On using an hybrid MPI-Thread programming for the implementation of a parallel sparse direct solver on a network of SMP nodes

Pascal Hénon, Pierre Ramet and Jean Roman

LaBRI, UMR CNRS 5800 & ENSEIRB
INRIA Futurs ScAlApplix Project,
351, cours de la Libération, F-33405 Talence, France
{henon|ramet|roman}@labri.fr

Abstract. Since the last decade, most of the supercomputer architectures are based on clusters of SMP nodes. In those architectures the exchanges between processors are made through shared memory when the processors are located on a same SMP node and through the network otherwise. Generally, the MPI implementations provided by the constructor on those machines are adapted to this situation and take advantage of the shared memory to treat messages between processors in a same SMP node. Nevertheless, this transparent approach to exploit shared memory does not avoid the storage of the extra-structures needed to manage efficiently the communications between processors. For high performance parallel direct solvers, the storage of these extra-structures can become a bottleneck. In this paper, we propose an hybrid MPI-thread implementation of a parallel direct solver and analyse the benefits of this approach in terms of memory and run-time performances.

1 Introduction and Background

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. The authors presented in previous works [2–4] an efficient static scheduling based on a mixed 1D/2D block distribution for a parallel supernodal version of sparse Cholesky factorization with total *local aggregation on processors*. Parallel experiments, using MPI communications, were run on IBM SP machines at CINES (Montpellier, France) on a large collection of sparse matrices containing industrial 3D problems that reach 1 million of unknowns, and have shown that our PaStiX software compares very favorably to PSPASES [6].

Nevertheless, a major memory bottleneck for our parallel supernodal factorization scheme is caused by this local aggregation mechanism. The local aggregation mechanism is due to the fact that during the factorization of some local column-blocks, a processor has to update several times a block A_{ij} mapped on another processor. Indeed, it would be costly to send each contribution for a same non-local block in separated messages. As illustrated on figure 1, the so-called “local aggregation” variant of the supernodal parallel factorization consists in adding locally any contribution for a non-local block A_{ij} in a local temporary block noted AUB_{ij} (*aggregated update block*) and sending it once all the contributions destinated to the non-local block A_{ij} have been added.

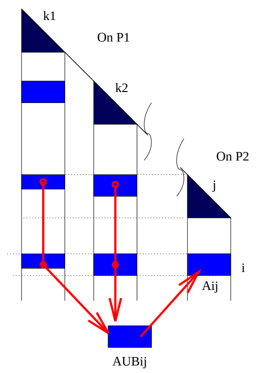

Fig. 1. Local aggregation of block updates. Column-block k_1 and k_2 are mapped on processor P_1, column-block j is mapped on processor P_2. Contributions from the processor P_1 to the block A_{ij} of processor P_2 are locally summed in AUB_{ij}.

For very large matrices from 3D problems, the highest peak of memory consumption due to the storage of aggregated block contributions during the factorization amounts several time the local factorized matrix part on a processor. Thus in the *full MPI* version of our parallel solver, we introduced an optimized communication mechanism to control the overhead of memory while loosing acceptable run-time performance [4].

Until now, we have discussed the parallelization in a full distributed memory environment. Each processor was assumed to manage a part of the matrix in its own memory space, and any communication needed to update non local blocks of the matrix was considered under the message passing paradigm. Nowaday, the massively parallel high performance computers are generally designed as networks of SMP nodes. Each SMP nodes consists in a set of processors that share the same physical memory. In [5], we have studied the improvement of our static scheduling to take into account the communication modeling of multilevel memories on network of SMP nodes. However, on those SMP-nodes architectures, to fully exploit shared memory advantages, a relevant approach is to use an hybrid MPI-thread implementation.

In the framework of direct solver, this approach aims at solving 3D problems with more than 10 millions of unkowns, which is now a reachable challenge with these new SMP supercomputers. The rational that motived this hybrid implementation was that the communications within a SMP node can be advantageously substituted by direct accesses to shared memory between the processors in the SMP node using threads. As a consequence, the MPI communication are only used between processors that host threads from different MPI processes.

This kind of approach has been used in WSMP [1] which is a the parallel multifrontal solver. The main difference between WSMP and our work is on the way to exploit parallelism inside the SMP nodes. In WSMP approach, the parallelism inside SMP nodes is automatically exploited using multi-threaded BLAS. In our approach, we use our static regulation algorithms to decide exactly how the factorization of a supernode has to be splitted in elementary BLAS tasks between threads.

The remainder of the paper is organized as follows: the section 2 describes the main features of our method. We provide some experiments in section 3. At last, we give some conclusions in section 4.

2 Overview of our hybrid MPI-Thread implementation

As said in the previous section, the local aggregation mechanism is inherent to the message passing model used in our parallel factorization algorithm. Thus, in an SMP context, the simplest way to lower the use of aggregate update blocks is to avoid the message passing communications between processors on a same SMP node. In order to avoid these intra-node communications, we use threads that share the same memory space. In an SMP node, a unique MPI process spawns a thread on each processor. These threads are then able to address the whole memory space of the SMP node. Each thread is therefore able to access the matrix part mapped on its MPI process. By this way, though the computational tasks are distributed between the threads, any update of the matrix part of the MPI process is directly performed in the local matrix. The communication between MPI processes still uses the local aggregation scheme we described earlier.

This hybrid MPI-thread strategy is pictured on figures 2 and 3. The figure 2 sketches the situation when each processor is assigned a MPI process. In this case, all the communications are performed using MPI. The extra-memory needed to store the aggregate update blocks (AUB) is maximal since each MPI process communicates their updates to any other MPI process through AUBs. The figure 3 shows the situation when only one MPI process is used per SMP node and inside each SMP node a thread is assigned on each processor. In this case, all the threads on a SMP node share the same memory space and update directly any column block assigned to their MPI process; in addition, the local aggregation is made globally for the whole set of column blocks mapped on the MPI process. This implies that the amount of extra-memory required for the storage of the AUBs depends only on the number of MPI processes. Therefore, the additional memory needed to store the AUBs is all the more reduced that the SMP node are wider (in terms of number of processors).

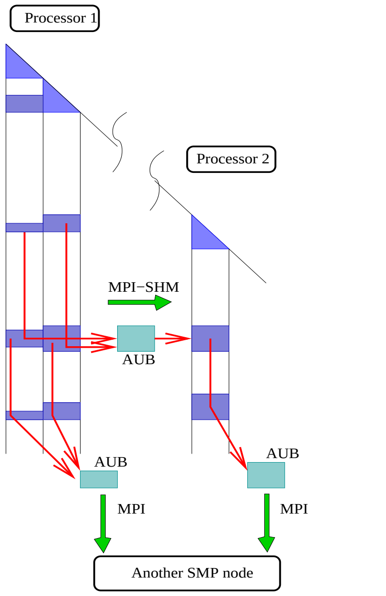

Fig. 2. Processor 1 and 2 belong to the same SMP node. Data exchanges when only MPI processes are used in the parallelization.

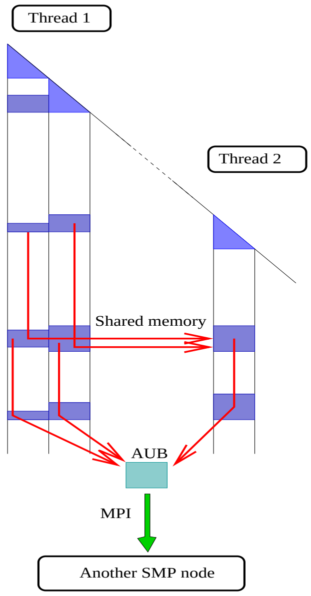

Fig. 3. Thread 1 and 2 are created by one MPI process. Data exchanges when there is one MPI process per SMP node and one thread per processor.

3 Experiments

All of the algorithms described in this paper have been integrated in the PaStiX software [2, 3], that make use of the static mapping and sparse matrix ordering software package Scotch [8] version 3.4.

The parallel experiments were run on an 28 NH2 nodes [IBM SP3] (16 Power3+ with 16Go per node) located at CINES (Montpellier, France) with a network based on a Colony switch. We have also used 2 SMP nodes [IBM SP4] (32-ways Power4+ with 64Go per node) with a network based on a Federation switch to validate our approach on our largest test cases. Switch interrupts are enabled with default delay to perform non-blocking communication efficiently. All computations are performed in double precision (the relative backward error observed on our problems is arround 10^{-15} for our direct factorization) and all time results are given in seconds. The blocking size parameter for BLAS3 computations is set to 60 and we use here a one dimensional distribution as default. We have apply an LDL^t factorization on symmetric matrices and an LU factorization for unsymmetric matrices (but with a symmetric pattern). In all the following tables, the symbol "-" is used when the time measurments are not significant due to memory swapping.

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 factorized matrix L and OPC is the number of operations required for the factorization. Matrices are sorted in decreasing order of $\frac{\mathrm{NNZ_L}}{\mathrm{OPC}}$ which is a measure of the potential data reuse [7].

Name	Columns	$\mathrm{NNZ_A}$	$\mathrm{NNZ_L}$	OPC	$\frac{\mathrm{NNZ_L}}{\mathrm{OPC}}$	Description
OILPAN	73752	1761718	8.912337e+06	2.984944e+09	2.98e-3	symetric
QUER	59122	1403689	9.118592e+06	3.280680e+09	2.78e-3	symetric
SHIP001	34920	2304655	1.427916e+07	9.033767e+09	1.58e-3	symetric
X104	108384	5029620	2.634047e+07	1.712902e+10	1.54e-3	symetric
MT1	97578	4827996	3.114873e+07	2.109265e+10	1.48e-3	symetric
INLINE	503712	18660027	1.762790e+08	1.358921e+11	1.29e-3	symetric
BMWCRA1	148770	5247616	6.597301e+07	5.701988e+10	1.16e-3	symetric
CRANKSG1	52804	5280703	3.142730e+07	3.007141e+10	1.05e-3	symetric
SHIPSEC8	114919	3269240	3.572761e+07	3.684269e+10	0.97e-3	symetric
CRANKSG2	63838	7042510	4.190437e+07	4.602878e+10	0.91e-3	symetric
SHIPSEC5	179860	4966618	5.649801e+07	6.952086e+10	0.81e-3	symetric
SHIP003	121728	3982153	5.872912e+07	8.008089e+10	0.73e-3	symetric
THREAD	29736	2220156	2.404333e+07	3.884020e+10	0.62e-3	symetric
AUDI	943695	39297771	1.214519e+09	5.376212e+12	2.26e-4	symetric
MHD1	485597	24233141	1.629822e+09	1.671053e+13	9.75e-5	unsymetric

Our experiments were performed on a collection of sparse matrices from the PARASOL ESPRIT Project and from CEA. The values of the associated measurments in Table 1 come from scalar column symbolic factorization.

On each MPI process p, we compute the size of the local allocation for matrix coefficients (denoted coeff_alloc(p)) and of the local allocation for the AUB (denoted AUB_alloc(p)). The memory efficiency $mem_{\mathrm{eff}}(P)$, on P processes, is defined as follow:

$$mem_{\mathrm{eff}}(P) = \frac{\sum_p \{\mathrm{coeff\_alloc}(p)\}}{P \cdot \max_p \{\mathrm{coeff\_alloc}(p) + \mathrm{AUB\_alloc}(p)\}}$$

The memory efficiency measures how the total memory allocations (matrix coefficients and AUBs) is reduced when the number of processors increases. When $mem_{\mathrm{eff}}(P) = 1$, it means that there is no memory overhead due to the parallelization; this can only happen when a single MPI process is used.

In table 2, for the two largest symmetric problems, the number of processors vary between 16 (1 NH2 node) and 128 (4 NH2 nodes) of the IBM SP3. We analyze the factorization time and the memory efficiency when we use 1, 4, 8 and 16 threads per MPI process. As expected, we can see that using 1 thread for each CPU is always profitable for memory efficiency. For instance, with the AUDI matrix, on 128 processors, the memory efficiency increases from 0.23 to 0.70. Beside, we can notice that the best factorization time is almost always obtain with 8 threads by MPI process. Figure 4 shows graphically that we keep a good memory efficiency and a good time scalability when we increase the number of threads by MPI process.

Table 2. Factorization time (and memory efficiency) on SP3.

Name	Number of processors			
	16	32	64	128
INLINE 1 thread/process	**13.39** (0.77)	7.99 (0.70)	6.07 (0.59)	4.58 (0.42)
INLINE 4 threads/process	14.23 (0.94)	7.08 (0.89)	4.28 (0.84)	3.33 (0.78)
INLINE 8 threads/process	13.46 (0.98)	**6.68** (0.94)	**4.12** (0.89)	**3.07** (0.81)
INLINE 16 threads/process	13.89 (**1.00**)	8.41 (**0.98**)	4.59 (**0.94**)	3.51 (**0.90**)
AUDI 1 thread/process	-	-	211.35 (0.34)	134.45 (0.23)
AUDI 4 threads/process	**472.67** (0.81)	266.89 (0.71)	155.23 (0.60)	87.56 (0.43)
AUDI 8 threads/process	476.40 (0.93)	**265.09** (0.81)	**145.19** (0.70)	**81.90** (0.54)
AUDI 16 threads/process	481.96 (**1.00**)	270.16 (**0.89**)	152.58 (**0.83**)	86.07 (**0.70**)

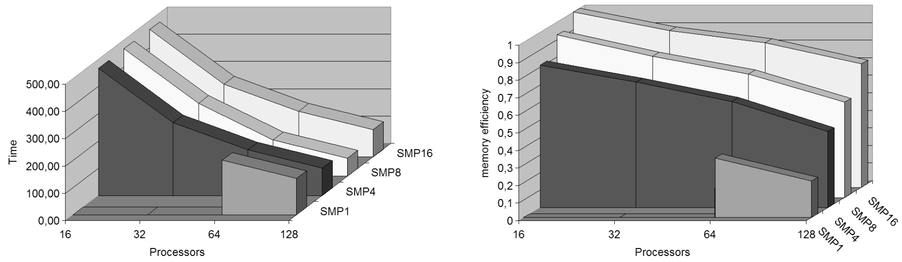

Fig. 4. Factorization time and memory efficiency on IBM SP3 for AUDI problem.

In table 3, we present experiments on IBM SP4. For 1 node (32 processors) then 2 nodes (64 processors), we increase the number of threads by MPI process between 4 to 32. Here again, the memory reduction is really substantial. We also notice that the best factorization time is obtain when using 8 threads by MPI process. This can be explained by the fact that this kind of node presents 4 I/O cards for communications on the Federation switch. This behaviour is verified for both symmetric problem (AUDI test case) and unsymmetric problem (MHD1 test case).

Another interesting remark is that the MPI-thread implementation allows us to manage more efficiently a two dimensional distribution mapping. It is well known that this kind of distribution is payfull in terms of scalability compared to a one dimensional distribution for large matrices arising from 3D problems.

Table 3. Factorization time (and memory efficiency) on SP4.

Name	Number of processors	
	32	64
AUDI 4 threads/process	94.21 (0.71)	60.03 (0.57)
AUDI 8 threads/process	**93.14** (0.82)	47.80 (0.74)
AUDI 16 threads/process	96.31 (0.92)	**47.28** (0.81)
AUDI 32 threads/process	100.40 (**1.00**)	51.02 (**0.92**)
MHD1 4 threads/process	199.17 (0.29)	115.97 (0.16)
MHD1 8 threads/process	**187.89** (0.49)	**111.99** (0.31)
MHD1 16 threads/process	197.99 (0.73)	115.79 (0.49)
MHD1 32 threads/process	202.68 (**1.00**)	117.80 (**0.78**)

Indeed, a two dimensional distribution generates a lot of small messages that are advantageously substituted by direct accesses to shared memory.

In table 4, we provide some results on a 3D magneto-hydrodynamic problem (MHD1 test case) when we set a two dimensional distribution on the 4 first levels of the block elimination tree. A complete analysis is out of the scope of this paper, but this result illustrates another benefit of using our MPI-thread implementation.

Table 4. MHD1 with 1D or 2D distribution on IBM SP4 (32 processors).

Name	Implementation	
	Full MPI	MPI-thread
MHD1 with 1D distribution	115.97	117.80
MHD1 with 2D distribution	111.51	**90.68**

Finally, we give some comparisons (see tables 5 and 6) on the IBM SP3 for the whole set of test cases in terms of factorization time and memory efficiency for both MPI and MPI-thread implementations. While the memory reduction is the most crutial criteria to optimize for parallel direct solver implementations, we focus our study on setting 1 thread for each CPU unit (that is to say 16 threads by MPI process).

Table 5. Factorization time (and memory efficiency) on SP3 (MPI only).

Name	Number of processors			
	16	32	64	128
OILPAN	**.61** (0.74)	**.38** (0.71)	**.37** (0.62)	**.37** (0.58)
QUER	**.90** (0.69)	**.45** (0.69)	**.45** (0.55)	**.47** (0.55)
SHIP001	**1.39** (0.67)	**.78** (0.59)	**.75** (0.48)	**.73** (0.46)
X104	**2.76** (0.58)	**1.86** (0.49)	**1.79** (0.39)	**1.12** (0.28)
MT1	**3.52** (0.62)	**1.55** (0.51)	**1.41** (0.41)	**1.53** (0.33)
INLINE	**13.39** (0.77)	**7.99** (0.70)	**6.07** (0.59)	**4.58** (0.42)
BMWCRA1	**6.32** (0.72)	**3.39** (0.59)	**2.49** (0.43)	**2.18** (0.37)
CRANKSEG1	**4.19** (0.58)	**2.04** (0.47)	**1.58** (0.40)	**1.92** (0.32)
SHIPSEC8	**6.07** (0.45)	**3.87** (0.34)	**3.70** (0.26)	**3.71** (0.23)
CRANKSEG2	**5.19** (0.54)	**2.94** (0.43)	**2.24** (0.40)	**2.64** (0.28)
SHIPSEC5	**9.42** (0.46)	**5.52** (0.38)	**4.67** (0.27)	**4.78** (0.21)
SHIP003	**9.50** (0.55)	**5.84** (0.47)	**4.77** (0.30)	**4.84** (0.23)
THREAD	**6.48** (0.39)	**3.65** (0.30)	**3.90** (0.24)	**3.90** (0.20)
AUDI	-	-	**211.35** (0.34)	**134.45** (0.23)

These results are in agreement with previous remarks and are also verified on IBM SP4. But, on this architecture, the factorization times obtained are less significant because for those problem sizes, the measured times are often lower than 1 second. As a conclusion for these experiments, we obtain a good memory scalability and the factorization time is most often reduced whatever the number of processors, thanks to the MPI-thread implementation. Up to 64

Table 6. Factorization time (and memory efficiency) on SP3 (MPI-thread).

Name	Number of processors			
	16	32	64	128
OILPAN	**.95** (1.00)	**.60** (0.96)	**.36** (0.91)	**.31** (0.84)
QUER	**.81** (1.00)	**.54** (0.96)	**.46** (0.87)	**.34** (0.86)
SHIP001	**1.02** (1.00)	**.64** (0.94)	**.51** (0.91)	**.51** (0.80)
X104	**2.20** (1.00)	**1.53** (0.95)	**1.11** (0.86)	**.91** (0.77)
MT1	**2.10** (1.00)	**1.28** (0.94)	**.91** (0.89)	**.82** (0.84)
INLINE	**13.89** (1.00)	**8.41** (0.98)	**4.59** (0.94)	**3.51** (0.90)
BMWCRA1	**5.96** (1.00)	**3.25** (0.96)	**2.05** (0.86)	**1.28** (0.83)
CRANKSEG1	**2.87** (1.00)	**1.67** (0.96)	**1.26** (0.88)	**.98** (0.78)
SHIPSEC8	**4.52** (1.00)	**3.09** (0.85)	**2.46** (0.82)	**2.33** (0.76)
CRANKSEG2	**4.15** (1.00)	**2.46** (0.96)	**1.51** (0.83)	**1.45** (0.71)
SHIPSEC5	**7.44** (1.00)	**4.68** (0.87)	**3.40** (0.78)	**3.05** (0.68)
SHIP003	**7.71** (1.00)	**5.22** (0.91)	**3.29** (0.82)	**3.13** (0.70)
THREAD	**3.91** (1.00)	**2.72** (0.81)	**2.53** (0.65)	**2.54** (0.59)
AUDI	**481.96** (1.00)	**270.16** (0.89)	**152.58** (0.83)	**86.07** (0.70)

processors, the factorization time is always reduced what shows that the MPI-thread implementation also improves the time scalability.

4 Concluding remarks

In this paper, we have proposed an hybrid MPI-thread implementation of a direct solver and have analyzed the benefits of this approach in terms of memory and run-time performances.

To validate this approach, we have performed experiments on IBM SP3 and SP4. In all test cases, the memory overhead is drastically reduced thanks to the hybrid MPI-thread implementation. In addition, we observe a significant decreasing of the factorization time and a better global scalability of the parallel solver. This techniques allow us to treat large 3D problems where the memory overhead was a bottleneck for the use of direct solvers. Recently, we have factorized a 3D problem from CEA with 10 millions of unknowns (NNZ=6.68e+9, OPC=4.29e+13) in less than 400 seconds on 2 SMP nodes (32-ways Power4+ with 64Go per node); for this case, the memory efficiency is about 0.95. We have only considered in this paper the factorization step, which is the most time consuming step, but the same improvements are also verified for the triangular solve step.

Finally, a comparison with WSMP [1] is still in progress and will be published in a survey article on the PaStiX direct solver.

References

1. Anshul Gupta, Mahesh Joshi, and V. Kumar. Wsmp: A high-performance shared- and distributed-memory parallel sparse linear equation solver. Report, University of Minnesota and IBM Thomas J. Watson Research Center, 2001.
2. 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.

3. 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.
4. 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.
5. P. Hénon, P. Ramet, and J. Roman. Efficient algorithms for direct resolution of large sparse system on clusters of SMP nodes. In *SIAM Conference on Applied Linear Algebra, Williamsburg, Virginie, USA*, July 2003.
6. 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.
7. X. S. Li. *Sparse Gaussian Elimination on High Performance Computers*. PhD thesis, University of California at Berkeley, 1996.
8. 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.