PaStiX:
A Parallel Direct Solver for Sparse SPD Matrices based on Efficient Static Scheduling and Memory Managment [1]

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

December 4, 2000

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

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; see [7] for a complete survey on high performance sparse factorization.

From a practical point of view, we have mainly worked on the parallelization of an industrial vectorized code for structural mechanics, which is a 2D and 3D finite element code and non linear in time. This computational finite element code solves plasticity (or thermo-plasticity, possibly coupled with large displacements) problems. Since the matrices of these systems do not have good properties, classical iterative methods do not behave well. Therefore, as we want to obtain a robust versatile industrial software tool, we must use high-performance direct sparse solvers; moreover, as we need to solve very large sparse systems (more than one million unknowns for 3D problems), parallelism is necessary for reasons of memory capabilities and acceptable solving time.

In this paper, we focus on the block partitioning and scheduling problem for high performance sparse LDL^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. 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, 15].

• 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 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.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.

There are mainly two approaches for numerical factorization algorithms: the *multifrontal* approach [2, 17], and the *supernodal* [18, 19] with *fan-in* or *fan-out* variations [5, 6]. Independently of these different methods, a static or dynamic scheduling of block computations can be used. For homogeneous parallel architectures, it is usefull to find an efficient static scheduling [8, 9, 16]. In this context, this scheduling can be induced by a fine cost computation/communication modeling.

The PSPASES solver [10] is based on a multifrontal approach without pivoting for symmetric definite positive systems. It uses MeTiS [11] for computing a fill-reducing ordering which is based on a multilevel nested dissection algorithm; when the graph is separated into p parts, one subgraph per processor, a multiple mininum degree (MMD [13]) 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 [12] are compared for nonsymmetric problems. The first uses a multifrontal approach with dynamic pivoting for stability while the second 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 [13]) ordering.The PARDISO solver [20] is an efficient implementation of a parallel sparse direct factorization for unsymmetric matrices in the context of shared memory architectures. Parallelism is exploited with a combination of left-looking and right-looking supernodal algorithms; the parallel pivoting methods use complete or Bunch and Kaufmann supernode pivotingin order to reach a compromise between numerical stability and scalability during the factorization process.

In this paper, we focus on the block partitioning and scheduling problem for high performance sparse supernodal LDL^T factorization without pivoting for symmetric definite positive systems; our algorithmic framework is closed to the PSPASES one [10]. In order to reduce significantly the number of messages, we aggregate locally contributions to a same block before sending them. The authors presented in [9] 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 scheduling of the block computations for a parallel solver based on a supernodal approach 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. Our study is suitable for heterogeneous parallel/distributed architectures whose performances are predictable by using a cost modeling for computations and communications. This paper extends this preliminary work by presenting a complete description of our mixed 1D/2D distribution strategy, a description and an analysis of partial aggregation technique (for memory constraints), complete numerical experiments for case studies with more than $1.5\,10^6$ unknowns on an IBM SP2 calculator, and our last experiments on an IBM SP3 with SMP nodes.

The paper is organized as follows. Section 2 describes our block repartitioning and scheduling algorithm. Section 3 provides many numerical experiments on an IBM SP2 for a representative class of large sparse matrices from industrial problems, including performance results and analysis; results for the two biggest case studies are also presented on an IBM SP3. Then, we conclude with remarks on the benefits of this study and on our future work.

2 Static partitioning and mapping

Let us consider the block data structure of the factored 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. We use a parallel supernodal version of sparse LDL^t factorization with total local aggregation: all non-local block contributions are aggregated locally in block structures. These aggregated update blocks, denoted in the following by AUB, can be built from the block symbolic factorization. The solver algorithm can be expressed in terms of dense block computations, or *tasks*, performed, as much as possible, on compacted sets of blocks for BLAS efficiency. The proposed algorithm can yield 1D (column-block) or 2D (block) distributions. The block computations as well as the emissions and receptions of aggregated update blocks are fully ordered with compatible priorities.

Before running the general parallel algorithm 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 give rise to several problems, which can be divided into two categories: on the 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.

The measure of workload is usually very rough because only numbers of operations are taken into account. However, in order to be efficient, solver algorithms are block-oriented to take advantage of BLAS subroutines, whose efficiencies are very far from being linear in terms of numbers of operations. Moreover, workload encompasses block computations but does not take into account all of the other phenomena that occur in parallel factorization, such as extraworkload generated by the local aggregation of contributions and idle-waits due to the latency generated by message passing.

The obtaining of high performances at run-time requires computational efficient solutions to several scheduling problems. These problems mainly consist in the order to process computational tasks that are locally ready and, in the order to send and to receive the AUB.

We tackle all of these problems by 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 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 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 yielded by block symbolic factorization. Pothen and Sun presented such a strategy in [16]. It starts by splitting the root and assigning it to the set of candidate processors Q. 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, recursively, each subtree is assigned to a subset of Q proportionally to its workload. During this partitioning phase, the column-blocks corresponding to large supernodes are split using the blocking size suitable to achieve good BLAS efficiency.

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 that leads this phase is to simulate parallel factorization as mapping comes along. Thus, the algorithm starts by mapping the leaves of the elimination tree (those which have only one candidate processor). Then, we choose to mapped the task whose all the contributions have been computed the earlier thanks to the simulation. Once this task is elected, we have to decide on which candidate processor this task is to be mapped. 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. So, we compute for each of the candidate processors the time at which it will have completed the task if it is mapped onto it. This is possible thanks to the simulation that gives the elapsed time on a processor and the time at which all contributions to this task have been computed (taking into account the overcost due to aggregation of local contributions). The task is mapped onto the candidate processor that will be able to compute it the soonest. Then we iterate the process untill all the tasks have been mapped. Once this step is over, the task computations are ordered with respect to the rank at which the tasks have been mapped. In the same manner, the sending of the AUB are ordered according to the rank of the tasks on the receiving processor.

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. Moreover, in order to apply the overlapping optimization made during the mapping phase, we can deduce a block computation and communication scheme that fully drives the parallel solver.

We have explained in the previous paragraph our strategy to improve the parallel factorization in terms of time. Another important problem that occurs in the industrial problems is the performance in terms of memory, particularly the memory overhead due to the extra structures needed to manage distributed data and communication exchanges.

In the case of the LDL^t supernodal factorization with total local aggregation of contributions, this overhead cost is mainly caused by the local aggregation of contributions. Indeed, an aggregated update block AUB_{ij} on a given processor is composed by an overlapped block structure of all the contributions from this processor to the block (i, j) mapped on another processor. Thus any AUB needs to be allocated in the local memory, since its first contribution has been computed until it is updated by its last contribution and sent according to the communication scheme. The total amount of simultanously allocated memory for AUB can be very important and often reachs a peak of 50 percent of the memory needed to store the distributed matrix. A solution to reduce this extra memory requirement is to reduce the number of simultanously allocated AUB by sending some of them before they are completed. That is to say that we allow some of the AUB to be sent before they have been updated by their last contribution, and to be reallocated when the next contribution following the sending of the partial AUB is computed. This method is called partial local aggregation.

In addition to the reduction of extra-memory, we are currently working on a more general version of our strategy that is suitable for heterogeneous architectures and more particulary those based on SMP nodes like the IBM SP3. Our previous communication time model is extended to take into account the (less costly) data exchanges by shared memory, and the (most costly) data exchanges performed by the network. For example, if we consider the AUB of the tasks already mapped on a processor p_1 to be sent to the task we want to mapped, and a candidate processor p_2 of this task, then the time for sending this AUB to p_2 is computed using the intra (*resp.* extra) node communication model whether p_1 and p_2 do (*resp.* do not) belong to the same SMP node. Numerical experiments are in progress to validate this extension and some preliminary results are given in section 3.6. More generally, our strategy is suitable for heterogeneous architectures with predictable performances.

3 Numerical Experiments

In section 3.1, we first introduce our experimental environment. Then we present in section 3.2 the time performances for the pre-processing steps. Before the analysis of the parallel factorization times in section 3.4, we describe some experiments to find the threshold for mixed 1D/2D distributions (see section 3.3). Finally, section 3.5 gives some results for the partial aggregation technique and section 3.6 presents our last experiments on a IBM SP3 with SMP nodes.

3.1 Experimental environment and test cases

All of the algorithms described in this paper have been integrated in the PaStiX software [9], based on libraries that make use of version 3.4 of the Scotch static mapping and sparse matrix ordering software package [14], both developed at LaBRI.

The parallel experiments were run on an 192 node IBM SP2 at CINES (Montpellier, France); the processors are 120 MHz Power2SC thin nodes (480 MFlops peak performance) having 256 MBytes of physical memory each. Switch interrupts

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 for the factorization.*

Name	Columns	$\mathrm{NNZ_A}$	$\mathrm{NNZ_L}$	OPC	Description
GRID1023	1046529	4179980	5.615708e+07	2.083481e+10	Regular 2D Mesh
CUBE39	59319	730778	2.210534e+07	2.240674e+10	Regular 3D Mesh
CUBE47	103823	1290898	4.828456e+07	6.963850e+10	Regular 3D Mesh
BCSSTK32	44609	985046	5.239146e+06	1.162900e+09	Rutherford-Boeing
M14B	214765	1679018	6.236747e+07	6.112540e+10	Rutherford-Boeing
OCEAN	143437	409593	2.029997e+07	1.301477e+10	Rutherford-Boeing
BBMAT	38744	1274141	1.716094e+07	1.250040e+10	Rutherford-Boeing
TOOTH	78136	452591	1.031143e+07	6.267094e+09	Rutherford-Boeing
SMDOOR	162610	3873534	2.541937e+07	1.530774e+10	PARASOL
OILPAN	73752	1761718	8.912337e+06	2.984944e+09	PARASOL
BMWCRA_1	148770	5247616	6.597301e+07	5.701988e+10	PARASOL
INVEXTR1	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
SHIP003	121728	3982153	5.872912e+07	8.008089e+10	PARASOL
SHIPSEC5	179860	4966618	5.649801e+07	6.952086e+10	PARASOL
SHIPSEC8	114919	3269240	3.572761e+07	3.684269e+10	PARASOL
THREAD	29736	2220156	2.404333e+07	3.884020e+10	PARASOL
X104	108384	5029620	2.634047e+07	1.712902e+10	PARASOL
BMW3_2	227362	5530634	4.420244e+07	3.007981e+10	PARASOL
CRANKSG1	52804	5280703	3.142730e+07	3.007141e+10	PARASOL
CRANKSG2	63838	7042510	4.190437e+07	4.602878e+10	PARASOL
COLOGB30	20373	1394292	7.849464e+06	4.359803e+09	3D Cologne
COLOGB75	50208	3473292	2.248613e+07	1.532035e+10	3D Cologne
COUP1500T	994983	70303275	5.219997e+08	3.79847e+11	3D Coupole
COUP2000T	1326483	93734775	6.888813e+08	5.00924e+11	3D Coupole

are enabled with default delay to perform non-blocking communication efficiently. The sequential experiments for the preprocessing steps were run on one node of this IBM SP2. All computations are performed in double precision and all time results are given in seconds.

Our experiments are performed on a collection of sparse matrices from the Rutherford-Boeing Collection, from the PARASOL ESPRIT Project and from CEA/CESTA (3D Cologne Challenge, Coupole). The values of the associated measurements in table 1 come from scalar column symbolic factorization.

3.2 Partitioning and mapping results

Table 2 presents the performances of our scheduling strategy for a number of processors varying from 2 to 64. We can notice that this algorithm is poor time consuming; its complexity grows as $\Theta(M.log(P))$ where M is the number of blocks and P the number of processors.

Table 2. *Performance results in seconds the scheduling algorithm, the Symbolic Factorization and the Partionning and Ordering step. For 2D and 3D grids we don't use the* SCOTCH *software but the optimal nested dissection algorithm.*

Name	Number of processors						SF_Time	PO_Time
	2	4	8	16	32	64		
GRID1023	1.15	1.12	1.77	2.20	3.27	10.27	3.31	-
CUBE39	0.28	0.38	0.47	0.80	1.63	3.11	0.58	-
CUBE47	0.94	1.17	1.39	2.70	4.72	8.92	1.35	-
BCSSTK32	0.13	0.14	0.16	0.19	0.28	0.52	0.55	6.83
M14B	5.89	6.73	7.52	8.42	10.06	16.32	2.43	150.81
OCEAN	3.89	4.11	4.24	4.86	6.43	12.55	1.29	63.76
TOOTH	1.82	1.94	2.16	2.50	3.41	7.22	0.96	38.10
SMDOOR	0.31	0.34	0.39	0.51	0.98	2.90	1.93	9.27
OILPAN	0.14	0.14	0.14	0.19	0.31	0.92	0.88	3.77
BBMAT	0.49	0.58	0.69	0.87	1.06	1.69	0.69	65.75
BMWCRA_1	0.33	0.39	0.58	0.87	1.08	1.56	2.59	45.37
INVEXTR1	0.87	0.96	1.17	1.39	1.69	2.40	0.62	47.27
MIXTANK	0.57	0.85	0.97	1.16	1.70	3.72	0.61	46.61
MT1	0.08	0.12	0.18	0.26	0.84	1.46	1.80	12.26
QUER	0.10	0.10	0.10	0.14	0.29	0.62	0.69	2.98
SHIP001	0.05	0.05	0.08	0.11	0.19	0.49	0.88	5.51
SHIP003	0.70	0.95	1.32	1.85	3.36	8.53	2.05	14.25
SHIPSEC5	0.88	0.94	1.38	1.76	3.28	8.00	2.43	12.90
SHIPSEC8	0.51	0.75	1.02	1.43	2.85	7.88	1.62	8.52
THREAD	0.11	0.51	0.56	0.71	2.35	6.20	0.87	11.07
X104	0.13	0.19	0.22	0.31	0.78	2.15	1.95	10.86
BMW3_2	0.57	0.73	0.83	1.00	1.80	5.13	3.12	27.67
CRANKSG1	0.09	0.13	0.26	0.41	0.89	2.02	1.99	17.08
CRANKSG2	0.11	0.17	0.31	0.52	0.71	1.91	2.55	20.84
COLOGB30	0.03	0.05	0.06	0.80	0.17	0.33	0.10	6.90
COLOGB75	0.08	0.11	0.14	0.20	0.29	0.88	0.24	17.57
COUP1500T	1.46	1.55	1.69	1.99	2.65	5.19	1.17	-
COUP2000T	2.10	2.13	2.27	2.70	3.57	6.77	2.05	-

3.3 Experimental threshold for mixed 1D/2D distribution

The table 3 presents some experiments on the threshold to shift beetween an 2D distribution and a 1D distribution during the first step of the partitioning and mapping algorithm (cf. section 2). For the moment, our threshold criterion only takes into account the number of candidate processors for a node. We plane to improve this threshold criterion by taking into account the computation cost of the subtree rooted at the node in addition to this number of candidate processor. As shown in table 3, a good experimental approximation gives a threshold around 12 processors on the IBM SP2 calculator. We will use this criterion to perform all the tests in the sections 3.4 - 3.6.

3.4 Parallel factorization results

Table 4 reports the performances 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

Table 3. *Influence of the threshold between 2D and 1D distributions on factorization time.*

Name	Threshold	Number of processors				
		4	8	16	32	64
CUBE47	6	-	39.59	27.17	17.82	14.77
	8	-	38.61	25.08	**17.05**	13.53
	12	-	**38.23**	**24.08**	17.15	**12.59**
	16	-	38.37	25.14	19.54	15.21
BMWCRA_1	6	-	31.03	18.91	9.59	7.01
	8	-	31.12	19.06	9.62	6.56
	12	-	30.97	**18.75**	**9.55**	**5.94**
	16	-	**30.77**	19.11	9.55	6.11
COLOGB75	6	18.70	10.18	5.75	4.16	3.25
	8	**18.61**	8.99	5.84	4.09	3.34
	12	18.72	**8.99**	**5.69**	**3.89**	3.14
	16	20.24	8.99	5.85	4.02	**3.13**

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 (operations count) value because of amalgamation performed by the ordering and of block computations.

The table 4 shows that a good scalability is achieved for all the test problems. On both both moderate and large size grid and irregular problems, the measured performances vary between 1.08 and 3.24 Gigaflops on 16 nodes and between 2.36 and 11.62 on 64 nodes, the better scalability being achieved on the largest tests.

In [9], we compare factorization times performed by our PaStiX software and by PSPASES version 1.0.3 based on a multifrontal approach [10], on a collection of sparse matrices in the RSA format. 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. 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.

3.5 Influence of the partial aggregation strategy

The table 5 presents, for one test, the time penalty on the parallel factorization induced by the partial aggregation method. The results are performed on different numbers of processors and different percentages of memory reduction. The first column gives the percentage of memory reduction with regard to the total extra-memory due to the AUB on a processor. The other columns give the time of the parallel factorization using this memory reduction. The last row of the table gives the ratio R_{mem} in percentage between the memory needed to store the AUB and the memory needed to store the distributed matrix coefficients on a processor.

One can notice that the times for 0% (that correspond to a total aggregation

Table 4. *Factorization performance results (***time** *in seconds and Gigaflops) on the IBM SP2.*

Name	Number of processors				
	4	8	16	32	64
GRID1023	-	**14.57** (1.43)	**8.61** (2.42)	**5.56** (3.74)	**3.83** (5.44)
CUBE39	**24.20** (0.93)	**13.20** (1.70)	**8.88** (2.52)	**6.32** (3.55)	**5.03** (4.46)
CUBE47	**73.31** (0.95)	**38.23** (1.82)	**24.08** (2.89)	**17.15** (4.06)	**12.59** (5.53)
BCSSTK32	**2.46** (0.47)	**1.41** (0.82)	**1.08** (1.08)	**0.78** (1.49)	**0.71** (1.63)
M14B	-	**46.95** (1.30)	**25.84** (2.37)	**14.60** (4.19)	**9.21** (6.64)
OCEAN	**18.39** (0.71)	**9.78** (1.33)	**6.26** (2.08)	**4.12** (3.16)	**4.17** (3.12)
BBMAT	**14.73** (0.85)	**8.55** (1.46)	**5.24** (2.39)	**3.39** (3.69)	**2.72** (4.60)
TOOTH	**10.70** (0.59)	**6.60** (0.95)	**4.01** (1.56)	**2.77** (2.26)	**2.49** (2.52)
SMDOOR	**15.52** (0.99)	**9.10** (1.68)	**5.25** (2.91)	**4.07** (3.76)	**2.91** (5.25)
OILPAN	**3.81** (0.78)	**2.23** (1.34)	**1.51** (1.98)	**1.03** (2.90)	**0.92** (3.24)
BMWCRA_1	-	**30.97** (1.84)	**18.75** (3.04)	**9.55** (5.97)	**5.94** (9.60)
INVEXTR1	**6.31** (0.60)	**4.03** (0.93)	**2.52** (1.49)	**1.76** (2.14)	**1.57** (2.40)
MIXTANK	**10.02** (0.73)	**6.33** (1.16)	**3.80** (1.93)	**3.32** (2.20)	**3.10** (2.36)
MT1	**20.35** (1.04)	**12.91** (1.63)	**6.96** (3.03)	**4.33** (4.87)	**3.51** (6.01)
QUER	**4.46** (0.74)	**2.77** (1.18)	**1.90** (1.73)	**1.35** (2.43)	**1.04** (3.16)
SHIP001	**10.91** (0.83)	**6.13** (1.47)	**3.78** (2.39)	**2.34** (3.86)	**1.96** (4.60)
SHIP003	**109.78** (0.73)	**45.83** (1.75)	**26.86** (2.98)	**16.69** (4.80)	**12.03** (6.66)
SHIPSEC5	**79.12** (0.88)	**40.36** (1.72)	**21.46** (3.24)	**15.36** (4.53)	**11.76** (5.91)
SHIPSEC8	**43.40** (0.85)	**27.71** (1.33)	**15.40** (2.39)	**11.93** (3.09)	**9.23** (3.99)
THREAD	**41.14** (0.94)	**22.85** (1.70)	**13.45** (2.89)	**11.79** (3.29)	**6.70** (5.80)
X104	**19.69** (0.87)	**11.02** (1.55)	**6.32** (2.71)	**5.30** (3.23)	**3.85** (4.44)
BMW3_2	**34.02** (0.88)	**20.78** (1.45)	**11.52** (2.61)	**7.90** (3.81)	**6.67** (4.51)
CRANKSG1	**33.54** (0.90)	**19.20** (1.57)	**10.54** (2.85)	**6.82** (4.41)	**5.34** (5.63)
CRANKSG2	**47.99** (0.96)	**25.33** (1.82)	**14.39** (3.20)	**8.45** (5.45)	**6.04** (7.62)
COLOGB30	**5.25** (0.83)	**3.23** (1.35)	**2.06** (2.12)	**1.69** (2.58)	**1.34** (3.25)
COLOGB75	**18.72** (0.82)	**8.99** (1.70)	**5.69** (2.69)	**3.89** (3.94)	**3.14** (4.88)
COUP1500T	-	-	-	-	**33.01** (11.51)
COUP2000T	-	-	-	-	**43.12** (11.62)

Table 5. *Influence of memory reduction (with partial aggregation technique) on the factorization time for BMWCRA_1 matrix.*

% reduction	Number of processors			
BMWCRA_1	8	16	32	64
0%	**35.04**	**20.23**	**12.37**	**7.30**
20%	**35.94** (+2.5%)	**20.82** (+2.9%)	**12.77** (+3.2%)	**7.40** (+1.4%)
30%	**37.97** (+8.4%)	**22.24** (+9.9%)	**13.17** (+6.5%)	**7.75** (+6.2%)
40%	**41.34** (+18.0%)	**26.32** (+30.1%)	**16.36** (+32.2%)	**10.12** (+38.6%)
50%	**45.10** (+28.7%)	**30.69** (+51.7%)	**19.28** (+55.9%)	**12.51** (+71.4%)
R_{mem}	9%	22%	35%	59%

scheme) are greater than those given in table 4. The reason is that allocations of the AUB to perform partial aggregation are *dynamically* made as they come along the factorization whereas, they are made *statically* in table 4. These results show that the memory reduction is acceptable, in terms of time penalty, up to 30% of extra-memory reduction. For larger reduction percentages, the overhead increase as the number of processors grows. In that context, our extra-memory management leads to a good memory scalability.

3.6 Experiments on IBM SP3

In this section, we present our last experiments on our two biggest test cases that run on an IBM SP3 at the University of Minnesota. This calculator consists of 78 WinterHawk+ nodes; each WinterHawk+ node has 4 processors (375 MHzPower3) with more than 4 GB of memory. On 16 nodes (64 processors) the factorization of the COUP1500T matrix is performed in 11.8s (32.2 gigaflops), and the COUP2000T matrix in 15.1s (33.3 gigaflops). We think that the achieved performances are rather good results; they confirm that our scheduling strategy is suitable for heterogeneous SMP architectures.

4 Conclusion and Perspectives

In this paper, we have presented an efficient mixed 1D/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 finer 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. The experimental studies concerning the partial aggregation strategy and SMP parallel machines must be carried on and extended to more general heterogeneous architectures.

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 Appl. Mech. Eng.*, 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.

[5] C. Ashcraft, S. C. Eisenstat, J. W.-H. Liu, B. W. Peyton, and A. Sherman. A compute-ahead implementation of the fan-in sparse distributed factorization scheme. Technical report, Departement of Computer Science, York University, September 1994.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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, 1998.

[12] 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*, pages 22–24, March 1999.

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

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

[15] 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.

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

[17] 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.

[18] 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.

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

[20] O. Schenk, K. Gärtner, and W. Fichtner. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. *BIT*, 40(1):158–176, 2000.