A NUMA Aware Scheduler for a Parallel Sparse Direct Solver

Mathieu Faverge, Pierre Ramet

A NUMA Aware Scheduler for a Parallel Sparse Direct Solver[☆]

Mathieu Faverge[a], Pierre Ramet[a]

[a] *INRIA Bordeaux - Sud-Ouest & LaBRI, ScAlApplix project,*
Université Bordeaux 1
351 cours de la Libération 33405 Talence, France

1. Introduction

Over the past few years, parallel sparse direct solvers have made significant progress [1, 3, 4]. They are now able to solve efficiently real-life three-dimensional problems with several millions of equations. Since the last decade, most of the supercomputer architectures are based on clusters of SMP (Symmetric Multi-Processor) nodes. In [5], 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. New NUMA (Non Uniform Memory Access) architectures have now an important effect on memory access costs, and introduce new problems of contentions 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. We also introduce a simple way to schedule dynamically an application based on a dependency tree while taking into account NUMA effects. Results obtained with these modifications are illustrated on performances obtained by our PaStiX solver on different platforms and matrices.

2. 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]. 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 different architectures. First, we study the cost of placement combinations of threads and memory on a set of BLAS functions. Table 1 and 2 shows the NUMA factor on one node of the NUMA8 architecture[1]. The results confirmed the presence of a shared memory bank for each chip of two cores. We also observed that the effects are more important for computations which do not reuse data as BLAS routines of level 1 and where data transfer is the bottleneck. The NUMA factor increases to 1.58 on this architecture.

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

[1] NUMA8 is a cluster of ten nodes of four dual-core opteron with 32GB of memory per node.

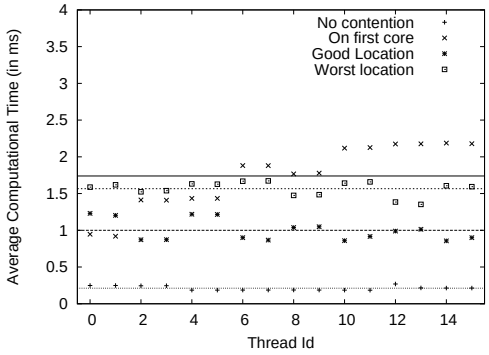
		Computational thread location							
		0	1	2	3	4	5	6	7
Data location	0	**1.00**	1.34	1.31	*1.57*	**1.00**	1.34	1.31	*1.57*
	1	1.33	**1.00**	*1.57*	1.31	1.33	**1.00**	*1.57*	1.31
	2	1.28	*1.57*	**1.00**	1.33	1.29	*1.57*	**1.00**	1.32
	3	*1.56*	1.32	1.34	**1.00**	*1.56*	1.31	1.33	**0.99**
	4	**1.00**	1.34	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 1: Influence of data placement on `dAXPY` on NUMA8

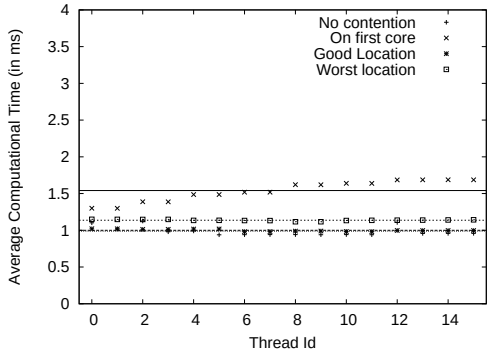
		Computational thread location							
		0	1	2	3	4	5	6	7
Data location	0	**1.00**	1.04	1.04	*1.07*	**1.00**	1.04	1.04	*1.07*
	1	1.04	**1.00**	*1.07*	1.04	1.04	**1.00**	*1.08*	1.04
	2	1.04	*1.07*	**1.00**	1.04	1.04	*1.07*	**1.00**	1.04
	3	*1.08*	1.04	1.04	**1.00**	*1.07*	1.04	1.05	**1.00**
	4	**1.00**	1.04	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 2: Influence of data placement on `dGEMM` on NUMA8

In summary, this study highlights the need to take into account possible NUMA effects during the memory allocation in threaded applications on multi-core architectures. Secondly, we study the NUMA factor with contention problems on a completely loaded computer which is more realistic. The experience is based on the computation of BLAS functions on a set of vectors/matrices on our architecture NUMA16[2]. We compare three different ways of data allocation: *On first core* (memory is allocated on the memory bank closed to the first chip-set), *Good location* (each thread allocates its own data-sets closed to it) and *Worst location* (data-sets are allocated away from the thread which needs them). The curve *(No contention)* represents the computational time for each core in the best case without contention. The Figure 2 highlights the important NUMA factor on this architecture. We observe an average factor of 1.7 on computations of level 1, but can increases to 2.2 in the worst location.



(a) dAXPY on NUMA16 architecture

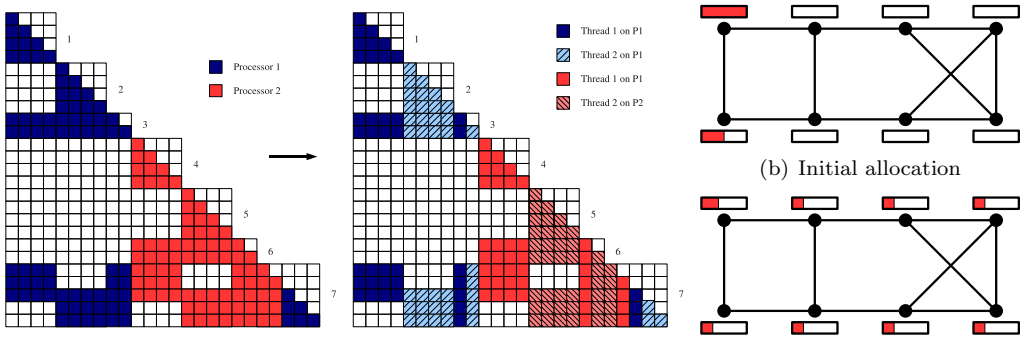


(b) dGEMM on NUMA16 architecture

The problem is that applications usually have a sequential initialization step that allocates and fills data-sets. Memory is so allocated on the first core of the system and the worst results can be obtained on threaded applications. 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.

We implement this solution on the hybrid MPI/thread version of the PaStiX solver. 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

[2]NUMA16 is a node of eight dual-core opteron with 64GB of memory.



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

(b) Initial allocation

(c) New NUMA-aware allocation

Figure 1: NUMA-aware allocation

on the node. The problem is therefore that all data-sets are allocated close to the core where the initialization step has occurred. Memory allocation is not evenly spread on the node and access times are thus not optimal (see Figure 1(b)). In the new version of PaStiX, data structures have been modified to allow each thread to allocate its part of the matrix as shown in Figure 1(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 1(c) and thus allows to obtain the best memory access as seen previously. Table 3, in next section, highlights the influence of NUMA-aware allocation on the factorization time on different platforms: NUMA8, NUMA16, and a symmetric architecture[3] called SMP16. A gain of 5% to 35% can be observed on NUMA architectures between the initial allocation (column V0) end the new NUMA-aware allocation (column V1). 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. These effects increase with the size of the platforms used.

3. 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. 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. We will now focus on the direct sparse linear solver PaStiX, which is our target application for this work. In its initial version, PaStiX schedules statically the tasks thanks to communications and BLAS costs models.

The proposed solution is compound of two pre-processing steps. The first one distributes efficiently data among all the nodes of the cluster thanks to the costs model used in the static

[3]SMP16 is a cluster of ten nodes of 16 power5 with 32GB of memory per node

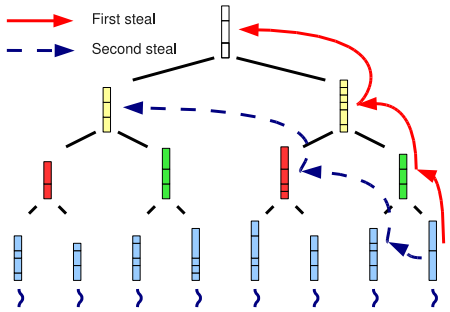
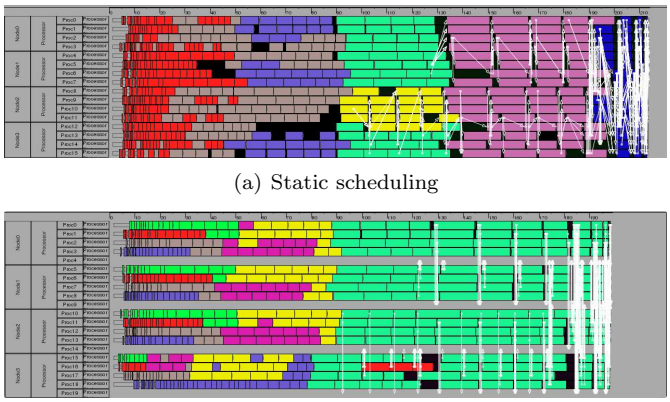


Figure 2: Work stealing algorithm



(a) Static scheduling

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

Figure 3: Gantt diagrams for the *MATR5* test case on NUMA8 with 4 MPI processes of 4 threads. Idle time is represented by black blocks and communications by white arrows.

scheduler. The second one builds a tree of tasks queue based on a proportional mapping over the local elimination tree. Thus, we obtain lists of tasks attached to different set of threads/cores on the architecture, as described in Figure 2. Tasks in the queues associated with the leaves of the tree are allocated by the only thread/core that is allowed 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 nodes 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 node. During the factorization, once a thread has no more jobs in its set of ready tasks, it steals jobs in the queues of its critical path as described by the filled arrows in Figure 2. 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. In the case where there is no jobs in this branch, it tries to steal a task in the sons of the nodes that belong to its critical path as described by the dotted arrows on the Figure 2.

Matrix	N	NNZ_A	NNZ_L	NUMA8			NUMA16			SMP16		
				V0	V1	V2	V0	V1	V2	V0	V1	V2
MATR5	485 597	24 233 141	1 361 345 320	437	410	**389**	527	341	**321**	162	161	**150**
AUDI	943 695	39 297 771	1 144 414 764	256	217	**210**	243	185	**176**	101	100	**100**

Table 3: 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 two Gantt diagrams (Figures 3) highlights the idle-time reduction thanks to the dynamic scheduling on the `MATR5` test case with 4 processes of 4 threads. Each color corresponds to a level in the elimination tree, black blocks highlight idle-times and white arrows are communications. The first diagram shows results obtain with the original static scheduler, and the second one with the

dynamic scheduler. The reduction of idle-times illustrated by the diagrams are confirmed by the factorization times on the two matrices reported in Table 3 which shows the improvements on the PaStiX solver inside one multi-cores node. All test cases are run using eight threads for NUMA8 platform and sixteen threads for NUMA16 and SMP16 clusters. Firstly, we observe that results are improved for the two test cases (but also for all matrices that have been factorized but where results are not presented in this abstract) and for the three architectures when the dynamic scheduler is enabled.

4. 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.

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 [6] 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 a recent 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] J. Antony, P. P. Janes, and A. 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] A. Gupta. Recent progress in general sparse direct solvers. In *LNCS*, volume 2073, pages 823–840, 2001.

[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, Jan. 2002.

[5] 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 *LNCS*, pages 1050–1057, Poznan, Pologne, Sept. 2005.

[6] S. Thibault, R. Namyst, and P.-A. Wacrenier. Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework. In *EuroPar'07*, volume 4641 of *LNCS*, pages 42–51, Rennes, France, Aug. 2007.