we focus on the block partitioning and scheduling problem for high performance sparse supernodal factorization without pivoting for symmetric positive definite systems. Thus, our algorithmic framework is close to the PSPASES one [21]. We presented in [18, 19, 20] a preliminary version of this work describing a mapping and scheduling algorithm based on a combination of 1D and 2D block distributions. This algorithm computes an efficient static schedule of the block computations for a parallel solver based on a supernodal approach with total agregation of contribution blocks, such that the parallel solver is fully driven by this scheduling. This can be done by very precisely taking into account the computational costs of the BLAS 3 primitives, the communication cost and the cost of local aggregations. Our study was suitable for homogeneous parallel/distributed architectures whose performances are predictable.

To solve 3D problems with more than 10 millions of unkowns, which is now a reachable challenge with new SMP supercomputers, we must keep a good scalability and control memory overhead due to agregation. In this context, we present two major improvements. The first one consists of taking into account heterogeneous architectures and more particulary those based on SMP nodes like the IBM SP3. Our communication model, used during the static scheduling and mapping step, is extended to manage both data exchanges by shared memory (less costly) and data exchanges by network (more costly). The second improvement is an adaptation of our static computation and communication scheduling algorithm to anticipate the sending of partially aggregated blocks in order to free memory dynamicaly. By doing this, we are able to divide the aggregated memory overhead while keeping good performance, and many experiments have shown that aggregated memory reduction is acceptable, up to 50%, in term of time penalty [17]. Another natural consequence of this improvement is that we can build accurately the memory access scheme for blocks. So we can prefetch the block access during the computation and communication scheduling with a compatible order. This allows an efficient implementation of an "out-of-core" version of our PaStiX solver.

The paper is organized as follows. In section 2 we introduce the algorithmic framework for our parallel sparse symmetric factorization before describing our block repartitioning and scheduling algorithm. Section 3 provides many numerical experiments on an IBM SP3 for a representative class of large sparse matrices from industrial problems, including performance results and analysis, for the LL^T Cholesky factorization. In section 4, we present our solutions to control memory overhead. Finally, we conclude with some prospects of our future work.

2 Description of algorithms

2.1 Parallel factorization algorithm

Let us consider the block data structure of the factorized matrix L computed by the block symbolic factorization. Recall that each of the N column-blocks holds one dense diagonal block and some dense off-diagonal blocks. Then we define the two sets: $BStruct(L_{k*})$ 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 (see [17, 18, 19, 20] for details).

Let us now consider a parallel supernodal version of sparse 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 [8] as processors communicate using only aggregated update blocks. These aggregated update blocks, denoted in what follows by AUB, can be built from the block symbolic factorization. These contributions are locally aggregated before being sent. The proposed algorithm can yield 1D (column-block) or 2D (block) distributions [17, 19]. Block computations can be classified in four types, and the associated tasks are defined as follows: $\forall k, 1 \leq k \leq N, \forall i, j \in BStruct(L_{*k}), i \geq j$,

- `COMP1D(k)` : *factorize the column-block k and compute all the contributions for the column-blocks in $BStruct(L_{*k})$*
- `FACTOR(k)` : *factorize the diagonal block k*
- `BDIV(j,k)` : *update the off-diagonal block j in column-block k*
- `BMOD(i,j,k)` : *compute the contribution of the block i in column-block k for block i in column-block j.*

2.2 Partitioning and mapping phase

Before running the general parallel algorithm reintroduce above, we must perform a step consisting of partitioning and mapping 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.

Our main algorithm is based on a static regulation led by a time simulation during the mapping phase. Thus, the partitioning and mapping step generates a fully ordered schedule used in the parallel factorization. This schedule aims at statically regulating all of the issues that are classically managed at runtime. 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 distribution strategy is divided in two distinct phases. The partition phase splits column-blocks associated with large supernodes, builds, for each column-block, a set of candidate processors for its mapping, and determines if it will be mapped using a 1D or 2D distribution. Once the partitioning step is over, the task graph is built. In this graph, each task is associated with the set of candidate processors of its column-block. The mapping and scheduling phase then optimally maps each task onto one of these processors.

The partitioning algorithm is based on a recursive top-down strategy over the block elimination tree provided by block symbolic factorization. Pothen and Sun

presented such a strategy in [28]. Our algorithm starts by splitting the root and assigning it to a set of candidate processors Q that is the set of all processors. Given the number of candidate processors and the size of the supernodes, it chooses the strategy (1D or 2D) that the mapping and scheduling phase will use to distribute this supernodes. Then each subtree is recursively assigned to a subset of Q proportionally to its workload.

Once the partitioning phase has built a new partition and the set of candidate processors for each task, the election of an owner processor for each task falls to the mapping and scheduling phase. The idea behind this phase is to simulate parallel factorization as each mapping comes along. Thus, for each processor, we define a timer that will hold the current elapsed computation time, and a ready task heap. At a given time, this task heap will contain all tasks that are not yet mapped, that have received all of their contributions, and for which the processor is a candidate. The algorithm starts by mapping the leaves of the elimination tree (those which have only one candidate processor). After a task has been mapped, the next task to be mapped is selected as follows: we take the first task of each ready task heap and choose the one that comes from the lowest node in the elimination tree. The communication pattern of all the contributions for a task depends on the already mapped tasks and on the candidate processor for the ownership of this task. The task is mapped onto the candidate processor that will be able to compute it the soonest.

As a conclusion about the partitioning and mapping phase, we can say that we obtain a strategy that allows us to take into account, in the mapping of task computations, all the phenomena that occur during the parallel factorization. Thus we achieve a block computation and communication scheme that drives the parallel solver efficiently.

2.3 Modeling for clusters of SMP nodes

Another important point is that our strategy can take into account heterogenous architecture. For example, in the case of an architecture based on SMP nodes, the time to send an AUB from a processor p_1 to a processor p_2 is estimated using the intra-node communication model if p_1 and p_2 belong to the same SMP node, or using the extra-node communication model if p_1 and p_2 belong to different SMP nodes.

However, it is important to take into account that the startup and the bandwidth of a communication depend on the number of simultaneous sends performed inside a same SMP node. Our algorithm does not allow us to know the number of simultaneous communications performed inside a same SMP node, so we introduce a communication model that define the startup and the bandwidth as a function of the number of candidate processors assigned to a supernode of the elimination tree. We consider that if m candidate processors are assigned to a supernode, there are $m/2$ simultaneous communications in average.

3 Run-time performances

All of the algorithms described in this paper have been integrated in the PaStiX software [18, 19], based on libraries that make use of version 3.4 of the Scotch static mapping and sparse matrix ordering software package [26], both developed

at LaBRI and in the ScAlApplix [1] INRIA project.

The parallel experiments were run on an 28 NH2 nodes (16 Power3+/375Mhz, 1.5Gflops, 16Go) located at CINES (Montpellier, France) with a network based on a Colony switch. All computations are performed in double precision and all time results are given in seconds. In all the following tables, the symbol "-" is used when the time measurments are not significant due to memory swapping.

Name	Columns	NNZ_A	NNZ_L	OPC	$\frac{NNZ_L}{OPC}$	Description
GRID1023	1046529	4179980	5.615708e+07	2.083481e+10	2.69e-3	2D Mesh
CUBE39	59319	730778	2.210534e+07	2.240674e+10	0.99e-3	3D Mesh
CUBE47	103823	1290898	4.828456e+07	6.963850e+10	0.69e-3	3D Mesh
BCSSTK32	44609	985046	5.239146e+06	1.162900e+09	4.50e-3	Ruth-Boeing
BBMAT	38744	1274141	1.716094e+07	1.250040e+10	1.37e-3	Ruth-Boeing
TOOTH	78136	452591	1.031143e+07	6.267094e+09	1.64e-3	3D Mesh
OCEAN	143437	409593	2.029997e+07	1.301477e+10	1.56e-3	3D Mesh
M14B	214765	1679018	6.236747e+07	6.112540e+10	1.02e-3	3D Mesh
OILPAN	73752	1761718	8.912337e+06	2.984944e+09	2.98e-3	PARASOL
QUER	59122	1403689	9.118592e+06	3.280680e+09	2.78e-3	PARASOL
INVEXTR1	30412	906915	7.256566e+06	3.766788e+09	1.93e-3	PARASOL
SMDOOR	162610	3873534	2.541937e+07	1.530774e+10	1.66e-3	PARASOL
SHIP001	34920	2304655	1.427916e+07	9.033767e+09	1.58e-3	PARASOL
X104	108384	5029620	2.634047e+07	1.712902e+10	1.54e-3	PARASOL
MT1	97578	4827996	3.114873e+07	2.109265e+10	1.48e-3	PARASOL
BMW3_2	227362	5530634	4.420244e+07	3.007981e+10	1.47e-3	PARASOL
MIXTANK	29957	982542	9.280247e+06	7.316933e+09	1.26e-3	PARASOL
BMWCRA_1	148770	5247616	6.597301e+07	5.701988e+10	1.16e-3	PARASOL
CRANKSG1	52804	5280703	3.142730e+07	3.007141e+10	1.05e-3	PARASOL
SHIPSEC8	114919	3269240	3.572761e+07	3.684269e+10	0.97e-3	PARASOL
CRANKSG2	63838	7042510	4.190437e+07	4.602878e+10	0.91e-3	PARASOL
SHIPSEC5	179860	4966618	5.649801e+07	6.952086e+10	0.81e-3	PARASOL
SHIP003	121728	3982153	5.872912e+07	8.008089e+10	0.73e-3	PARASOL
THREAD	29736	2220156	2.404333e+07	3.884020e+10	0.62e-3	PARASOL
COLOGB30	20373	1394292	7.849464e+06	4.359803e+09	1.80e-3	3D Cologne
COLOGB75	50208	3473292	2.248613e+07	1.532035e+10	1.47e-3	3D Cologne
COUP1500T	994983	6.93e+07	5.22e+08	3.80e+11	1.38e-3	3D Coupole
COUP2000T	1326483	9.24e+07	6.89e+08	5.01e+11	1.38e-3	3D Coupole
COUP3000T	1989483	1.38e+08	1.04e+09	7.60e+11	1.37e-3	3D Coupole
COUP5000T	3315483	2.31e+08	1.73e+09	1.27e+12	1.36e-3	3D Coupole
COUP8000T	5304483	3.69e+08	2.78e+09	2.03e+12	1.37e-3	3D Coupole
COUP40000T	26520483	1.85e+09	1.50e+10	10.8e+12	1.39e-3	3D Coupole

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 factorized matrix L and OPC is the number of operations required for the factorization. Matrices are sorted in decreasing order of* $\frac{NNZ_L}{OPC}$ *which is a measure of the potential data reuse [23].*

Our experiments are performed on a collection of sparse matrices from the Rutherford-Boeing Collection, from the PARASOL ESPRIT Project and from CEA (3D Cologne, Coupole). The almost part of theses matrices are structural mechanics and CFD matrices. The values of the associated measurements in Table 1 come from scalar column symbolic factorization.

We can see that we obtain a rather good scalability for all the test problems. In most large cases, the efficiency results are better than those obtained on IBM SP2 [20] and this shows that our technics are well suited for SMP architectures. Hence, the factorization of the COUP8000T matrix reached around 100 Gflops on 128 processors, so about 50% of the peak performance.

[1] http://www.labri.fr/scalapplix

Name	Number of processors			
	16	32	64	128
GRID1023	**2.59** (8.03)	**1.72** (12.11)	**1.18** (17.58)	**.94** (21.98)
CUBE39	**2.59** (8.62)	**1.87** (11.91)	**1.58** (14.15)	**1.58** (14.11)
CUBE47	**7.08** (9.83)	**4.94** (14.07)	**3.91** (17.80)	**3.34** (20.84)
BCSSTK32	**.30** (3.76)	**.25** (4.54)	**.24** (4.71)	**.25** (4.58)
TOOTH	**1.23** (5.06)	**.87** (7.18)	**.84** (7.45)	**.82** (7.59)
OCEAN	**2.41** (5.38)	**1.41** (9.19)	**1.04** (12.39)	**.97** (13.37)
M14B	**7.00** (8.72)	**4.25** (14.37)	**3.03** (20.15)	**2.70** (22.57)
OILPAN	**.44** (6.66)	**.37** (8.05)	**.32** (9.28)	**.29** (10.29)
QUER	**.52** (6.27)	**.42** (7.76)	**.39** (8.21)	**.37** (8.66)
INVEXT	**.76** (4.89)	**.54** (6.88)	**.52** (7.18)	**.56** (6.61)
SHIP001	**1.06** (8.50)	**.70** (12.74)	**.59** (15.28)	**.55** (16.19)
X104	**1.83** (9.35)	**1.38** (12.33)	**1.16** (14.72)	**1.22** (14.03)
MT1	**1.91** (11.03)	**1.24** (16.96)	**1.04** (20.14)	**.94** (22.42)
BMW3	**3.30** (9.11)	**2.23** (13.47)	**1.69** (17.70)	**1.46** (20.48)
MIXTANK	**1.20** (6.07)	**1.14** (6.38)	**1.12** (6.49)	**1.13** (6.42)
BMWCRA1	**4.84** (11.76)	**2.90** (19.29)	**1.83** (29.72)	**1.48** (37.31)
CRANKSEG1	**3.08** (9.73)	**1.91** (15.70)	**1.58** (19.02)	**1.29** (23.17)
SHIPSEC8	**4.56** (8.07)	**3.64** (10.10)	**2.73** (13.48)	**2.50** (14.71)
CRANKSEG2	**3.97** (11.58)	**2.45** (18.73)	**1.93** (23.78)	**1.59** (28.93)
SHIPSEC5	**6.26** (11.09)	**4.43** (15.68)	**3.33** (20.82)	**3.03** (22.87)
SHIP003	**7.20** (10.28)	**4.82** (16.59)	**3.49** (22.89)	**2.84** (27.78)
THREAD	**4.16** (9.33)	**3.12** (12.41)	**2.77** (13.98)	**2.40** (16.14)
COLOGB30	**.57** (7.54)	**.45** (9.50)	**.39** (10.92)	**.38** (11.24)
COLOGB75	**1.72** (8.88)	**1.11** (13.73)	**.87** (17.50)	**.86** (17.79)
COUP1500T	**28.71** (13.22)	**15.88** (23.91)	**7.94** (47.81)	**4.92** (77.15)
COUP2000T	**40.52** (12.36)	**19.67** (25.46)	**9.93** (50.43)	**5.80** (86.32)
COUP3000T	-	**29.48** (26.06)	**16.36** (46.96)	**8.67** (88.59)
COUP5000T	-	-	**26.49** (49.50)	**14.15** (92.57)
COUP8000T	-	-	**42.65** (49.00)	**22.49** (92.93)

Table 2. *Factorization performance results (***time** *in seconds and Gigaflops) on IBM SP3 in double precision.*

Name	Columns	NNZ_A	NNZ_L	OPC	Number of processors		
					356	512	768
COUP40000T	26.5e+06	1.85e+09	1.50e+10	10.8e+12	**34**	**27**	**20**

Table 3. *Factorization performance results in seconds on AlphaServer.*

Other experiments were run on the parallel computer at CEA (France). This supercomputer is an Compaq AlphaServer SC with 660 nodes ES45 (four EV7 at 1Ghz and 1Go per processor). The factorization times are closed from those obtained on the IBM SP3. On our largest problem (see table 3), we reach 500 Gflops on 768 processors, so about 50% of the peak performance.

4 Memory Aspects

In the previous sections, we have presented a mapping and scheduling algorithm for clusters of SMP nodes and we have shown the benefits on run-time performances of such strategies. In addition to the problem of run-time performances, another important aspect in the direct resolution of very large sparse systems is the great memory requirements usually needed to factorize the matrix in parallel. These memory requirements can be caused by either the structures needed for communication (the AUB structures in our case) or by the matrix cœfficients themselves.

To deal with those problems of memory management, our statically scheduled factorization algorithm can take advantage of the determinist access (and allocation) pattern to all data stored in memory. Indeed the data access pattern is determined

by the computational task ordering that drives the access to the matrix coefficient blocks, and by the communication priorities that drive the access to the AUB structures. In the next section, we present two ways of using this predictable data access pattern:

- the first one consists in reducing the memory used to store the AUB by allowing some AUB still uncompleted to be sent in order to temporaly free some memory. In this case, according to a memory limit given by the user, the AUB access pattern is used to determined which partially updated AUB should be sent in advance to minimize the impact on the run-time performance;

- the second one is applied to an "out-of-core" version of the parallel factorization algorithm. In this case, according to a memory limit given by the user, we use the coefficient block access pattern to reduce the I/O volume and to anticipate I/O calls.

4.1 Reducing the memory overcost by using partial agregate technique

A critical point in industrial large-scaled applications can be the memory overcost caused by the structures related to the distributed data managment and the communications.

In the case of the supernodal factorization with total agregation of the contributions, this overcost is mainly due to the memory needed to store the AUB until they are entirely updated and sent. Indeed, an agregated update block AUB is an overlapping block of all the contributions from a processor to a block mapped on another processor. Hence an AUB structure is present in memory since the first contribution is added within, and is released when it has been updated by its last contribution and actually sent. In some case, particulary for matrices issued from 3D problems, the amount of memory needed for the AUB still in memory can become important compared to the memory needed to store the matrix cœfficients. The table 4 shows the average percentage of memory needed to store the AUBs compared to the local matrix cœfficients. As shown, this percentage is very high and increases with the number of processors, particulary for the cases issued from 3D meshes (CUBE47, SHIP003, BMWCRA1).

	Number of processors			
Name	16	32	64	128
THREAD	**264.0**	**325.5**	**366.0**	**172.2**
SHIP003	**95.0**	**129.5**	**205.9**	**330.9**
CUBE47	**103.7**	**233.5**	**332.8**	**344.9**
BMWCRA1	**9.1**	**23.2**	**41.4**	**55.0**

Table 4. *Percentage of memory needed for AUB storage compared to memory needed for the matrix cœfficients in the factorized matrix.*

A solution to address this problem is to reduce the number of AUBs simultaneously present in memory. Then, the technique consists in sending some AUBs partially updated before their actual completion and then temporaly save some memory until the next contribution to be added in such AUB. This method is called *partial agregation.*

The *partial agregation* induces a time penalty compared to the *total agregation* due to more dynamic memory reallocations and to an increased volume of communications. Nevertheless, by using the knowledge of the AUB access pattern and the priority set on the messages, one can minimize this overcost.

Indeed by using the static scheduling, we are able to know by following the ordered task vector and the communication priorities, when an allocation or a deallocation will occur in the numerical factorization. That is to say that we are able to trace the memory consumption along the factorization tasks without actually run it. Then, given a memory limit set by the user, the technique to minimize the number of partially updated AUBs needed to enforce this limitation is to choose some partially updated AUBs among the AUBs in "memory" that will be updated again the later in the task vector. This is done whenever this limit is overtaken in the logical traversal of the factorization task vector.

The figure 1 shows the time penalty observed for 4 test problems with different levels of AUB memory constraint. That stresses the interest of the partial agregation technique; there is a fine trade-off between run-time performances provided by the total agregation strategy and the low memory overcost provided by an agregation-free algorithm. These results show that the memory reduction is acceptable in terms of time penalty up to 50% extra-memory reduction. As we can see, an extra-memory reduction about 50% induces a factorization time overhead between 2.7% and 28.4% compare to the time obtained with total agregation technique. In that context, our extra-memory management leads to a good memory scalability.

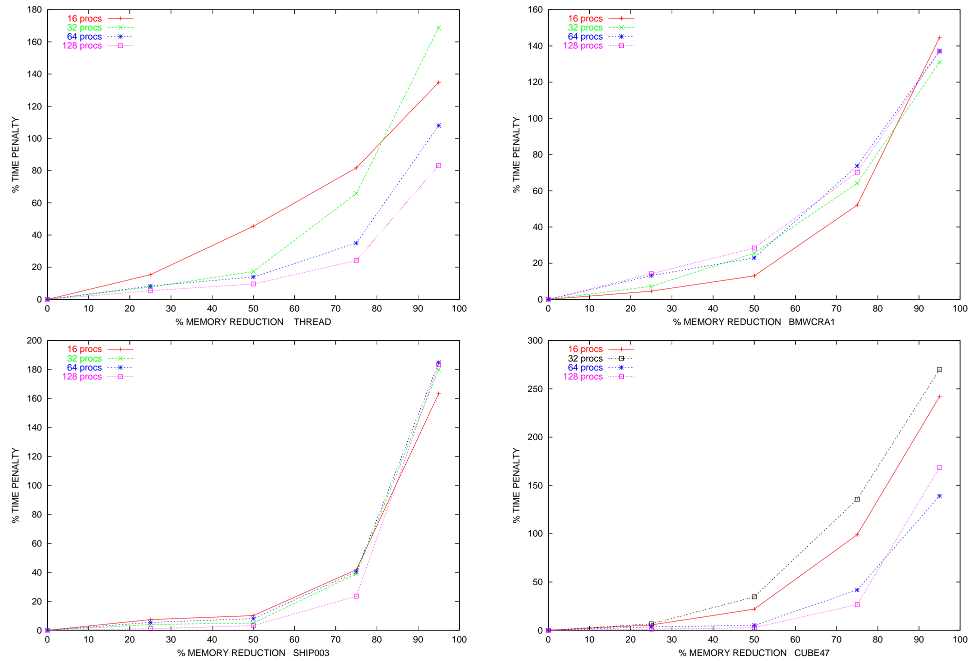

Figure 1. *Percentage of time penalty / percentage of memory overhead reduction.*

Futhermore, this technique allows us to factorize the AUDI matrix (3D problem from PARASOL collection with more than 5 Tflop which is a difficult problem for direct methods in terms of memory overhead) in 188s on 64 Power3 processors with a reduction of 50% of memory overhead (about 28 Gflops).

4.2 High performance "Out-Of-Core" technique

An "Out-of-Core" version consists in storing only the data needed for current computations. This technique suits naturally to our algorithm: indeed, as mentioned in the previous section, the data access pattern can be deduced from the computational task scheduling. It is commonly known that the time to read a block on disk will be more expensive than the time to compute the task associated to this block. So, the performance of the "Out-of-Core" factorization will strongly depend on the I/O volume. The aim of this study is to provide an efficient prefetch algorithm for asychronous I/O as well as to reduce the volume of data exchanges between disk and memory. We only consider an "Out-of-Core" version of the total agregation strategy. Indeed, the reception of a partially updated AUB requires to load a block from disk that is useless for the current computation.

During computation of the column-block k, the "Out-of-Core" algorithm needs to take into account both access to:
- column-blocks of the local matrix in $BStruct(L_{*k})$,
- the AUB structure needed to add contributions from column-block k for column-blocks mapped on other processors.

To reduce the swap of blocks between disk and memory, if we need to free memory, we will choose to swap a block that will be re-accessed the later during the factorization. Using our static scheduling and an algorithm similar to the one used for the partial agregation, we apply this criteria to build the best scheme to swap data.

5 Concluding remarks

In this paper, we have presented an efficient distribution scheme and the induced static scheduling of the block computations for a parallel sparse supernodal direct solver. We performed numerical experiments on various test problems with more than 10^7 unknowns and we have explain how to use this static scheduling to control the memory overhead inherent to direct methods.

We think that we have reached good performances for a generic MPI based implementation on a shared nothing environnement. In order to fully take advantage of clusters of SMP nodes, we plan to apply our algorithm to an MPI/thread implementation. In this version, all processors in a same SMP node will share the local part of the matrix distributed per node. This will allow to substitute all communications within a node by direct memory copies and to perform agregation of contributions per node. Communications inside a node will be suppressed and, in the same time, the number of communication between nodes will be strongly reduced.

Bibliography

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

[2] P. R. Amestoy and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. Supercomputer Applics*, 7:64–82, 1993.

[3] P. R. Amestoy, I. S. Duff, and J. Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods in Appl. Mech. Engrg.*, 184:501–520, 2000.

[4] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis, tuning and comparison of two general sparse solvers for distributed memory computers. Technical Report RT/APO/00/2, ENSEEIHT-IRIT, June 2000. France-Berkeley project report, also Lawrence Berkeley National Laboratory report LBNL-45992.

[5] 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.

[6] 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.

[7] C. Ashcraft, S. C. Eisenstat, J. W.-H. Liu, B. W. Peyton, and A. Sherman. A compute-ahead fan-in scheme for parallel sparse matrix factorization. In D. Pelletier, editor, *Fourth Canadian Supercomputing Symposium*, pages 351–361, June 1990.

[8] 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.

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

[10] J. M. Conroy, S. G. Kratzer, and R. F. Lucas. Multifrontal sparse solvers in message passing and data parallel environments - a comparitive study. In *Proceedings of PARCO*, 1993.

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

[12] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[13] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific and Statistical Computing*, 5(3):633–641, 1984.

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

[15] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. IEEE Trans. on Parallel and Distributed Systems, 8(5):502–520, May 1997.

[16] M. T. Heath, E. G.-Y. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.

[17] P. Hénon. *Distribution des Données et Régulation Statique des Calculs et des Communications pour la Résolution de Grands Systèmes Linéaires Creux par Méthode Directe*. PhD thesis, LaBRI, Universit Bordeaux I, Talence, France, November 2001.

[18] P. Hénon, P. Ramet, and J. Roman. A Mapping and Scheduling Algorithm for Parallel Sparse Fan-In Numerical Factorization. In *Proceedings of EuroPAR'99*, number 1685 in Lecture Notes in Computer Science, pages 1059–1067. Springer Verlag, September 1999.

[19] P. Hénon, P. Ramet, and J. Roman. PaStiX: A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions. In *Proceedings of Irregular'2000*, number 1800 in Lecture Notes in Computer Science, pages 519–525. Springer Verlag, May 2000.

[20] P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, January 2002.

[21] 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.

[22] G. Karypis and V. Kumar. METIS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0.* University of Minnesota, September 1998.

[23] X. S. Li. *Sparse Gaussian Elimination on High Performance Computers.* PhD thesis, University of California at Berkeley, 1996.

[24] X. S. Li and J. W. Demmel. A scalable sparse direct solver using static pivoting. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, March 22-24, 1999.

[25] J. W.-H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11(2):141–153, 1985.

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

[27] F. Pellegrini, J. Roman, and P. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. *Concurrency: Practice and Experience*, 12:69–84, 2000. Preliminary version published in *Proceedings of Irregular'99*, LNCS 1586, 986–995.

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

[29] 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.

[30] E. Rothberg and A. Gupta. An evaluation of left-looking, right-looking, and multifrontal approaches to sparse cholesky factorization on hierarchical-memory machines. *Int. J. High Speed Comput.*, 5:537–593, 1993.

[31] 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.

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