Dynamic scheduling for sparse direct solver on NUMA architectures

Mathieu Faverge and Pierre Ramet **

ScAlApplix Project, INRIA Bordeaux Sud-Ouest and LaBRI UMR 5800
Université Bordeaux 1, 33405 Talence cedex, France
{faverge,ramet}@labri.fr

Abstract. Over the past few years, parallel sparse direct solvers made significant progress and are now able to efficiently work on problems with several millions of equations. This paper presents some improvements on our sparse direct solver PaStiX[1] for distributed Non-Uniform Memory Access architectures. We show results on two preliminary works: a memory allocation scheme more adapted to these architectures and a better overlap of communication by computation. We also present a dynamic scheduler that takes care of memory affinity and data locality.
Keywords. sparse direct solver, NUMA architecture, dynamic scheduling, multi-cores

1 Introduction

Over the past few years, parallel sparse direct solvers made significant progress [1, 4, 6, 10]. They are now able to solve efficiently real-life three-dimensional problems with several millions of equations.

Nevertheless, the need of a large amount of memory is often a bottleneck in these methods. To control memory overhead, the authors presented in [8] a method which exploits the parallel blockwise algorithmic approach used in the framework of high performance sparse direct solvers in order to develop robust parallel incomplete factorization based preconditioners for iterative solvers. But for some applications, direct solvers remain generic and successful approach. Since the last decade, most of the supercomputer architectures are based on clusters of SMP nodes. In [7], the authors proposed a hybrid MPI-thread implementation of a direct solver that is well suited for SMP nodes or modern multi-core architectures. This technique allows treating large 3D problems where the memory overhead due to communication buffers was a bottleneck for the use of direct solvers. Thanks to this MPI-thread coupling, our direct solver PaStiX has been successfully used by the French CEA to solve a symmetric complex sparse linear system arising from a 3D electromagnetism code with more than

** This work is supported by the ANR grants 06-CIS-010 SOLTICE and 05-CIGC-002 NUMASIS (`http://solstice.gforge.inria.fr/` and `http://numasis.gforge.inria.fr/`)

[1] http://gforge.inria.fr/projects/pastix/

45 millions unknowns on the TERA-10 CEA supercomputer. Solving this system required about 1.4 Petaflops (in double precision) and the task was completed in about half an hour on 2048 processors. To our knowledge a system of this size and this kind has never been solved by a direct solver.

In the context of distributed NUMA architectures, a work has recently begun, in collaboration with the INRIA Runtime team, on studying optimization strategies, and improving the scheduling of communications, threads and I/O. Our solvers will use NewMadeleine and Marcel libraries [3, 12] in order to provide an experimental application to validate those strategies.

This work is based on two preliminary modifications. First, the main data structure has been improved to be more suitable for NUMA architectures and memory hierarchy. Second, computational and communication tasks are splitted to anticipate as much as possible the data receptions. Then, we will present major changes in the scheduling of our solver. Some numerical experiments are provided on two kinds of architectures for two test cases with about one million of unknowns.

2 Preliminary Works

2.1 NUMA-Aware Memory Allocation

Modern supercomputers are based on non-uniform memory access (NUMA) architectures. New multi-cores processors own several levels of cache with hierarchical access and supercomputers (especially with opteron barcelona processors) distribute memory over the nodes. This implies differences in access time and memory bandwidth following the geographical locality of dataset and threads which use it [9, 2]. That is why it is interesting to study the effect of allocation mechanism on our solver. PaStiX allocates the matrix in only one memory block for each cluster during the initialisation step. And for the factorization, threads get data from this dataset for their computations. The problem is that the system does not necessary returns memory blocks near cores where associated computations will be done since it has not this information. Usually, blocks are allocated close to the unit that requests memory. In this case, memory is allocated nearby the core which computes the initialisation step. All threads use the memory bus to access it, and the bandwidth limits the data transfer. In some cases, the system will spread memory blocks using a round-robin algorithm over the cluster. This method produces a less important memory bus overload, but each thread does not have the best access to its dataset. So, we choose to change our data structure to enable each thread to allocate its own data. Thus each dataset is allocated closed to the thread which need it. Moreover, each thread is bound to one core for all the computations, so we do not need to move datasets after they were allocated. Our experiments show that this method improves performances when architectures have important NUMA effects and do not create overload on SMP nodes.

2.2 Using threads dedicated to communications

The computation of each task is divided in two steps. In the first step, a thread waits for local and remote contributions needed before to compute the task. In the second step, local computations are performed (block-diagonal factorization, updates for blocks and contributions). During the second step, some local tasks are released and remote contributions are sent and some other are asked for next computations.

It is well known that overlapping communication by computation is an important problem for most MPI applications. But some idle time still remains in our static scheduling and the efficiency could be improved by adapting dynamically the communication scheme built during the preprocessing step. Moreover, most MPI implementations do not overlap communications properly. A non-overlapped rendezvous forces a computing thread to delay the data exchange until a call to *MPI_Wait*. To avoid this problem, we ensure communication progress thanks to one or more dedicated threads to manage data receptions (like in PIOMan implementation [13]).

3 Dynamic scheduling for NUMA architecture

In order to achieve efficient parallel sparse factorization, three preprocessing phases are commonly required :

- The *ordering* phase, which computes a symmetric permutation of the initial matrix A such that factorization will exhibit as much as concurrency as possible while incurring low fill-in.
- The *block symbolic factorization* phase, which determines the block data structure of the factored matrix L associated with the partition resulting from the ordering phase. This structure consists of N column blocks, each of them containing a dense symmetric diagonal block and a set of dense rectangular off-diagonal blocks. From this block structure of L, one can deduce the weighted elimination quotient graph that describes all dependencies between column blocks, as well as the super-nodal 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 resulting blocks onto the processors of the target architecture.

In this work, we focus on the last preprocessing phase, detailed in [5], that computes a scheduling used during the numerical factorization phase.

In a static scheduling, a BLAS2 and BLAS3 time model gives weights for each column block in the elimination tree. Then, a recursive top-down algorithm over the tree assigns a set of candidate processors to column blocks. Processors chosen to compute a node are assigned to its sons proportionally to their cost. It is possible to map a same processor on two different branches. An additional time model for communications is used to simulate the numerical factorization. A

task is mapped on one of its candidate processors using a greedy algorithm that distributes a task onto the processor that is able to compute it the soonest. Thus, for each processor, we obtain a vector of local tasks fully ordered by priority.

This static scheduling gives very good results on almost architectures. However we want to implement a dynamic scheduler at least as efficient as the static one. The objective is to reduce some observed idle times due to approximations in our time cost models especially when communications have to be estimated. The other main objective is to preserve memory affinity and locality particularly on nodes with a large number of cores. As seen in previous section, the effects of memory locality can be very important on these architectures. It would be difficult to take these effects into account in our static scheduling.

The algorithm we have implemented is expected to schedule dynamically the computations inside an SMP node but without modifying the distribution of the data. It is therefore necessary to compute, in a first step, a static distribution of data on SMP nodes, ie on each MPI process. To preserve memory affinity, a same processor is not allowed to be mapped on two different branches. Column blocks are splitted using a smaller blocksize to ensure to be able to use more threads than cores on architectures. In a second step, a set of candidate threads is assigned to each local task corresponding to a column block. A tree is then built where a queue is defined on each node; each queue contains the local tasks, ordered by priority, assigned to a same subset of threads.

At run time, we use the tree, denoted by T, of task queues q_n (where n is a node of the tree) with as many leaves as threads required to factorize the matrix. Threads get tasks from their own queue following the algorithm 1. If the queue is empty, the thread is allowed to get a task from the queues found in the path to the root of the tree. This corresponds to a work-stealing algorithm constrained by memory affinity criteria.

Algorithm 1: Main algorithm

```
/* Main loop for thread t                                          */
while some tasks remain in tree T do
    n = t ;
    while ((n is not root(T)) and (queueSize(q_n) is null)) do
        n = father(T, n) ;
    if n is root(T) then
        continue ;
    else
        i = queueGet(q_n) ;
    compute task i ;
    add ready tasks in corresponding queues ;
```

In practice, we observe that associate only one thread to leaves of tree is not enough to ensure activities to all the physical processors because upper-tasks could have to wait some contributions. We choose to associate a minimum of two threads in each branch and this provides good results as we can see in section 4. Moreover, if the number of thread is lower that the number of core, the threads are bound to a core in order to preserve memory affinity. But it can be interesting to increase the number of threads to fill in some remaining idle-time. In that case, threads can not be bound to processors and memory accesses are then not optimal. Therefore we plan to use the MARCEL "BubbleSched" framework [12] to group different threads and their datasets in a bubble and to bind this bubble on a part of the target architecture.

4 Numerical experiments

Architectures and test cases. In this section, we describe some experiments performed on two types of architectures. For NUMA experiments, we used two dual-core opteron architectures: HAGRID with 8 processors and BORDERLINE with 4 processors, each processor having 2 cores with 4GB of memory. Opteron architecture is represented on figure 1, HAGRID cluster corresponds to the whole scheme and BORDERLINE cluster corresponds only to the left part. Experiments on communications have been performed on a SMP cluster of IBM Power5 with 16 processors per node and connected by a "Federation" network which provides an *MPI_THREAD_MULTIPLE* implementation.

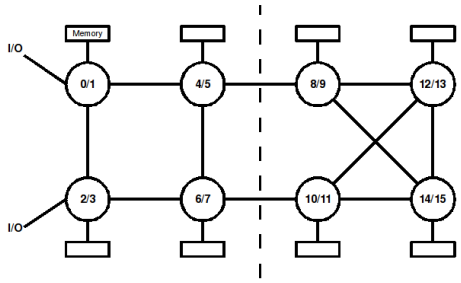

Fig. 1. Opteron Architecture

Name	Columns	NNZ_A	NNZ_L	Symmetric
AUDI	943 695	39 297 771	1.214e+09	Yes
MHD	485 597	24 233 141	1.629e+09	No

Table 1. Matrices used for our experiments

We consider two test cases (see table 1) where NNZ_A is the number of off-diagonal terms in the triangular part of the original matrix, NNZ_L is the number of off-diagonal terms in the complete factor. AUDI test case (structural mechanic problems from PARASOL collection) is a symmetric problem whereas MHD (Magneto-Hydro-Dynamic 3D problem) is an unsymmetric problem.

Threads	Allocation	Hagrid		Borderline	
		AUDI	MHD	AUDI	MHD
4	Global	**698**	**1270**	442	795
	Local	699	1300	**421**	**751**
8	Global	500	761	249	447
	Local	**363**	**662**	**217**	**408**
16	Global	386	507	-	-
	Local	**254**	**428**	-	-

Fig. 2. Impact of NUMA allocation (in seconds)

NUMA-aware allocation. In table 2, we can see that binding each thread to a core and allocating its datasets close to that core (`Local` line) improves the results on NUMA architectures compared to the global allocation strategy (`Global` line). The same behaviour is also observed on our SMP cluster in a less significant way.

Proc Number	Thread Number	AUDI			MHD		
		Initial	1 Thrd	2 Thrds	Initial	1 Thrd	2 Thrds
2	1	684	**670**	672	1120	**1090**	1100
	2	388	**352**	354	594	**556**	558
	4	195	**179**	180	299	**279**	280
	8	100	**91.9**	92.4	158	**147**	**147**
	16	60.4	**56.1**	**56.1**	113	88.3	**87.4**
4	1	381	**353**	**353**	596	**559**	568
	2	191	**179**	180	304	**283**	284
	4	102	**91.2**	94.2	161	**148**	150
	8	55.5	**48.3**	54.9	98.2	**81.2**	87.3
	16	33.7	**32.2**	32.5	59.3	56.6	**56**
8	1	195	**179**	183	316	**290**	300
	2	102	**90.7**	94	187	**153**	164
	4	56.4	**47.1**	50.7	93.7	**78.8**	101
	8	31.6	**27.6**	32.4	58.4	**50**	58.7
	16	21.7	**20.4**	32.3	49.3	**41.6**	43.5

Table 2. Impact of the number of dedicated threads for communications (in seconds)

Using threads dedicated to communications. In table 2, we compare the original version and a new implementation that uses 1 or 2 specific threads to enable communication to progress. The first column is the number of MPI processes and the second column is the number of threads dedicated to computations for each process. We can see that the overlapping with threads dedicated to communications is almost better than without such dedicated threads. However, we expected that the use of two threads should have been more efficient since the network interface includes two communication cards. This can be explained by the fact that the IBM MPI implementation already uses both cards, and by its performance issues with *MPI_THREAD_MULTIPLE* described in [11].

Proc Number	Thread Number	AUDI		MHD	
		Static	Dynamic	Static	Dynamic
1	16	99.6	107	156	156
	32	**96.3**	97.8	**149**	**149**
	64	99.2	101	151	152
2	16	56.1	56.8	88.3	87
	32	55.6	**54.5**	92.5	**83.8**
	64	59	55.3	89.9	94.3
4	16	**32.2**	36.6	56.6	**54.6**
	32	33.2	35.3	70.6	68.7
	64	43.3	39.6	72	62.3

Table 3. Comparison between the static and the dynamic scheduler on SMP cluster (in seconds)

Dynamic scheduling. Static and dynamic schedulers are compared in table 3, both use one thread dedicated to communications and NUMA allocation is enabled. Even if our SMP cluster presents a limited NUMA effect, the dynamic scheduler allows some improvements on the execution time, especially for the unsymmetric case. Since the volume and the number of communications are more important than for the symmetric case, there are potentially more idle times to reduce.

Figures 3 and 4 show the Gantt diagram for the factorization of the MHD test case with 4 MPI processes and 4 threads per MPI process. The first figure corresponds to an execution with a NUMA-aware memory allocation and for the second figure we use the dynamic scheduler and one thread is dedicated to data receptions per MPI process. For both figures, the reception date for a communication is recorded only when the contribution is taken into account in the algorithm and not exactly when the message is received by the MPI process. A task is drowned with a colour that corresponds to its level in the elimination tree. A dark colour is used to represent idle time. We can observe some permutations (alternated colours) in the computation task order to fill in some idle-time. In this case, the number of permutation (ie. the number of

time a thread needs to take a task in bubble at a higher level) is average 20 permutations per node.

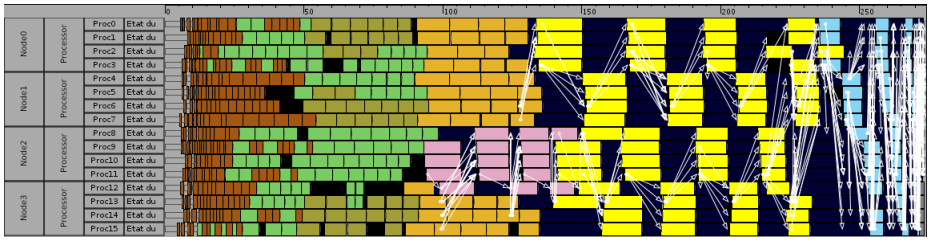

Fig. 3. Static scheduling on quad dual-core opteron architecture (MHD test case).

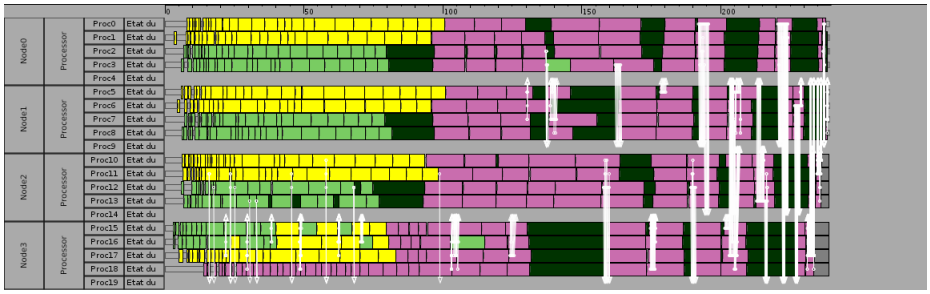

Fig. 4. Dynamic scheduling on quad dual-core opteron architecture (MHD test case).

5 Conclusion and future works

The results are encouraging since we already improved the execution time on a SMP cluster. We have to validate the approach on parallel architectures that present significant NUMA effects. We now plan to use the MARCEL bubble scheduler to improve results on NUMA architectures. Different threads and their datasets will be grouped in a bubble and bound to a part of the target architecture. The static scheduling of communications will also need to be adapted. In an Out-of-Core context, new problems linked to the scheduling and the management of the computational tasks may arise since processors may be slowed down by I/O operations. Thus, we will have to design and study specific algorithms for this particular context by extending our work on scheduling for heterogeneous platforms.

References

1. P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. 23(1):15–41, 2001.
2. Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Exploring thread and memory placement on NUMA architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In *HiPC*, pages 338–352, 2006.
3. Olivier Aumage, Elisabeth Brunet, Nathalie Furmento, and Raymond Namyst. NewMadeleine: a fast communication scheduling engine for high performance networks. In *CAC 2007 held in conjunction with IPDPS*, March 2007.
4. Anshul Gupta. Recent progress in general sparse direct solvers. In *Lecture Notes in Computer Science*, volume 2073, pages 823–840, 2001.
5. 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, Cancun, Mexique*, volume 1800 of *Lecture Notes in Computer Science*, pages 519–525. Springer, May 2000.
6. 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.
7. P. Hénon, P. Ramet, and J. Roman. On using an hybrid MPI-Thread programming for the implementation of a parallel sparse direct solver on a network of SMP nodes. In *Proceedings of PPAM'05, Poznan, Pologne*, volume 3911 of *Lecture Notes in Computer Science*, pages 1050–1057, September 2005.
8. P. Hénon, P. Ramet, and J. Roman. On finding approximate supernodes for an efficient ILU(k) factorization. *Parallel Computing*, 2008. accepted.
9. Kevin Barker and Kei Davis and Adolfy Hoisie and Darren Kerbyson and Michael Lang and Scott Pakin and José Carlos Sancho. Experiences in Scaling Scientific Applications on Current-generation Quad-core Processors. In *proceedings of LSPP'08 held in conjunction with IPDPS*, Miami, Florida, USA, April 2008.
10. Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.
11. Rajeev Thakur and William Gropp. Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE. In *PVM/MPI*, pages 46–55, 2007.
12. Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework. In *EuroPar'07*, Rennes, France, August 2007.
13. Francois Trahay, Alexandre Denis, Olivier Aumage, and Raymond Namyst. Improving reactivity and communication overlap in MPI using a generic I/O manager. In *EuroPVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 170–177. Springer, 2007.