MPC: A Unified Parallel Runtime for Clusters of NUMA Machines

Marc Pérache[1], Hervé Jourdren[1], and Raymond Namyst[2]

[1] CEA/DAM Île de France Bruyères-le-Châtel F-91297 Arpajon Cedex
[2] Laboratoire Bordelais de Recherche en Informatique 351, cours de la Libération F-33405 Talence cedex

Abstract. Over the last decade, Message Passing Interface (MPI) has become a very successful parallel programming environment for distributed memory architectures such as clusters. However, the architecture of cluster node is currently evolving from small symmetric shared memory multiprocessors towards massively multicore, Non-Uniform Memory Access (NUMA) hardware. Although regular MPI implementations are using numerous optimizations to realize *zero copy* cache-oblivious data transfers within shared-memory nodes, they might prevent applications from achieving most of the hardware's performance simply because the scheduling of heavyweight processes is not flexible enough to dynamically fit the underlying hardware topology. This explains why several research efforts have investigated hybrid approaches mixing message passing between nodes and memory sharing inside nodes, such as MPI+OpenMP solutions [1,2]. However, these approaches require lots of programming efforts in order to adapt/rewrite existing MPI applications.

In this paper, we present the MultiProcessor Communications environnement (MPC), which aims at providing programmers with an efficient runtime system for their existing MPI, POSIX Thread or hybrid MPI+Thread applications. The key idea is to use user-level threads instead of processes over multiprocessor cluster nodes to increase scheduling flexibility, to better control memory allocations and optimize scheduling of the communication flows with other nodes. Most existing MPI applications can run over MPC with no modification. We obtained substantial gains (up to 20%) by using MPC instead of a regular MPI runtime on several scientific applications.

1 Introduction

Over the last decade, Message Passing Interface (MPI) has become a very successful parallel programming environment for distributed memory architectures such as clusters. This is mainly due to its efficiency and its portability. MPI is organized around the concept of a set of communicating tasks (often processes) using send and receive primitives. This feature allows programmers to split their application into several parts in an intuitive way and execute them on different processing nodes, processors or cores. On the top of the basic send/receive procedures, MPI offers a rich set of primitives available from C and Fortran interfaces

E. Luque, T. Margalef, and D. Benítez (Eds.): Euro-Par 2008, LNCS 5168, pp. 78–88, 2008.

which makes it possible to build powerful parallel applications. These characteristics led MPI to be adopted by a huge community of users and consequently, there are a lot of MPI applications developed in many different scientific and industrial fields.

Currently, the architecture of cluster nodes is evolving from small symmetric shared memory multiprocessors towards massively multicore, Non-Uniform Memory Access (NUMA) hardware [3]. The emergence of these deeply hierarchical architectures raises the need for a careful distribution of threads and data. Indeed, cache misses and NUMA penalties become more and more important with the complexity of the machine, making these constraints as important as parallelization itself. Parallel programming methods thus have to perfectly match the underling architecture to achieve high performance.

Distributed memory approaches such as MPI do not fully exploit the shared memory underlying architecture and thus may loose some efficiency. Shared memory approaches, based on explicit multithreading or language-generated multithreading (e.g. OpenMP) are more accurate on shared memory architectures. Thus, many hybrid approaches, typically mixing MPI and OpenMP, have been proposed to better exploit cluster of multiprocessors. However, they suffer from many drawbacks, including the fact that they require lots of programming efforts in order to adapt/rewrite existing MPI applications.

In this paper, we propose another approach which consists in a powerful runtime able to run MPI applications using user-level threads. This paper is organized as follows. Section 2 discusses the main parallel programming approaches used over clusters of multiprocessors. We then present our MPC environment in Section 3. Section 4 introduces our experiments and shows the efficiency of the MPC library.

2 Common Approaches for Programming Clusters of NUMA Nodes

To address the problem of programming efficiently clusters of multiprocessor nodes, several programming approaches have been explored.

Advanced MPI Implementations. Many MPI implementations are highly optimized for some specific architectures [4]. Such implementations typically offer specific hardware support for modern network interface cards (Quadrics, Myrinet, ...), *zero copy* data transfers, etc. They exhibit excellent performance regarding latency and bandwidth of communications, but do not fully exploit NUMA node capabilities.

OpenMP Implementations for Clusters. OpenMP is a very convenient way to parallelize existing codes, but it was initially limited to shared memory computing nodes. Nowadays, implementations such as Intel Cluster OpenMP [5] or OpenMPD [6] allow OpenMP applications to run over distributed memory

architectures, thanks to the use of software DSM runtime[1]. Obviously, any memory access that triggers the consistency mechanism is much more expensive than an ordinary access to a processor's memory. In fact, a memory access requiring the consistency mechanism can be hundreds to thousands of times slower than an access to any level of cache or hardware memory. That is why this approach is usually not as efficient as expected.

Hybrid MPI + OpenMP Approaches. Mixing MPI and OpenMP looks attractive to benefit from both data sharing on large shared memory nodes (thanks to OpenMP) and multiple processing node usage (thanks to MPI). Nevertheless, MPI and OpenMP implementations aren't very comprehensive to each other [7]. For instance, most MPI implementations use *busy waiting* techniques to increase the performance of communication event detection. Such a policy usually leads to disastrous performance in a multithreaded context. Moreover, MPI implementations are generally not fully thread-safe. Thus, the integration of MPI and OpenMP seems to be a promising approach, but is currently quite difficult to realize in practice.

Process Virtualization. Process virtualization [8] is an efficient way to benefit from shared memory computing nodes. It dissociates tasks and processes. In standard MPI approaches, a task is a process. With this approach, tasks are mapped to threads. Thus, load balancing, *zero copy* method, overloading[2], ... are easier to implement. AMPI [9] and TOMPI [10] are two MPI implementations that use process virtualization. MPC also uses process virtualization to implement its distributed memory API. As emphasized in the remainder of this paper, collective operations and scheduling have been strongly optimized in this context.

3 MPC: MultiProcessor Communications

The purpose of MPC is to provide a single API for programming distributed and shared memory architectures [11]. The MPC library implementation solves the issues of mixing *distributed* and *shared* memory approaches thanks to its unified runtime. The design of MPC follows four objectives. The first one is portability, which is required for most scientific computer codes. The second goal is dynamic workload balancing, which is crucial for example with adaptive mesh refinement. Another important goal is to reach a high level of performance. Last but not least, the fourth goal is to provide an API that allows easy migration from an MPI application, or a multithreaded one, to the MPC unified framework. This section is organized as follows. First of all the MPC execution model is described. Then, we present the specialized MxN thread library. Follows the optimized scheduler that integrates collective communications. The MPC allocator is introduced in the last part.

[1] Distributed Shared Memory.
[2] Using more tasks than cores.

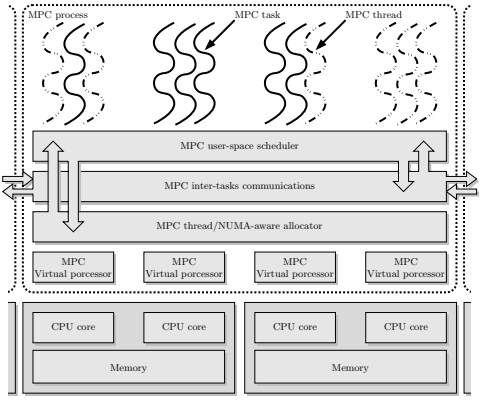


Fig. 1. Execution model of MPC, with an example involving both MPC-tasks and MPC-threads (hybrid distributed/shared memory approach)

3.1 Execution Model

MPC supports mixed shared- and distributed-memory approaches *via* task virtualization. Most MPI implementations map the P MPI tasks on P cores. In contrast, the T MPC-tasks are mapped on T internal threads. These threads are then mapped on P cores ($P \leq T$) thanks to the MPC MxN scheduler. Communications between MPC-tasks use shared memory for intra node communications, and MPI (or sockets) for inter nodes communications. MPC also provides a thread API. MPC-threads are directly mapped onto internal threads. Whichever the programming model, tasks are mapped onto these internal threads which in turn are mapped onto cores. Figure 1 summarizes the general execution model of MPC.

This execution model relies on the design of an optimized thread scheduler. This scheduler deals with inter-task message passing through the MPC *message passing* API and task synchronizations though the MPC *thread* API. This optimized scheduler is the key factor of MPC's performances. It allows to hide the hybrid approach difficulties by providing efficient overloading mechanism, by overlapping communications with computing work, ... This model also allows an optimized implementation of communications between MPC-tasks using user-level *zero copy* allowed by the shared address space among tasks.

When using overloading, there are more tasks than cores for efficient load balancing, thanks to the large number of tasks that increases scheduling possibilities. Overloading is also an optimization method for some well-balanced codes that may become cache-oblivious. An SPMD[3] parallel code may typically be subdivided into more subdomains than processors. Thus, each subdomain will be smaller and core cache memory usage will be increased. Nevertheless, overloading requires an highly-optimized communication library to increase code performances.

[3] SPMD: Single Program Multiple Data.

The choice of a multithreaded approach as the execution model of MPC allows to address three of our initials goals: portability, performances and load balancing. The last goal – "easy migration from existing codes to MPC" – is achieved thanks to the unified distributed/shared memory API. MPC provides a large subset of both the message passing interface (MPI) and the POSIX thread API. MPC uses process virtualization like AMPI or TOMPI but it features a more sophisticated implementation of scheduling and memory allocation. MPC also extends process virtualization to hybrid MPI+thread approaches.

3.2 Specialized MxN Thread Scheduling

The MPC execution model maps tasks on lightweight user-level threads integrated in a MxN thread library such as PM2 [12] or NGPT [13]. MxN libraries allow to create and schedule threads by user-level code. This brings fast context switching and total control over scheduling. Thus, it is feasible to strongly optimize scheduling and to provide load balancing according to task communications and synchronizations.

The MPC scheduler is optimized to deal with tasks communications and synchronizations. It provides an integrated polling method, used for task waiting for operation completion. This method avoids busy waiting and thus allows task overloading. A second optimization deals with collective communications. The MPC scheduler embeds mechanisms to perform collective communications inside the scheduler. Our MxN thread library also provides user-defined load balancing methods on the top of MPC scheduler, *via* the MPC API. It allows to move tasks between cores and determine workload.

MPC is also optimized for NUMA architectures where data locality is crucial. MPC provides a NUMA-aware memory allocator to insure data locality. Whereas load balancing may generate task migrations that break data locality, our scheduler provides primitives capable of migrate thread's data according to the core used by this thread and the memory hierarchy.

3.3 Scheduler-Integrated Collective Communications

The MPC scheduler is optimized for centralized collective communications such as broadcast, reduce and barrier. These communications share a unique resource (counter for barrier, buffer for reduce and broadcast). These communications also have a similar logical execution path. The similarity between these collective communications allows us to optimize them together.

Overloading execution requires optimisations to reach high performances. First of all, the collectiver communications have to avoid busy waiting. The overloading approach allows many tasks on different communicators on the same core. Thus, a collective communication call have to provide efficient execution of non participating tasks. That's why busy waiting methods must be avoided to reach a good efficiency. Overloading implies smart methods to "freeze" and "wakeup" tasks efficiently, without disturbing the execution of the other ready threads running on the same core.

The second issue comes with the large number of tasks that may be involved on a single collective communication (see Section 4.1). So, $O(n)$ wakeup methods are prohibited. We need an efficient wakeup method associated with our freeze method, that ideally perform $O(1)$ wakeup and freeze.

The following section is organized as follows: first the collective communication algorithm is described. Then freeze/wakeup methods are introduced. Thread migration is taken into account in the last part.

Collective Communication Algorithm. The algorithm used to perform collective communications is generic[4] and divided into two parts. The first one is the per-core part. At a given time, there is only one executed thread on each core. That is why the first part of this algorithm corresponds to a centralized lock free approach. Its pseudo-code is described in function *contribute_local_core* in Figure 2(a).

The second part of this algorithm performs inter-core collective communications. This part uses a tree based algorithm in order to maximize data locality and scalability. This part of the algorithm allows to synchronize cores where all participating tasks in the collective communication have done their contribution to the collective communication call. The pseudo-code of inter-core collective communications is described in function *contribute_local_group* in Figure 2(b).

```
contribute_local_core (core_rank, data_in,
    data_out,function)
  if (virtual_core[core_rank].nb_tasks == 0)
      copy data_in to virtual_core[core_rank].data_in
  else
      function (data_in, virtual_core[core_rank].data_in)
  endif
  virtual_core[core_rank].nb_tasks++
  if (virtual_core[core_rank].nb_tasks ==
      virtual_core[core_rank].nb_tasks_total)
      contribute_local_group(top_level, function)
      for i in virtual_core[core_rank].task_list do
          copy i.data_out
      done
      wakeup current_task to virtual_core[core_rank].task_list
  else
      freeze (current_task,data_out) in virtual_core[core_rank].task_list
  endif
```

(a) Per-core contribution.

```
contribute_local_group (level, function)
  if (root == level)
      copy level.data_in to level.data_out
  else
      lock father
      if (level.father.nb_tasks == 0)
          copy level.data_in to level.father.data_in
      else
          function (level.data_in, level.father.data_in)
      endif
      level.father.nb_tasks++
      if (level.father.nb_tasks == level.father.nb_tasks_total)
          contribute_local_group (level.father, function)
          for i in level.task_list do
              copy i.data_out
          done
          wakeup level.task_list
          unlock father
      else
          unlock father
          freeze (current_task,level.data_out) in level.task_list
  endif
```

(b) Per-group (i.e. between cores) contribution.

Fig. 2. Scheduler-integrated collective communications algorithm (data_in and data_out values are input and output arrays used during reduction and broadcast)

[4] The same code is used for barrier, reduction and broadcast.

Freeze/wakeup Methods. All the complexity of efficient collective communications lies in the task freeze and wakeup methods. These methods have been inserted into the thread scheduler in order to provide $O(1)$ complexity. The freeze function allows to insert the calling task into a list that matches the internal scheduler ready thread list structure. This function does not require lock because at a time, there is only one executed thread per core.

Regarding the wakeup function, this function only takes the freezed thread list created within the freeze function. Then, it inserts it directly into the scheduler ready list using a $O(1)$ technique.

Thread Migration. The algorithm presented above assumes that the distribution of tasks among cores is known thanks to the *nb_tasks_total* variable, whereas MPC allows task migration for load balancing. Thus, we have extended the previous algorithm using a lazy update method to deal with task migration. The aim of the lazy update method is to perform the lowest number of updates. Thus, it does not perform collective communication structure update for each migration. It only requires a check at each collective communication call to determine if the current core used by the calling task is the same than the one used for the previous collective communication call. In migration case, first of all, the calling task is temporary moved to its previous core. Then, it performs the collective communication call. Finally, it schedules a collective communication initialization. This initialization will be performed by all tasks at the next collective communication call. Such a method allows to aggregate all migrations between two collective communication calls.

3.4 Optimized NUMA-Aware and Thread-Aware Allocator

The memory allocator if often a bottleneck for parallel multithreaded programs [14]. It may severely limits program performance and scalability on multiprocessor systems. Allocators suffer from problems that include poor scalability and heap organization leading to false sharing [15]. That is why programmers hoping to achieve performance improvements often use custom memory allocators [16].

The MPC NUMA-aware thread-aware allocator combines a global heap and per-thread heaps. Thus, it avoids false sharing and provides very low synchronization cost in the most common cases. The allocator is linked to the MPC topology module and thread scheduler to maximize data locality on NUMA architectures. The specificity of our allocator mainly resides in the NUMA-aware aspect. The well-known use of multiple local heaps allows to avoid false-sharing, allows scalability, but does not insure data locality. That's why our allocator is linked with the topology module of MPC. This module determines the memory hierarchy and drives the system physical page allocation. This allocation insures that a new allocated page will be local to a thread but does not insure long time data locality. That's why the MPC allocator communicates with the user-level thread scheduler of MPC, in order to move tasks' pages according to thread migration among cores.

Combination of multiple heaps and data locality techniques provides to MPC an efficient and scalable allocator, suited to large multiprocessor NUMA nodes.

4 Experimental Results

Two machines have been used, with very different architectures. The first one is a node of the Bull TERA-10 cluster of CEA/DAM Île-de-France. This node consists of 8 Dual core Itanium2 Montecito processors (16 cores) distributed over four Quad Brick Blocks (QBB). Each QBB has two processors and must be viewed as a NUMA node inside a global shared-memory machine of four QBB and 48 GB of memory. The second machine consists of 2 Quad-core Xeon (8 cores). This machine is a UMA SMP machine.

Experiments reported here are based on two basic numerical kernels. The first one is an *advection* benchmark, corresponding to a 2D upwind *explicit* scheme on a regular Cartesian grid. In SPMD parallel mode, the scheme just requires one point-to-point communication per direction (update of ghost cells), and one reduction for the prediction of the next time step (CFL condition). The second numerical kernel is a *conduction* benchmark, corresponding to a 2D Cartesian grid *implicit* heat conduction solver, based on a a five-point stencil and a Conjugate Gradient method with diagonal preconditionning. In SPMD mode, this kernel can be distinguished from the precedent one due to the conjugate gradient method that involves many reductions at each time step (scalar product).

4.1 Scalability Results with Domain Overloading

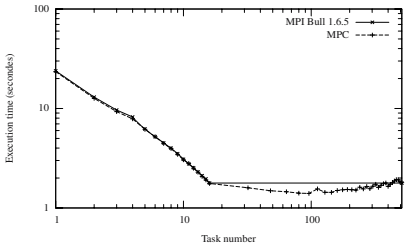
Results are given Figure 3. The first part of each curves illustrates the scalability of MPC versus MPI. The MPI Bull implementation is the manufacturer implementation available on TERA-10. This implementation is optimized according to the underlying architecture and thus, it reaches very good performances. The MPI implementation available on Xeon machine is a standard MPICH2 implementation.

Comparison of MPI Bull and MPC shows the rather good performances achieved by MPC. More generally, in scalability terms, MPI and MPC implementations reach similar performances on both architectures.

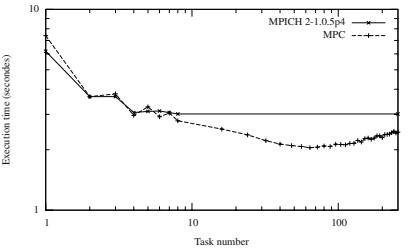
To cover the whole spectrum, we have performed multi-node execution. The following table presents relative execution time to MPI for the *advection* and *conduction* benchmarks on 32 core Itanium2 Montecito architecture (2 nodes) with $4,000 \times 1,000$ cells.

Benchmark	Number of tasks								
	32	64	128	192	256	320	384	448	512
Advection (50 cycles)	0.99	0.90	0.84	0.80	0.81	0.84	0.86	0.85	0.87
Conduction (25 cycles)	1.02	0.96	0.92	0.89	0.92	0.91	0.93	0.91	0.91

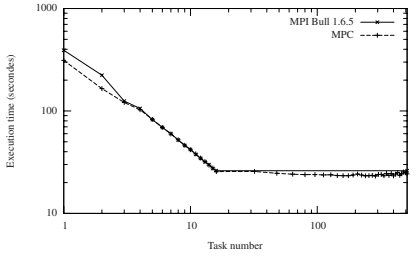
These results illustrate the low overhead of MPC in multi-node context, demonstrating also the benefits of overloading as in single-node context. Let us mention that a large scale run over $4,096$ processors of TERA-10 has been



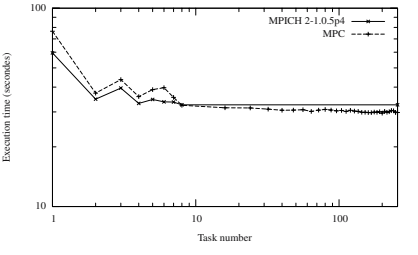
(a) Execution time for the *advection* benchmark on 16 core Itanium2 Montecito architecture with $2,000 \times 1,000$ cells and 50 cycles



(b) Execution time for the *advection* benchmark on 8 core Xeon $1,000 \times 1,000$ cells and 50 cycles



(c) Execution time for the *conduction* benchmark on 16 core Itanium2 Montecito architecture with $4,000 \times 1,000$ cells and 10 cycles



(d) Execution time for the *conduction* benchmark on 8 core Xeon with $2,000 \times 1,000$ cells and 10 cycles

Fig. 3. Evaluation of scalability and overloading method on representative scientific computing 2D code. Parallelization is done via 1D domain decomposition.

performed [17] to check scalability of robustness of the MPC library, along with numerous computations with the HERA AMR platform [18], up to several thousands processors.

These codes have been tested with domain overloading. In this context, MPC allows better execution times thanks to cache effects. This also illustrates the good performances of our scheduler-integrated collective communications, especially with the *conduction* benchmark, that performs a high number of reductions per cycle. Overloading allows to gain more than 10% and up to 20% on the execution time, without *any* modification to the original code.

4.2 Memory Allocation and Data Placement Results

In order to evaluate the MPC NUMA-aware and thread-aware allocator, the *advection* benchmark on TERA-10 has been used. Results are summarized in the following table.

Allocator	Number of tasks								
	1	2	4	6	8	10	12	14	16
Standard	23.47s	12.79s	8.27s	5.44s	4.21s	3.25s	2.66s	2.22s	2.02s
MPC	23.47s	12.64s	7.86s	5.20s	3.95s	3.05s	2.51s	2.08s	1.76s
Gain	0.00%	1.19%	5.22%	4.62%	6.58%	6.57%	5.98%	6.73%	14.77%

A reduction of 14% of the overall execution time is observed in multithread mode, when compared to the standard C library memory allocator. Results with the *conduction* benchmark are similar. Results are similar on the Xeon machine. The MPC allocator strongly contributes to the performances of MPC, performances that would not be achieved without an optimized NUMA-aware and thread-aware allocator.

5 Conclusion and Future Works

In this paper, we have introduced the MPC library. MPC offers a unified runtime for both distributed-memory and shared-memory parallel codes. The MPC internal execution model also insures a good integration of *hybrid* shared- and distributed-memory approaches *via* appropriate *thread scheduling*. To reach a high level of performance from SMP machines to large NUMA nodes, MPC provides a NUMA-aware and thread-aware *allocator* that contributes to the overall scalability and efficiency. The MPC scheduler is optimized to deal with a *very* large number of threads, providing pooling and highly efficient scheduler-integrated collective communications. The scheduler and allocator modules cooperate to preserve data locality, with or without thread migration, a crucial issue when dealing with NUMA nodes. Our experiments have shown that the MPC approach leads to a high level of performance on several HPC parallel codes.

Today, MPC allows an easy migration path for existing MPI parallel codes and multithread codes based on the POSIX thread API. As far as MPI codes are concerned, the main issue is thread safety. Such codes have to be thread safe to be converted to MPC. This limitation will soon disappear using precompilation or embedded virtual machines. Another evolution of MPC will concern a full implementation of the OpenMP standard, with an extension of MPC's scheduling policy to OpenMP-related tasks. With the support of the three most widely used parallel APIs[5], MPC will then be able to address most parallel computer codes, offering efficient scheduling, memory allocation, NUMA locality and loadbalancing on today and tomorrow architectures.

References

1. Cappello, F., Etiemble, D.: MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. SuperComputing (2000)
2. Smith, L., Bull, M.: Development of mixed mode MPI/OpenMP applications. Scientific Programming (2001)
3. Van der Steen, A.: Overview of recent supercomputers (2006)
4. Liu, J., Chandrasekaran, B., Jiang, J., Kini, S., Yu, W., Buntinas, D., Wyckoff, P., Panda, D.: Performance comparison of MPI implementations over InfiniBand Myrinet and Quarics (2003)
5. Hoeflinger, J.: Extending OpenMP* to clusters (2006)

[5] MPI 1.0, POSIX Thread and (planed) OpenMP 1.0.

6. Lee, J., Sato, M., Boku, T.: Design and implementation of OpenMPD: An OpenMP-like programming language for distributed memory systems. In: Chapman, B.M., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935. Springer, Heidelberg (2008)
7. Smith, L., Kent, P.: Development and performances of a mixed OpenMP/MPI quantum monte carlo code. Concurrency: Practice and Experience (2000)
8. Kalé, L.: The virtualization model of parallel programming: runtime optimizations and the state of art. In: LACSI (2002)
9. Huang, C., Lawlor, O., V., K.: Adaptive MPI. In: Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (2003)
10. Demaine, E.: A Threads-Only MPI implementation for the development of parallel programming. In: Proceedings of the 11th International Symposium on High Performance Computing Systems (1997)
11. Pérache, M.: Contribution à l'élaboration d'environnements de programmation dédiés au calcul scientifique hautes performances. PhD thesis, Bordeaux 1 University (2006)
12. Namyst, R.: PM2: un environnement pour une conception portable et une exécution efficace des applications parallèlles irrégulières. PhD thesis, Lille 1 university (1997)
13. Abt, B., Desai, S., Howell, D., Perez-Gonzalet, I., McCraken, D.: Next Generation POSIX Threading Project (2002), `http://www-124.ibm.com/developerworks/oss/pthread`
14. Berger, E., McKinley, K., Blumofe, R., Wilson, P.: Hoard: a scalable memory allocator for multithreaded applications. In: International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX) (2000)
15. Torrellas, J., Lam, M.S., L., H.J.: False sharing and spatial locality in multiprocessor caches. IEEE Transaction on Computers (1994)
16. Berger, E., Zorn, B., McKinley, K.: Composing high-performance memory allocators. In: Proceedings of the ACM SIGPLAN conferance on Programming Language Design and Implementation (2001)
17. Del Pino, S., Despres, B., Have, P., Jourdren, H., Piserchia, P.F.: 3d finite volume simulation of acoustic waves in the earth atmosphere. Computer and fluids (submitted)
18. Jourdren, H.: HERA: a hydrodynamic AMR platform for multi-physics simulations. In: Adaptive mesh refinement - theory and applications, LNCSE (2005)