An Efficient OpenMP Runtime System for Hierarchical Architectures

Samuel Thibault, François Broquedis, Brice Goglin, Raymond Namyst, Pierre-André Wacrenier

An Efficient OpenMP Runtime System for Hierarchical Architectures

Samuel Thibault, François Broquedis, Brice Goglin,
Raymond Namyst, and Pierre-André Wacrenier

INRIA Futurs - LaBRI
351 cours de la libération
33405 Talence cedex, France
{thibault,goglin,namyst,wacrenier}@labri.fr,
francois.broquedis@etu.u-bordeaux1.fr

Abstract. Exploiting the full computational power of always deeper hierarchical multiprocessor machines requires a very careful distribution of threads and data among the underlying non-uniform architecture. The emergence of multi-core chips and NUMA machines makes it important to minimize the number of remote memory accesses, to favor cache affinities, and to guarantee fast completion of synchronization steps. By using the BubbleSched platform as a threading backend for the GOMP OpenMP compiler, we are able to easily transpose affinities of thread teams into scheduling hints using abstractions called bubbles. We then propose a scheduling strategy suited to nested OpenMP parallelism. The resulting preliminary performance evaluations show an important improvement of the speedup on a typical NAS OpenMP benchmark application.

Keywords: *OpenMP, Nested Parallelism, Hierarchical Thread Scheduling, Bubbles, Multi-Core, NUMA, SMP.*

1 Introduction

The emergence of deeply hierarchical architectures based on multi-threaded multi-core chips and NUMA machines 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. They require some new programming models and new tools to make the most out of these underlying architectures.

As quoted by Gao *et al.* [GSS+06], it is important to expose domain-specific knowledge semantics to the various software components in order to organize computation according to the application and architecture. Indeed, the whole software stack, from the application to the scheduler, should be involved in the parallelizing, scheduling and locality adaptation decisions by providing useful information to the other components.

Therefore, in OpenMP frameworks, the information extracted by the compiler (about memory affinity and adherence to the same parallel section) can be

very useful for the guidance of task/thread scheduling. On the other hand, it is very important to rely on architecture specific constraints when making these scheduling decisions. A tight interaction between the OpenMP stack and the underlying hardware-aware scheduler is thus required.

The most delicate point, when dealing with irregular applications, is to exploit this knowledge at runtime (during the whole execution time) so as to maintain a good balancing of threads when events arise (task termination, creation of new embedded parallel sections, blocking synchronization, etc.).

In this paper, we propose a hierarchical threading library able to follow/obey scheduling directives and advices in a very powerful manner. Scheduling information (affinity, group membership) is attached to bubbles, which are abstractions that can recursively group threads or bubbles sharing common properties.

We report on preliminary experiences on top of a 8-way multi-core NUMA machine and we show that running OpenMP applications on top of our runtime system greatly enhances performance on hierarchical architectures under irregular conditions. We also propose insights regarding the extraction of useful information by the compiler for our runtime and discuss the addition of a couple of non-standard OpenMP directives that would improve performance.

2 Scheduling Applications Featuring Nested, Irregular Parallelism

Achieving the best possible performance when programming OpenMP applications requires developers to expose the parallelism and to explicitly design their code to drive its parallel behavior. Therefore, it is quite common nowadays to define per-thread specific data structures (in order to avoid false-sharing) and use a static, possibly pre-calculated, distribution of the workload to get good data locality [MM06]. Indeed, this model suits very well regular applications with coarse-grain parallelism.

However, this approach is hardly usable when dealing with irregular applications that rather need a dynamic load balancing mechanism. The use of complex synchronization schemes, or even blocking systems calls, may also be responsible for introducing irregularities regarding the computing load on the available processors. Using OpenMP dynamic scheduling directives can sometimes improve performance. In some cases, however, it may penalize data locality or even introduce false sharing effects, which can severely impact performance on hierarchical architectures.

Another approach is to increase the number of potential parallel tasks using nested parallelism, so that threads can be dynamically (re)allocated according to the workload disparity. The performance of such a dynamic thread management, when supported[1], heavily relies on the underlying runtime implementation, but also on the underlying operating system's scheduler. This explains why OpenMP users have been experiencing poor performance with the nested capabilities of

[1] Nested parallelism is currently an optional feature in OpenMP.

some OpenMP compilers, and have ended up performing explicit thread programming on top of OpenMP [BS05,GOM+00] or explicitely binding thread groups to processors [Zha06].

Nevertheless, there exists some very good implementations of OpenMP nested parallelism, such as Omni/ST [TTSY00] for instance. Such implementations are typically based on a fine-grain thread management system that uses a fixed number of threads to execute an arbitrary number of *filaments*, as done in the Cilk multithreaded system [FLR98]. The performance obtained over symmetrical multiprocessors is often very good, mostly because many tasks can be executed sequentially with almost no overhead when all processors are busy. However, since these systems provide no support for attaching high level information such as memory affinity to the generated tasks, many applications will actually achieve poor performance on hierarchical, NUMA multiprocessors.

One could probably enhance these OpenMP implementations to use affinity information extracted by the compiler so as to better distribute tasks or threads over the underlying processors. However, since only the underlying thread scheduler has complete control over scheduling events such as processor idleness, blocking syscall or even thread preemption, this information could only be used to influence task allocation at the beginning of each parallel section.

We believe that a better solution would be to transmit information extracted by the compiler to the underlying thread scheduler *in a persistent manner*, and that only a tight integration of application-provided meta-data and architecture description can let the underlying scheduler take appropriate decisions during the whole application run time. In other words, one can see this configurable scheduler framework as a domain-specific language enabling scientists to transfer their knowledge to the runtime system [GSS+06].

3 MaGOMP: an Implementation of GNU OpenMP for Hierarchical Machines

To evaluate the potential gain of providing a thread scheduler with persistent information extracted by an OpenMP compiler, we have extended the GNU OpenMP runtime system (i.e. the `libgomp` library) so as to rely on the *Marcel* thread library. This library provides facilities for attaching various information to groups of threads, together with a framework that helps to develop schedulers capable of using these metadata. Scheduling policies are simply developed as *plug-ins*.

Before describing our extensions to the GNU OpenMP compiler suite, we first present the most important features of the *Marcel* library.

3.1 The *Bubble* Scheduling Model

Marcel is a POSIX-compliant thread library featuring extensions for easily writing efficient, customized schedulers for hierarchical architectures. The API of *Marcel* provides functions to group threads using nested sets called **bubbles** [Thi05].

These abstractions allow programmers to model the relationships between the different threads of an application. Figure 1 illustrates this concept: four threads are grouped as pairs in bubbles (assuming they work on the same data), which are themselves grouped along another thread in a larger bubble (assuming they share information less often). Bubbles allow expression of relationships like data sharing, collective operations, or more generally a particular scheduling policy need (serialization, gang scheduling, etc.). Hierarchical machines are modelled with a hierarchy of runqueues. Each component of each hierarchical level of the machine is represented by one runqueue: one per logical processor, one per core, one per chip, one per NUMA node, and one for the whole machine. *Marcel*'s ground scheduler then uses a hierarchical *Self-Scheduling* algorithm. Whenever idle, a processor scans all runqueues that span it, and executes the first thread that is found, from bottom to top. For instance, if the thread is on a runqueue that represents a chip, it may be run by any processor of this chip (see Figure 2).

Fig. 1. Expressing thread relationships: graphical and tree-based representations.

As mentioned previously, *Marcel* provides a high-level API for writing powerful and portable schedulers that manipulate threads, bubbles and runqueues. Threads and bubbles are equally considered as **entities**, while bubbles and runqueues are equally considered as **scheduling holders**, so that we end up with entities (threads or bubbles) that we can schedule on holders (bubbles or runqueues). Primitives are then provided for manipulating entities in holders. Runqueues can be accessed through vectors, and can be walked through thanks to "parent" and "child" pointers. Some functions permit to gather statistics about bubbles so as to take appropriate decisions. This includes for instance the total number of threads and the number of running threads, but also various information such as the accumulated expected and current CPU computation time or memory usage, or the cache miss rates.

Writing a high-level scheduler actually reduces to writing some hook functions. The main one is actually called when the ground Self-Scheduler encounters a bubble during its search for the next thread to execute. The default implementation just looks for a thread in the bubble (or one of its sub-bubbles) and switches to it. The `bubble_tick()` hook is called when some time-slice for a bubble expires, and hence permits periodic operations on bubbles with a per-bubble notion of time. Of course, mere "daemon" threads can also be started for performing background operations. As a result, scheduling experts may manipu-

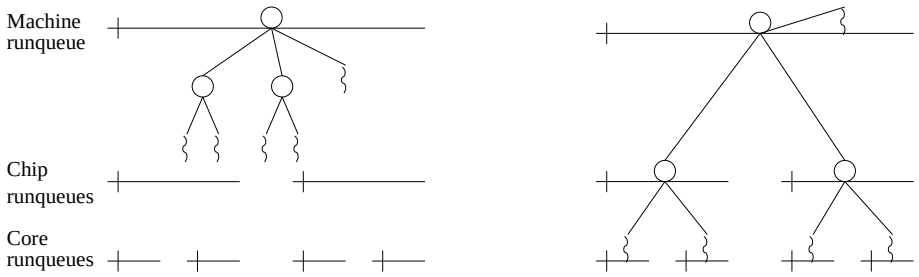


Fig. 2. Scheduling of bubbles and threads on the runqueues of a hierarchical machine.

late threads with a high level of abstraction by deciding the placement of bubbles on runqueues, or even temporarily putting some bubbles aside (by defining their own runqueues that the basic Self-Scheduler will not look at).

3.2 Generating Bubbles Out of OpenMP Parallel Sections

The GNU OpenMP compiler[gom], GOMP, is based on an extension of the GCC 4.2 compiler that converts OpenMP pragmas into threading calls. The creation of threads and teams is actually delegated to a shared library, `libgomp`, which contains an abstraction layer to map OpenMP threads onto various thread implementations. This way, any application previously compiled by GOMP may be relinked against an implementation of `libgomp` on another thread type and transparently work the same.

We used this flexible design to develop MaGOMP, a port of GOMP on top of the *Marcel* threading library in which BubbleSched is implemented. To do so, a *Marcel* adaptation of `libgomp` threads has been added to the existing abstraction layer. We rely on *Marcel*'s fully POSIX compatible interface to guarantee that MaGOMP will behave as well as GOMP on pthreads. Then, it becomes possible to run any existing OpenMP application on top of BubbleSched by simply relinking it.

Once *Marcel* threads are created they basically behave by default as native pthreads without any notion of team or memory affinity. BubbleSched hooks have been added in the `libgomp` code to provide information about thread teams by creating bubbles accordingly.

Therefore, when a thread encounters a nested parallel region and becomes the master of a new team, it creates a bubble within its currently holding bubble. Then, it moves itself into this new bubble and creates the team's slave threads inside it. Finally, the master dispatches the workload across the team. Once their work is completed, slave threads die while the master destroys the bubble and returns to its original team. As shown on Figure 3, only a few lines of code are needed to associate a nested team hierarchy with a bubble hierarchy.

```
void gomp_team_start (void (*fn) (void *), void *data, unsigned nthreads,
                      struct gomp_work_share *work_share) {
  struct gomp_team *team;
  team = new_team (nthreads, work_share);
  ... /* Pack 'fn' and 'data' into the 'start_data' structure */

  if (nthreads > 1 && team->prev_ts.team != NULL) {
    /* nested parallelism, insert a marcel bubble */
    marcel_bubble_t *holder = marcel_bubble_holding_task (thr->tid);
    marcel_bubble_init (&team->bubble);
    marcel_bubble_insertbubble (holder, &team->bubble);
    marcel_bubble_inserttask (&team->bubble, thr->tid);
    marcel_attr_setinitbubble (&gomp_thread_attr, &team->bubble);
  }

  for(int i=1; i < nbthreads; i++) {
      pthread_create (NULL, &gomp_thread_attr,
                      gomp_thread_start, start_data);
      ...
  }
}
```

Fig. 3. One-to-One correspondence between *Marcel*'s bubble and GOMP's team hierarchies.

3.3 A Scheduling Strategy Suited to OpenMP Nested Parallelism

The challenge of a scheduler for the nested parallelism of OpenMP resides in how to distribute the threads over the machine. This must be done in a way that favors both a good balancing of the computation and, in the case of multi-core and NUMA machines, a good affinity of threads, for better cache effects and avoiding the remote memory access penalty.

For achieving this, we wrote a **bubble spread** scheduler consisting of a mere recursive function that uses the API described in section 3.1 to greedily distribute the hierarchy of bubbles and threads over the hierarchy of runqueues. This function takes in an array of "current entities" and an array of "current runqueues". It first sorts the list of current entities according to their computation load (either explicitly specified by the programmer, or inferred from the number of threads). It then greedily distributes them onto the current runqueues by keeping assigning the biggest entity to the least loaded runqueue[2], and recurse separately into the sub-runqueues of each current runqueue.

It often happens that an entity is much more loaded than others (because it is a very deep hierarchical bubble for instance). In such a case, a recursive call is made with this bubble "exploded": the bubble is removed from the "current entities" and replaced by its content (bubbles and threads). How big a bubble needs

[2] This algorithm comes from the greedy algorithm typically used for resolving the bi-partition problem.

to be for being exploded is a parameter that has to be tuned. This may depend on the application itself, since it permits to choose between respecting affinities (by pulling intact bubbles as low as possible) and balancing the computation load (by exploding bubbles for having small entities for better distribution).

This way, affinities between threads are taken into account: since they are by construction in the same bubble hierarchy, the threads of the same external loop iterations are spread together on the same NUMA node or the same multicore chip for instance, thus reducing the NUMA penalty and enhancing cache effects.

Other repartition algorithms are of course possible, we are currently working on a even more affinity-based algorithm that avoids bubble explosions as much as possible.

4 Performance Evaluation

We validated our approach by experimenting with the BT-MZ application. It is one of the 3D Fluid-Dynamics simulation applications of the Multi-Zone version of the NAS Parallel Benchmark [dWJ03] 3.2. In this version, the mesh is split in the x and y directions into zones. Parallelization is then performed twice: simulation can be performed rather independently on the different zones with periodic face data exchange (coarse grain *outer* parallelization), and simulation itself can be parallelized among the z axis (fine grain *inner* parallelization). As opposed to other Multi-Zone NAS Parallel Benchmarks, the BT-MZ case is interesting because zones have very irregular sizes: the size of the biggest zone can be as big as 25 times the size of the smallest one. In the original SMP source code, outer parallelization is achieved by using Unix processes while the inner parallelization is achieved through an OpenMP static parallel section. Similarly to Ayguade *et al.* [AGMJ04], we modified this to use two nested OpenMP static parallel sections instead, using $n_o * n_i$ threads.

The target machine holds 8 dual-core AMD Opteron 1.8GHz NUMA chips (hence a total of 16 cores) and 64GB of memory. The measured NUMA factor between chips[3] varies from 1.06 (for neighbor chips) to 1.4 (for most distant chips). We used the class A problem, composed of 16 zones. We tested both the Native POSIX Thread Library of Linux 2.6 (NPTL) and the *Marcel* library, before trying the *Marcel* library with our *bubble spread* scheduler.

We first tried non-nested approaches by only enabling either outer parallelism or inner parallelism, as shown in Figure 4:

Outer parallelism($n_o * 1$): Zones themselves are distributed among the processors. Due to the irregular sizes of zones and the fact that there is only a few of them, the computation is not well balanced, and hence the achieved speedup is limited by the biggest zones.

Inner parallelism($1 * n_i$): Simulation in zones are performed sequentially, but simulations themselves are parallelized among the z axis. The computation

[3] The NUMA factor is the ratio between remote memory access and local memory access times.

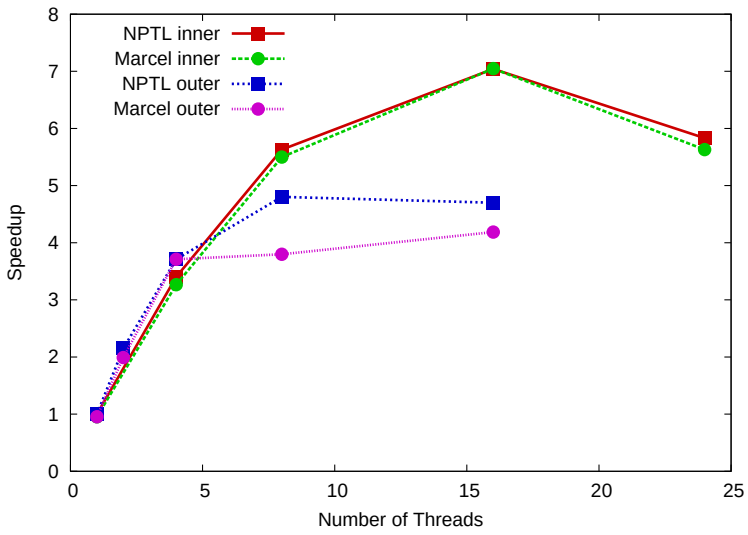


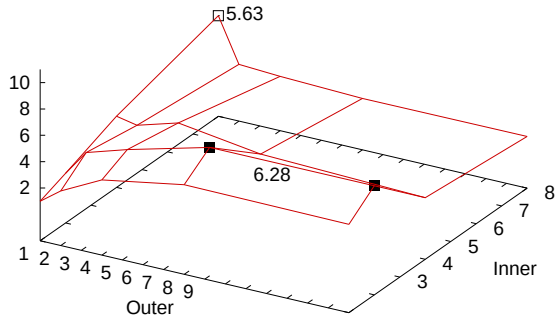
Fig. 4. Outer parallelism ($n_o * 1$) and inner parallelism ($1 * n_i$).

balance is excellent, but the nature of the simulation introduces a lot of inter-processor data exchange. Particularly because of the NUMA nature of the machine, the speedup is hence limited to 7.

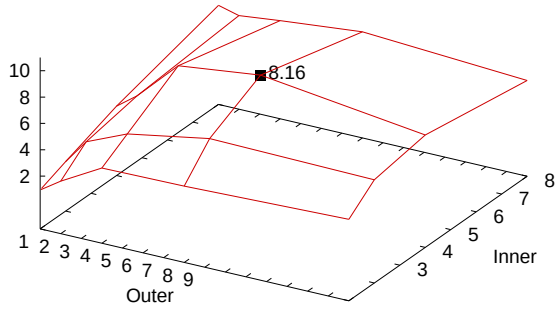
So as to get the benefits of both approaches (locality and balance), we then tried the nested approach by enabling both parallelisms. As discussed by Duran *et al.* [DGC05], the achieved speedup depends on the relative number of threads created by the inner and the outer parallelisms, so we tried up to 16 threads for the outer parallelism (i.e. the maximum since there are 16 zones), and up to 8 threads for the inner parallelism. The results are shown on Figure 5. The nested speedup achieved by NPTL is very limited (up to 6.28), and is actually worse than what pure inner parallelism can achieve (almost 7, not represented here because the "Inner" axis maximum was truncated to 8 for better readability). *Marcel* behaves better (probably because user threads are more lightweight), but it still can not achieve a better speedup than 8.16. This is due to the fact that neither NPTL nor *Marcel* takes affinities of threads into account, leading to very frequent remote memory accesses, cache invalidation, etc. We hence used our bubble strategy to distribute the bubble hierarchy corresponding to the nested OpenMP parallelism over the whole machine, and could then achieve better results (up to 10.2 speedup with $16 * 4$ threads). This improvement is due to the fact that the bubble strategy carefully distribute the computation over the machine (on runqueues) in an affinity-aware way (the bubble hierarchy).

It must be noted that for achieving the latter result, the only addition we had to do to the BT-MZ source code is the following line:

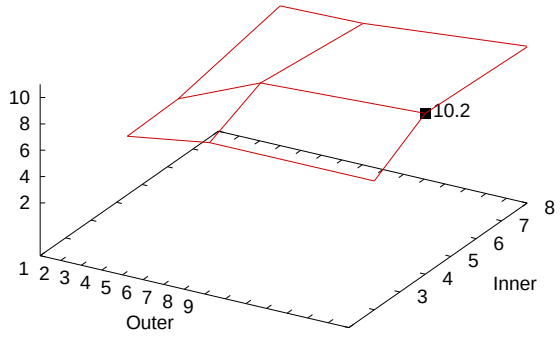
```
call marcel_set_load(int(proc_zone_size(myid+1)))
```



(a) NPTL



(b) Marcel



(c) Bubbles

Fig. 5. Nested parallelism.

that explicitly tells the bubble spread scheduler the load of each zone, so that they can be properly distributed over the machine. Such a clue (which could even be dynamic) is very precious for permitting the runtime environment to make appropriate decisions, and should probably be added as an extension to the OpenMP standard. Another way to achieve load balancing would be to create more or less threads according to the zone size [AGMJ04]. This is however a bit more difficult to implement than the mere function call above.

5 Conclusion

In this paper, we discussed the importance of establishing a persistent cooperation between an OpenMP compiler and the underlying runtime system for achieving high performance on nowadays multi-core NUMA machines. We showed how we extended the GNU OpenMP implementation, GOMP, for making use of the flexible *Marcel* thread library and its high-level *bubble* abstraction. This permitted us to implement a scheduling strategy that is suited to OpenMP nested parallelism. The preliminary results show that it improves the achieved speedup a lot.

At this point, we are enhancing our implementation so as to introduce just-in-time allocation for *Marcel* threads, bringing in the notion of "ghost" threads, that would only be allocated when first run by a processor. In the short term, we will keep validating the obtained results over several other OpenMP applications, such as Ondes3D (French Atomic Energy Commission). We will compare the resulting performance with other OpenMP compilers and runtimes. We also intend to develop an extension to the OpenMP standard that will provide programmers with the ability to specify load information in their applications, which the runtime will be able to use to efficiently distribute threads.

In the longer run, we plan to extract the properties of memory affinity at the compiler level, and express them by injecting gathered information into more accurate attributes within the bubble abstraction. These properties may be obtained either thanks to new directives *à la* UPC [4] [CDC+99] or be computed automatically via static analysis [SGDA05]. For instance, this kind of information is helpful for a bubble-spreading scheduler, as we want to determine which bubbles to explode or to decide whether or not it is interesting to apply a *migrate-on-next-touch* mecanism [NLRH06] upon a scheduler decision. All these extensions will rely on a memory management library that attaches information to bubbles according to memory affinity, so that, when migrating bubbles, the runtime system can migrate not only threads but also the corresponding data.

[4] The UPC `forall` statement adds to the traditional `for` statement a fourth field that describes the affinity under which to execute the loop

6 Software Availability

Marcel and BubbleSched are available for download within the PM2 distribution at `http://runtime.futurs.inria.fr/Runtime/logiciels.html` under the GPL license. The MaGOMP port of `libgomp` will be available soon and may be obtained on demand in the meantime.

References

AGMJ04. Eduard Ayguade, Marc Gonzalez, Xavier Martorell, and Gabriele Jost. Employing Nested OpenMP for the Parallelization of Multi-Zone Computational Fluid Dynamics Applications. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

BS05. R. Blikberg and T Sørevik. Load balancing and OpenMP implementation of nested parallelism. *Parallel Computing*, 31(10-12):984–998, October 2005.

CDC+99. W. Carlson, J.M. Draper, D.E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, George Mason University, May 1999.

DGC05. Alejandro Duran, Marc Gonzàles, and Julita Corbalán. Automatic Thread Distribution for Nested Parallelism in OpenMP. In *19th ACM International Conference on Supercomputing*, pages 121–130, Cambridge, MA, USA, June 2005.

dWJ03. Rob F. Van der Wijngaart and Haoqiang Jin. NAS Parallel Benchmarks, Multi-Zone Versions. Technical Report NAS-03-010, NASA Advanced Supercomputing (NAS) Division, 2003.

FLR98. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998. `http://theory.lcs.mit.edu/pub/cilk/cilk5.ps.gz`.

gom. GOMP – An OpenMP implementation for GCC. `http://gcc.gnu.org/projects/gomp/`.

GOM+00. Marc Gonzalez, Jose Oliver, Xavier Martorell, Eduard Ayguade, Jesus Labarta, and Nacho Navarro. OpenMP Extensions for Thread Groups and Their Run-Time Support. In *Languages and Compilers for Parallel Computing*. Springer Verlag, 2000.

GSS+06. Guang R. Gao, Thomas Sterling, Rick Stevens, Mark Hereld, and Weirong Zhu. Hierarchical multithreading: programming model and system software. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

MM06. Jaydeep Marathe and Frank Mueller. Hardware Profile-guided Automatic Page Placement for ccNUMA Systems. In *Sixth Symposium on Principles and Practice of Parallel Programming*, March 2006.

NLRH06. Markus Nordén, Henrik Löf, Jarmo Rantakokko, and Sverker Holmgren. Geographical Locality and Dynamic Data Migration for OpenMP Implementations of Adaptive PDE Solvers. In *Second International Workshop on OpenMP (IWOMP 2006)*, Reims, France, 2006.

SGDA05. Xipen Shen, Yaoqing Gao, Chen Ding, and Roch Archambault. Lightweight Reference Affinity Analysis. In *19th ACM International Conference on Supercomputing*, pages 131–140, Cambridge, MA, USA, June 2005.

Thi05. Samuel Thibault. A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines. In *Second International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, Cambridge / USA, 06 2005. ICS / ACM / IRISA.

TTSY00. Yoshizumi Tanaka, Kenjiro Taura, Mitsuhisa Sato, and Akinori Yonezawa. Performance Evaluation of OpenMP Applications with Nested Parallelism. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 100–112, 2000.

Zha06. Guansong Zhang. Extending the OpenMP standard for thread mapping and grouping. In *Second International Workshop on OpenMP (IWOMP 2006)*, Reims, France, 2006.