HAL
open science

A NUMA Aware Scheduler for a Parallel Sparse Direct Solver

Mathieu Faverge, Xavier Lacoste, Pierre Ramet

▶ **To cite this version:**

HAL Id: inria-00549827
https://inria.hal.science/inria-00549827

Submitted on 22 Dec 2010

INRIA

A NUMA Aware Scheduler for a Parallel Sparse Direct Solver

Mathieu Faverge — Xavier Lacoste — Pierre Ramet

N° 7498

Mai 2010

Stochastic Methods and Models

Rapport de recherche

A NUMA Aware Scheduler for a Parallel Sparse Direct Solver

Mathieu Faverge , Xavier Lacoste , Pierre Ramet

Abstract: Over the past few years, parallel sparse direct solvers made significant progress and are now able to solve efficiently industrial three-dimensional problems with several millions of unknowns. To solve efficiently these problems, PaStiX and WSMP solvers for example, provide an hybrid MPI-thread implementation well suited for SMP nodes or multi-core architectures. It enables to drastically reduce the memory overhead of the factorization and improve the scalability of the algorithms. However, today's modern architectures introduce new hierarchical memory accesses that are not handle in these solvers. We present in this paper three improvements on PaStiX solver to improve the performance on modern architectures : memory allocation, communication overlap and dynamic scheduling and some results on numerical test cases will be presented to prove the efficiency of the approach on NUMA architectures.

Key-words: sparse direct solver, NUMA architecture, multi-cores, dynamic scheduling

Un ordonnanceur pour solveur direct creux parallèle adapté aux architectures NUMA

Résumé : Pas de résumé

1 Introduction

Over the past few years, parallel sparse direct solvers have 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. The goal was to develop a robust and parallel incomplete factorization based on preconditioners for iterative solvers. But for some applications, direct solvers remain a generic and successful approach. Since the last decade, most of the supercomputer architectures are based on clusters of SMP (Symmetric Multi-Processor) 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 to treat large 3D problems where the memory overhead due to communication buffers was a bottleneck to the use of direct solvers. Thanks to this MPI-thread coupling, our direct solver PaStiX has been successfully used by the French Nuclear Agency (CEA) to solve a symmetric complex sparse linear system arising from a 3D electromagnetism code with more than 83 millions unknowns on the TERA-10 CEA supercomputer. Solving this system required about 5 Petaflops (in double precision) and the task was completed in about 5 hours on 768 processors. To our knowledge, a system of this size and this kind has never been solved by a direct solver.

New NUMA (Non Uniform Memory Access) architectures have an important effect on memory access costs, and introduce contentions problems which do not exist on SMP nodes. Thus, the main data structure of our targeted application have been modified to be more suitable for NUMA architectures. A second modification, relying on overlapping opportunities, allows us to split computational or communication tasks and to anticipate as much as possible the data receptions. We also introduce a simple way of dynamically schedule an application based on a dependency tree while taking into account NUMA effects. Results obtained with these modifications are illustrated by showing performances of the PaStiX solver on different platforms and matrices.

After a short description of architectures and matrices used for numerical experiments, we study data allocation and placement taking into account NUMA effects. Section 4 focuses on the improvement of the communication overlap as a preliminary work for a dynamic scheduler. Finally, a dynamic scheduler is described and evaluated on the PaStiX solver for various test cases.

2 Experimental platforms and test cases

Two NUMA (Non Uniform Memory Access) and one SMP (Symmetric Multi-Processor) platforms have been used in this work :

- The first NUMA platform, see Figure 1(a), denoted as "NUMA8" in the remaining of this paper, is a cluster of ten nodes interconnected by an Infiniband network. Each node is made of four Dual-Core AMD Opteron(tm) processors interconnected by HyperTransport. The system memory amounts to 4GB per core giving a total of 32GB.
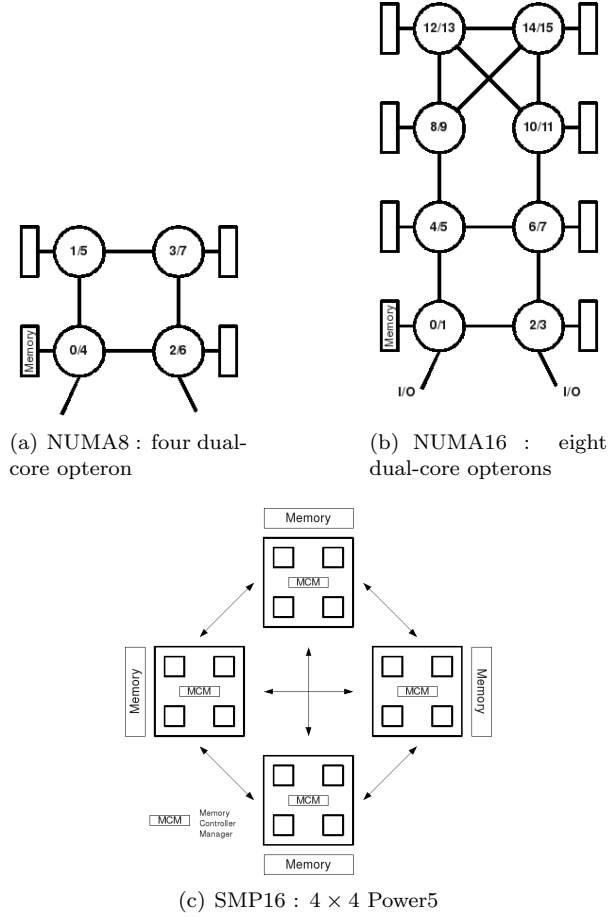
(a) NUMA8 : four dual-core opteron

(b) NUMA16 : eight dual-core opterons

(c) SMP16 : 4×4 Power5

Figure 1: Architectures used for benchmarks

- The second NUMA architecture, see Figure 1(b), called "NUMA16", is a single node of eight Dual-Core AMD Opteron(tm) processors with 64GB of memory.

- The SMP platform is a cluster of six IBM p575 nodes, see Figure 1(c), called "SMP16". These nodes are interconnected by an IBM Federation network. Four blocks of four Power5 with 8GB are installed in each node. However, the memory available is limited to 28GB by the system.

Tests on NUMA effects have been performed on the three platforms and tests on communications have been performed on the IBM Federation network and on the Infiniband network with respectively the IBM MPI implementation and Mvapich2 that support the `MPI_THREAD_MULTIPLE` threading model.

Our research targeted the parallel sparse direct solver PaStiX which uses a hybrid MPI-Threads implementation. Improvements made on the solver have been tested on a set of eight matrices described in the Table 1. All matrices are double precision real matrices with the exception of the last one (*HALTERE*)

Name	Columns	NNZ_A	NNZ_L	OPC
MATR5	485 597	24 233 141	1 361 345 320	9.84422e+12
AUDI	943 695	39 297 771	1 144 414 764	5.25815e+12
NICE20	715 923	28 066 527	1 050 576 453	5.19123e+12
INLINE	503 712	18 660 027	158 830 261	1.41273e+11
NICE25	140 662	2 914 634	51 133 109	5.26701e+10
MCHLNF	49 800	4 136 484	45 708 190	4.79105e+10
THREAD	29 736	2 249 892	25 370 568	4.45729e+10
HALTERE	1 288 825	10 476 775	405 822 545	7.62074e+11

NNZ_A is the number of off-diagonal terms in the triangular part of the 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.

Table 1: Matrices used for experiments

which is a double precision complex matrix. *MATR5* is an unsymmetric matrix (with a symmetric pattern).

3 NUMA-aware allocation

Modern multi-processing architectures are commonly based on shared memory systems with a NUMA behavior. These computers are composed of several chip-sets including one or several cores associated to a memory bank. The chipset are linked together with a cache-coherent interconnection system. Such an architecture implies hierarchical memory access times from a given core to the different memory banks. This architecture also possibly incurs different bandwidths following the respective location of a given core and the location of the data sets that this core is using [2, 9]. It is thus important on such platforms to take these processor/memory locality effects into account when allocating resources. Modern operating systems commonly provide some API dedicated to NUMA architectures which allow programmers to control where threads are executed and memory is allocated. These interfaces have been used in the following part to exhibit NUMA effects on our three architectures. The first one studies different placement combinations of threads and memory. The second one studies a more realistic case where all cores are simultaneously used. Finally, we present some results on our solver where such programming interface is used.

3.1 Data placement on NUMA architectures

To measure the NUMA effects of our architectures on BLAS routines, we use a test program which makes 2500 calls to `daxpy` and `dgemm` routines on a set of vectors and matrices: their sizes are respectively 128 and 128x128. Vectors and matrices used by BLAS calls are allocated on one core while the calling thread is bound to another core thanks to `sched_affinity` and `thread_policy_set` functions (for respectively LINUX and AIX systems) . The memory allocation scheme follows the first touch rule of the system.

The following results have been obtained on the quad-dual-core opteron architecture NUMA8. The time recorded for each combination of a thread location

accessing its data at a specific memory location are given in Table 2 for BLAS1 tests and in Table 3 for BLAS3 tests. The results are normalized by the computing time of the favorable case where the computation thread and the memory are located on the same core.

		Computational thread location							
		0	1	2	3	4	5	6	7
	0	**1.00**	1.34	1.31	*1.57*	**1.00**	1.34	1.31	*1.57*
	1	1.33	**1.00**	*1.57*	1.31	1.33	**1.00**	*1.57*	1.31
	2	1.28	*1.57*	**1.00**	1.33	1.29	*1.57*	**1.00**	1.32
Data location	3	*1.56*	1.32	1.34	**1.00**	*1.56*	1.31	1.33	**0.99**
	4	**1.00**	1.34	1.31	*1.57*	**1.00**	1.34	1.30	*1.58*
	5	1.33	**1.00**	*1.58*	1.30	1.33	**1.00**	*1.57*	1.30
	6	1.29	*1.57*	**1.00**	1.33	1.29	*1.57*	**1.00**	1.32
	7	*1.57*	1.32	1.33	**1.00**	*1.57*	1.32	1.35	**1.00**

Table 2: Influence of data placement on `dAXPY` on NUMA8

		Computational thread location							
		0	1	2	3	4	5	6	7
	0	**1.00**	1.04	1.04	*1.07*	**1.00**	1.04	1.04	*1.07*
	1	1.04	**1.00**	*1.07*	1.04	1.04	**1.00**	*1.08*	1.04
	2	1.04	*1.07*	**1.00**	1.04	1.04	*1.07*	**1.00**	1.04
Data location	3	*1.08*	1.04	1.04	**1.00**	*1.07*	1.04	1.05	**1.00**
	4	**1.00**	1.04	1.04	*1.07*	**1.00**	1.04	1.04	*1.07*
	5	1.05	**1.00**	*1.08*	1.04	1.05	**1.00**	*1.08*	1.04
	6	1.04	*1.07*	**1.00**	1.05	1.04	*1.07*	**1.00**	1.04
	7	*1.08*	1.04	1.05	**1.00**	*1.08*	1.04	1.05	**1.00**

Table 3: Influence of data placement on `dGEMM` on NUMA8

BLAS1 computations in Table 2 are limited by the bandwidth of memory bus and allow to deduce NUMA coefficients for this machine. The NUMA factor identified is about 1.3 for a single jump and 1.58 for two jumps on interconnection link. The results also confirm the presence of a shared memory bank for each chip of two cores as described on Figure 1(a). These coefficients depend on the architecture. For example, on the octo dual-cores opteron NUMA16, NUMA factor varies between 1.1 and 2.1 for one to four jumps. On the contrary, for the Power5 architecture SMP16, computation times are mostly independent from the dataset location.

It is therefore important to take into account data-sets location. It is especially true for computations which do not reuse data as BLAS routines of level 1 and where data transfer is the bottleneck. In this case, effects can quickly become expensive. However, HPC applications concentrate most calculations in BLAS routines of level 3. One can observe on Table 3 that there is always a significant factor which depends on the data distance. Thus, on the NUMA8 architecture, there is a coefficient of 1.04 for a single jump and 1.08 for 2 jumps. This factor increases up to 1.20 on architecture NUMA16 for 4 jumps. As expected, the memory location does not influence the computation times on the

SMP16 architecture.

In summary, this study highlights the need to take into account possible NUMA effects during the memory allocation in threaded applications. Applications usually have a sequential initialization step that allocates and fills data-sets. The main point here is to delay these operations to the threads used for computations. Each thread needs to allocate and fills its own memory to benefit from the default first touch algorithm. However, this does not reflect reality because usually one wishes to use all the resources and not just one core per node. Thus, the next part will study NUMA effects with contention problems on a completely loaded computer.

3.2 Contention on NUMA architectures

This second study points out contention effects on NUMA architectures. Our test program creates as many threads as computational cores available on the node. Each thread is bound to a dedicated core and computes a set of 2500 vectors additions or matrices multiplications, as in the previous study. The objective of these experiments is to measure the effect of contention on the computations (see Figure 2). For each case, three methods are used to allocate memory :

- *On first core* : all memory is allocated by the thread bound on the first core.

- *Good location* : each thread allocates its own data-sets.

- *Worst location* : memory is allocated with the worse location for each thread following the previous study. In practice, each thread t allocates data-sets for the thread $n - t$, where n is the number of cores for the architecture.

The last curve *(No contention)* represents the computational time for each core in the best case of the previous study (computations and data-sets on the same core) when there is no contention.

The results focus on the average execution time on each core of `daxpy` and `dgemm` functions on the three architectures. The curves are the average time of computations on all cores. All experiments are normalized with respect to the best case where each thread allocates and initializes its own memory. All results exhibit NUMA factors, with even bigger values for cases with contention. A factor of 1.7 to 2.8 can be observed on the NUMA8 architecture on BLAS routines of level 1, and from 1 to 2.3 on the NUMA16 architecture. On NUMA16 architecture, execution times are less regular over the different cores than on NUMA8 architecture, in the case of the worst placement. The system seems to give priority to threads close to the memory, rather than give an uniform access times to each one. Typically, an application designed with an initialization step that allocates memory before launching computational threads may suffer a great penalty on the execution time. In the same way, if the programmer relies on the system to allocate memory with a round-robin policy, the overhead could be significant.
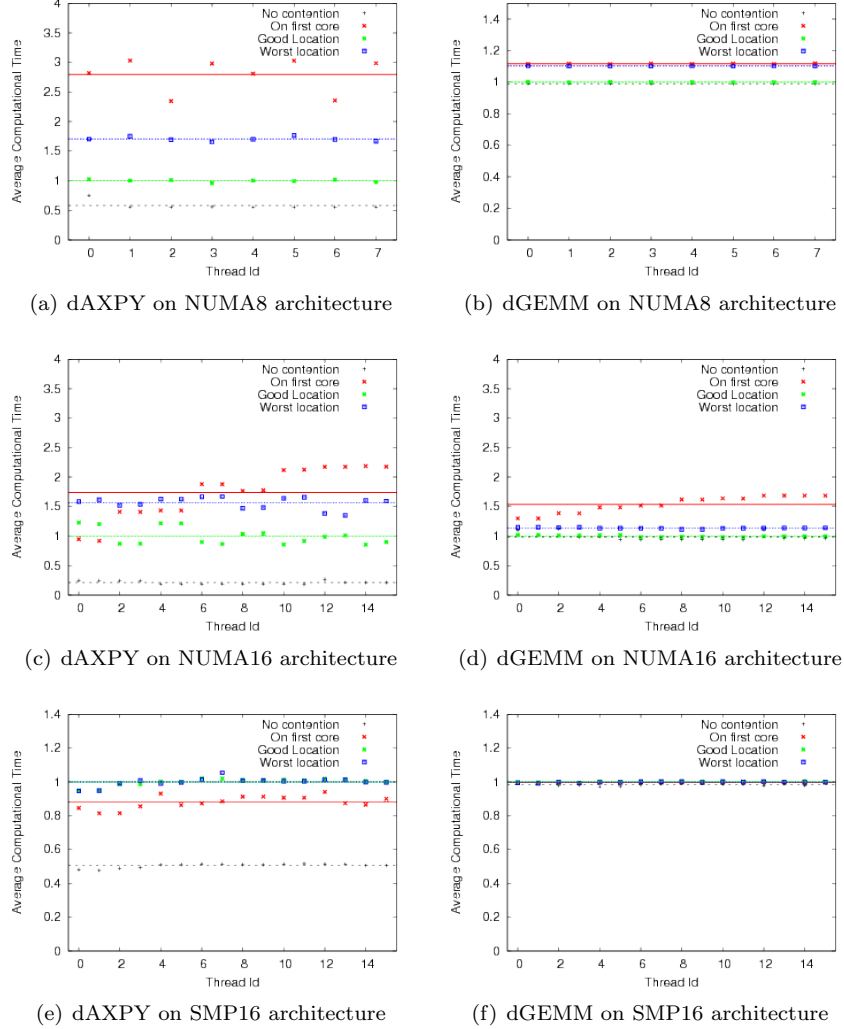
(a) dAXPY on NUMA8 architecture   (b) dGEMM on NUMA8 architecture

(c) dAXPY on NUMA16 architecture   (d) dGEMM on NUMA16 architecture

(e) dAXPY on SMP16 architecture   (f) dGEMM on SMP16 architecture

Figure 2: Influence of contention on three platforms

For BLAS routines of level 3, the gain is less significant since cache effet hides a large part of the data access. As expected, the tests on SMP architecture give homogeneous access to memory.

3.3   Results on PaStiX solver

The hybrid MPI/thread version of the PaStiX solver already uses a function to bind threads on cores for different systems (LINUX, AIX, MacOS, . . .). However, in its initial version, PaStiX does not take into account NUMA effects in memory allocation. The initialization step allocates all structures needed by computations and especially the part of the matrix computed on the node, and fills it with local coefficients provided by the user. After this step, as many threads as cores are created to compute the numerical factorization (eight

(a) Localization of new NUMA-aware allocation in the matrix

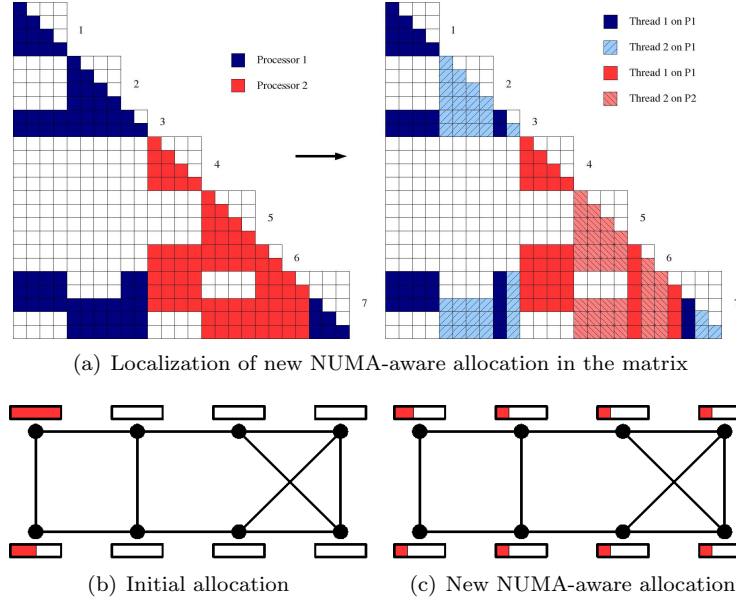(b) Initial allocation   (c) New NUMA-aware allocation

Figure 3: NUMA-aware allocation

threads for NUMA8 platform and sixteen threads for NUMA16 and SMP16 clusters). The problem is therefore that all data-sets are allocated close to the core where the initialization steps has occurred. Memory allocation is not evenly spread on the node and access times are thus not optimal (see Figure 3(b)).

In the new version of PASTIX, data structures have been modified to allow each thread to allocate and to fill its part of the matrix as shown in Figure 3(a). This example shows the allocation repartition on each process and on each thread of each process. The memory is better spread over the nodes as shown in Figure 3(c) and thus allows to obtain the best memory access as seen previously. Moreover, thanks to the method used to predict the static scheduling [5, 6], access to remote data is restrained, as well as in the results presented in [11].

Matrix	NUMA8		NUMA16		SMP16	
	Global	Local	Global	Local	Global	Local
MATR5	437	410	527	341	162	161
AUDI	256	217	243	185	101	100
NICE20	227	204	204	168	91.40	91
INLINE	9.70	7.31	20.90	15.80	5.80	5.63
NICE25	3.28	2.62	6.28	4.99	2.07	1.97
MCHLNF	3.13	2.41	5.31	3.27	1.96	1.88
THREAD	2.48	2.16	4.38	2.17	1.18	1.15
HALTERE	134	136	103	93	48.40	47.90

Table 4: Influence of NUMA-aware allocation on numerical factorization of PASTIX solver (in seconds)

Table 4 highlights the influence of NUMA-aware allocation on the factorization time on the different platforms. The Global columns are the times for

the numerical factorization in the initial version of PaStiX with a global allocation performed during the initialization step. The `Local` columns are the factorization times with the new NUMA-aware allocation: the columns-blocks are allocated locally by the thread which factorizes them. A gain of 5% to 15% is observed on the NUMA8 architecture on all the matrices (except for the complex test case where there is no improvement). The gains on the NUMA16 platform are of 10% to 35% even with the complex test case. And, as expected, the SMP16 architecture shows no meaningful improvement.

Finally, results on a high performance application confirm the outcome of the benchmarks realized previously about the importance of taking into account the locality of memory on NUMA architectures. This could indeed significantly improves the execution time of algorithms with potentially huge memory requirements. And as shown by the last results, these effects increase with the size of the platforms used.

4 Communication Overlap

In large distributed applications, communications are often a critical limit to the scalability and programmers try to overlap them by computations. Often, this consists in using asynchronous communications to give the MPI implementation the opportunity to transfer data on the network while the application performs computations. Unfortunately, not all implementations make the asynchronous communications progress efficiently. It is especially true when messages reach the *rendezvous* threshold. A non-overlapped *rendezvous* forces a computing thread to delay the data exchange until a call to *MPI_Wait* or *MPI_Test*.

To avoid this problem, we try to let communications progress thanks to one dedicated thread, following the *tasklet* mechanism used for instance in the PIOMan library implementation [14]. Moreover, in the perspective of a dynamic scheduling, it is important to receive data as soon as possible to release new tasks and provide more possibilities to choose the next task to compute.

The Gantt diagrams, presented at the end of the paper highlights these results (see Figures 6(b) and 6(a)). In the initial version, each thread manages its own communications using asynchronous mode. The time for a communication (white arrow) is significantly decreased thanks to a dedicated thread compared to the diagram of the initial version. A substantial overlap, obtained with the dedicated communication thread method, allows to reach better performances on factorization time as we can see on Table 7 (between the versions V0 and V1).

Another interesting question is to find how many threads should be dedicated to communication when several network cards are available in the platform nodes. For instance, two network cards per node are available in the SMP16 clusters. We study the impact of adding one more thread dedicated to communications on the PaStiX solver (see Table 5).

The results show that even with a small number of computation threads, having two threads dedicated to communication is not useful. The performances are quite similar for small numbers of MPI process but decrease significantly for 8 MPI process. We can conclude that this MPI implementation already makes a good use of the two network cards. Adding one more thread to stress MPI is

Nb.	Nb.	AUDI			MATR5		
Node	Thread	Initial	1 ThCcom	2 ThCom	Initial	1 ThCom	2 ThCom
	1	684	**670**	672	1120	**1090**	1100
	2	388	**352**	354	594	**556**	558
2	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**
	1	381	**353**	**353**	596	**559**	568
	2	191	**179**	180	304	**283**	284
4	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**
	1	195	**179**	183	316	**290**	300
	2	102	**90.7**	94	187	**153**	164
8	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 5: Impact of dedicated threads for communications on numerical factorization (in seconds) on SMP16

not a good solution. This result is not surprising since the article [12] highlights the bad performance of the IBM MPI implementation in multi-threaded mode compared to the single thread mode.

In the remaining of this paper, we choose to dedicate a single thread to manage communications.

5 Dynamic scheduling for NUMA architecture

We now present our works on the conception and on the implementation of a dynamic scheduler for applications which have a tree shaped dependency graph, such as sparse direct solvers, which are based on an elimination tree (see Figure 4(a)). In the following of this paper, Tnode will denote a node of the elimination tree, and we will call Cnode a node of the cluster architecture used for computations. Each Tnode corresponds to a column block to factorize and needs the updates from its descendants. In the case of a right-looking version of the standard Cholesky algorithm, it is possible to compute a dynamic scheduling on a such structure. When a Tnode of the tree is computed, it sends its contributions to different Tnodes upper in the tree and is able to know if a Tnode is ready to be computed or not. Hence, a simple list of ready tasks is the only data structure we need to design a dynamic algorithm.

However, Section 3 highlighted the problem of NUMA effects on memory allocation. The main problem here is to find a way to preserve memory affinity between the threads that look for a task and the location of the associated data during the dynamic scheduling. The advantage of such an elimination tree is that contributions stay close in the path to the root. Thus, to preserve memory affinity, we need to assign to each thread a contiguous part of the tree and lead

a work stealing algorithm which could takes into account NUMA effects.

Finally, there are two steps in proposed solution. The first one distributes efficiently data among all the Cnodes of the cluster. The second one builds a NUMA-aware dynamic scheduling on the local tasks mapped on each Cnodes.

We will now focus on the direct sparse linear solver PaStiX, which is our target application for this work. We summarize some details about the implementation of this solver and its static scheduling. Then we present the changes to develop a dynamic scheduling.

5.1 Sparse direct solver and static scheduling

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

- 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.

- 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 in 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 of the PaStiX solver, detailed in [5], that computes a scheduling used during the numerical factorization phase. To build a static scheduling, first, a proportionnal mapping algorithm is applied on the processors of the target architecture. A BLAS2 and/or 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 each Tnode (see Figure 4(a)). Processors chosen to compute a column block are assigned to its sons proportionally to the cost of each son and to the computations already affected to each candidate. It is possible to map the same processor on two different branches to balance the computations on all available resources. The last step corresponds to the data distribution over the Cnodes. A simulation of the numerical factorization is performed thanks to an additional time model for communications. Thus, all tasks are mapped on one of its candidates processors. Beforehand, tasks are sorted by priority based on the cost of the critical path in the elimination tree. Then a greedy algorithm distributes tasks onto the candidates able to compute it the soonest. We obtain a vector of local tasks fully ordered by priority for each processor.

This static scheduling gives very good results on most platforms. However we want to implement a dynamic scheduler at least as efficient as the static one,

(a) First proportional mapping
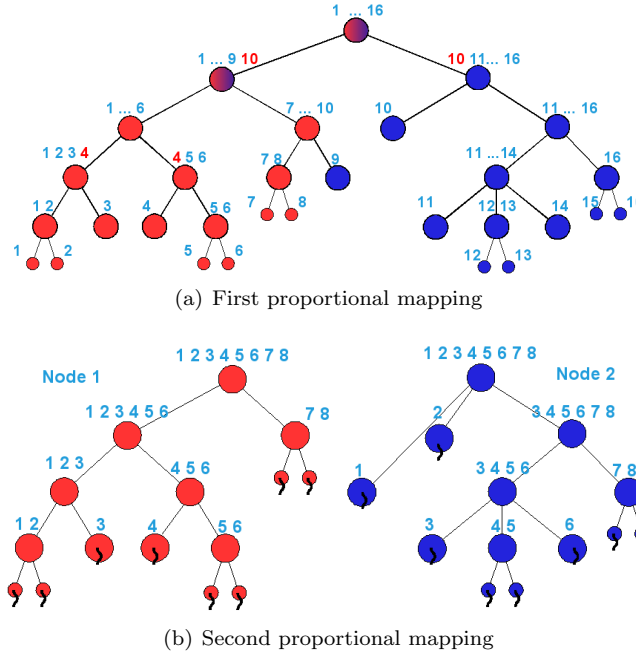

(b) Second proportional mapping

Figure 4: Proportional mapping in elimination tree for static and dynamic scheduler.

more suitable for NUMA or heterogeneous architectures. The objective is to reduce some observed idle times due to approximations in our time cost models, especially when communications have to be estimated. This can be highlighted on the Figure 6(a). The other main objective is to preserve memory affinity and locality particularly on architectures with a large number of cores. Our static scheduling can be naturally adapted since all threads are bound on a processor and data associated to the distributed tasks are allocated close to it.

5.2 Dynamic scheduling over an elimination tree

The dynamic scheduling expected has to dispatch tasks over the available threads on a same Cnode but does not have to re-assign tasks between them. The first step of our new algorithm is thus the same as in the static one: apply a proportional mapping of the elimination tree over the Cnodes and simulate the numerical factorization to distribute data. The simulation is based on the same cost models with all the processors or cores available in the system.

Once we have apply the first proportional mapping, each Cnode owns a set of subtrees of the initial elimination tree as in Figure 4(b). These subtrees are refined with a smaller block size to obtain fine grain parallelism. Then, a second proportional mapping is done on them based with the local number of available processors. In that case, we do not allow a thread to be candidate for two different subtrees to ensure memory affinity between tasks affected to each one. This provides a tree with a list of fully ordered tasks where a set of candidate threads is mapped on each node as shown in Figure 5.
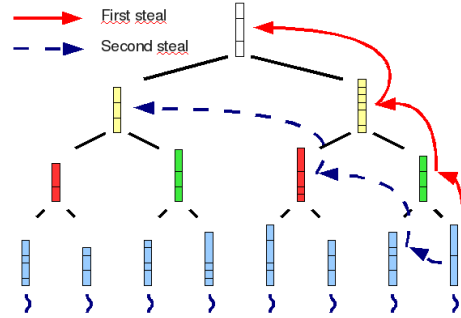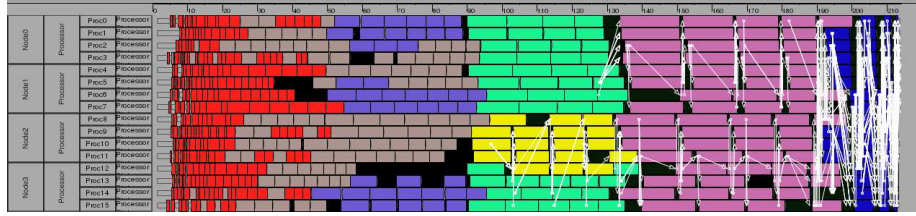
Figure 5: Work stealing algorithm

In the following, the nodes of the tree used for the work stealing algorithm will be called Snodes. A set of candidate threads assigned to each Snode is a subset of the candidate threads assigned to the father of this Snode.

This tree, denoted by T, of task queues q_n (where n is a Snode) has as many leaves as threads required to factorize the matrix. At runtime, as well as in the static version, threads are bound on one dedicated core in the order of a breadth-first search to ensure that closer threads will be in the same sets. During the numerical factorization, the threads will have to compute mainly the tasks which belongs to Snodes from their critical path on T. Thus, tasks in the queues q_n associated with the leaves of T are allocated by the only thread that is allow to compute them. Memory affinity is then preserved for the main part of the column blocks. However, we also have to allocate data associated with remaining tasks in Snodes which are not leaves. A set of threads are able to compute them. We choose, in the current implementation, to use a round-robin algorithm on the set of candidates to allocate those column blocks. Data allocation is then not optimal, but, if the cores are numbered correctly, this allocation reflects the physical mapping of cores inside a Cnode. It is important to notice

(a) Static scheduling

(b) Dynamic scheduling using a two ways of stealing method

Figure 6: Gantt diagrams for the *MATR5* test case on NUMA8 with 4 MPI process of 4 threads. Idle time is represented by black blocks and communications by white arrows.

that two cores with two successive id are not always closed on the architecture.

The Gantt diagrams in Figures 6 correspond to the execution for the *MATR5* test case (see Table 1) on the NUMA8 architecture. Jobs are executed on 4 MPI process of 4 threads and the diagram focuses on the activity of each core (one thread per line). The second diagram presents one more line per node corresponding to the communication thread. Each color corresponds to a level in the elimination tree (ie one color corresponds to one set of candidate processors) and black blocks highlight idle-times.

Once a thread t has no more jobs in its set of ready tasks q_t, it steals jobs in the queues of its critical path as described by the filled red arrows in Figure 5. Thus, we ensure that each thread works only on a subtree of the elimination tree and on some column blocks of its critical path. This ensures a good memory affinity and improves performances in the upper levels of the tree T especially when several threads are available.

However, there still remain idle times during the execution of the lower part of the tree T (mainly due to approximations of our cost models). Performances are improved using a two ways of stealing method (see Figure 6(b)). Once a thread has no more ready tasks in its critical path, it tries to steal a task in the sons of the Snodes that belong to its critical path as described by the dotted blue arrows on the Figure 5.

5.3   Results on PaStiX solver

This section resumes the results obtained with the improved algorithms implemented in the PaStiX solver and especially about the dynamic scheduler. The

Table 6 highlights the improvements on the solver inside a single Cnode. All test cases are run using eight threads for NUMA8 platform and sixteen threads for NUMA16 and SMP16 clusters. The column V0 (respectively V1) presents the factorization time obtained with the initial version without the NUMA-aware allocation (respectively with the NUMA-aware allocation). The third column gives the results with the NUMA-aware allocation and with the dynamic scheduler using the two ways of stealing method.

    Firstly, we observe that results are globally improved for all the test cases and for all the architectures when the dynamic scheduler is enabled. However, in few cases, factorization time obtained with the dynamic scheduling can be less efficient than with the static one. This is mainly due to the round-robin algorithm used to allocate data in the upper levels of the elimination tree. Secondly, the results presented on the SMP16 platforms show that the dynamic scheduler can improve performances and thus, confirm that problems on NUMA architecture are mainly due to weakness in memory location.

Matrix	NUMA8			NUMA16			SMP16		
	V0	V1	V2	V0	V1	V2	V0	V1	V2
MATR5	437	410	**389**	527	341	**321**	162	161	**150**
AUDI	256	217	**210**	243	185	**176**	101	100	**100**
NICE20	227	**204**	227	204	168	**162**	91.40	91	**90.30**
INLINE	9.70	**7.31**	7.32	20.90	15.80	**14.20**	5.80	**5.63**	5.87
NICE25	3.28	**2.62**	2.82	6.28	**4.99**	5.25	2.07	1.97	**1.90**
MCHLNF	3.13	**2.41**	2.42	5.31	3.27	**2.90**	1.96	1.88	**1.75**
THREAD	2.48	2.16	**2.05**	4.38	2.17	**2.03**	1.18	1.15	**1.06**
HALTERE	134	136	**129**	103	**93**	94.80	48.40	47.90	**47.40**

Table 6: Comparaison of numerical factorization time in seconds on three versions of PaStiX solver. V0 is the initial version with static scheduling and without NUMA-aware allocation. V1 is the version with NUMA-aware allocation and static scheduling. V2 is the version with NUMA-aware allocation and dynamic scheduling.

    The Table 7 presents results of the dynamic scheduler with multiple MPI process on NUMA8 and SMP16 platforms. All version have enable the NUMA-aware allocation. Once again, all test cases are run using, for each MPI process, eight threads for NUMA8 platform and sixteen threads for SMP16 clusters. The first version V0 does not use a dedicated thread for communications contrary to the two others versions, and the third version V2 uses the dynamic scheduler.

Nb. Node	NUMA8						SMP16					
	AUDI			MATR5			AUDI			MATR5		
	V0	V1	V2	V0	V1	V2	V0	V1	V2	V0	V1	V2
1	217	-	**210**	410	-	**389**	100	-	**100**	161	-	**150**
2	142	111	**111**	212	208	**200**	60.4	**56.1**	56.8	113	88.3	**87**
4	69	60.5	**57.7**	171	121	**114**	33.7	**32.2**	32.6	59.3	56.6	**54.6**
8	45.3	37.2	**35.6**	117	82.7	**78.8**						

Table 7: Comparaison of numerical factorization time in seconds of PaStiX solver with several MPI process. The three versions (V0, V1 and V2) have the NUMA-aware allocation enabled. V0 uses the initial communication model with static scheduling. V1 uses one thread dedicated to communications with static scheduling. V2 uses one thread dedicated to communications with dynamic scheduling.

As seen in section 4, using a thread dedicated to communications improves performances and using the dynamic scheduler still reduce the factorization time. The first improvement allows a better communications/computations overlap and the second improvement allows better reactivity to exploit incoming contributions from other MPI process. Results are more significant with the unsymmetric matrix *MATR5*, that generates more communications, than with symmetric matrices.

The improvements are mainly due to communications overlap and the gain obtained with the dynamic scheduler is about 5% on the factorization time. Even if our dynamic scheduler is perfectible, it already improves the hybrid MPI-thread version of the PASTIX solver for all the platforms.

6 Conclusion

The NUMA-aware allocation implemented in the PASTIX solver gives very good results and can be easily adapted to many applications. This points out that it is important to take care of memory allocation during the initialization steps when using threads on NUMA architectures.

Splitting communication and computational tasks also achieves some improvements in connection with the communication/computation overlap in the PASTIX solver. Future works on this subject is related to the PIOMAN library [14]. Such additional overlapping technique needs to be evaluated in the context of parallel sparse solvers. In the same time, we have to release some remaining constraints concerning the scheduling of communications by using the NEWMADELEINE library [3].

The dynamic scheduler gives encouraging results since we already improved the execution time for different test cases on platforms having or not a NUMA factor. The work stealing algorithm is perfectible. Firstly, it is possible to store informations about data locations to lead the steal in the upper levels of the elimination tree. Secondly, memory can be migrated closer to a thread, but such migration can be expensive and so needs to be controlled. We now plan to test the MARCEL bubble scheduler [13] to still improve performances in the context of applications based on a tree shaped dependency graph. Different threads and their datasets will be grouped in a bubble and bound to a part of the target architecture.

Finally, we are adapting the dynamic scheduler to the new Out-of-Core version of the PASTIX solver. New difficulties arise, related to the scheduling and the management of the computational tasks, 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. *SIMAX*, 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, Ultra-SPARC/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*, Long Beach, California, USA, 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 *Irregular'2000*, volume 1800 of *Lecture Notes in Computer Science*, pages 519–525, Cancun, Mexique, 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 *PPAM'05*, volume 3911 of *Lecture Notes in Computer Science*, pages 1050–1057, Poznan, Pologne, September 2005.

[8] P. Hénon, P. Ramet, and J. Roman. On finding approximate supernodes for an efficient ILU(k) factorization. *Parallel Computing*, 34:345–362, 2008.

[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 *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] María J. Martín, Inmaculada Pardines, and Francisco F. Rivera. Scheduling of algorithms based on elimination trees on NUMA systems. In *EuroPar'99*, volume 1685 of *Lecture Notes in Computer Science*, pages 1068–1072, Toulouse, France, 1999.

[12] Rajeev Thakur and William Gropp. Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE. In *EuroPVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 46–55, 2007.

[13] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework. In *EuroPar'07*, volume 4641 of *Lecture Notes in Computer Science*, pages 42–51, Rennes, France, August 2007.

[14] 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, 2007.

Contents