Scheduling Dynamic OpenMP Applications over Multicore Architectures

François Broquedis, François Diakhaté, Samuel Thibault,
Olivier Aumage, Raymond Namyst, and Pierre-André Wacrenier

INRIA Futurs - LaBRI — Université Bordeaux 1, France

Abstract. Approaching the theoretical performance of hierarchical multicore machines requires a very careful distribution of threads and data among the underlying non-uniform architecture in order to minimize cache misses and NUMA penalties. While it is acknowledged that OpenMP can enhance the quality of thread scheduling on such architectures in a portable way, by transmitting precious information about the affinities between threads and data to the underlying runtime system, most OpenMP runtime systems are actually unable to efficiently support highly irregular, massively parallel applications on NUMA machines.

In this paper, we present a thread scheduling policy suited to the execution of OpenMP programs featuring irregular and massive nested parallelism over hierarchical architectures. Our policy enforces a distribution of threads that maximizes the proximity of threads belonging to the same parallel region, and uses a NUMA-aware work stealing strategy when load balancing is needed. It has been developed as a *plug-in* to the forestGOMP OpenMP platform [TBG+07]. We demonstrate the efficiency of our approach with a highly irregular recursive OpenMP program resulting from the generic parallelization of a surface reconstruction application. We achieve a speedup of 14 on a 16-core machine with no application-level optimization.

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

1 Introduction

Cache-coherent multiprocessor architectures now commonly introduce multiple levels of locality preference between processor and caches or memory banks. The penalty paid for non-local memory accesses can deeply affect speed-ups when such expensive accesses frequently occur throughout application runs. It is therefore acknowledged that multithreaded programs must carefully distribute threads onto the processors to minimize both cache misses and NUMA penalties. Traditional "opportunistic" scheduling approaches used by most operating systems fail in exploiting hierarchical architectures efficiently however, because they lack information about application behaviour.

Successfully using NUMA architectures requires an in-depth knowledge of the application behaviour in terms of memory access patterns, affinity and

R. Eigenmann and B.R. de Supinski (Eds.): IWOMP 2008, LNCS 5004, pp. 170–180, 2008.

inter-thread collaborations, relationship and synchronization. Gao et al. share this analysis [GSS+06]: They emphasize the importance of exposing domain-specific knowledge semantics to the underlying scheduling layer. Parallel languages such as OpenMP, that rely on the combination of a dedicated compiler and a set of code annotations to extract the parallel structure of applications and to generate scheduling hints for the underlying runtime system, are a great step forward in this respect. However, they currently miss architecture-aware runtime systems that would make an effective and thorough *exploitation* of the gathered knowledge at runtime. As quoted in a proposal for task parallelism in OpenMP [ACD+07]: "The overhead associated with the creation of parallel regions, the varying levels of support in different implementations, the limits to the total number of threads in the application and to the allowed levels of parallelism, and the impossibility of controlling load balancing, make this approach impractical". Moreover, most advanced OpenMP compilers [TTSY00, HD07, THH+05, BS05, DGC05] (featuring super lightweight threads, work stealing techniques, etc.) are not yet NUMA-aware.

In this paper, we present an extension to the GNU OpenMP runtime system that is capable of running dynamic irregular programs over NUMA multicore machines very efficiently. Our runtime generates nested sets of threads called *bubbles*, which encapsulate threads sharing common data, each time an OpenMP parallel region is encountered [TBG+07]. We have designed a NUMA-aware scheduling policy that dynamically maps these *bubbles* onto the various levels of the underlying hierarchical architecture. When load balancing needs to be performed, threads are thus redistributed with respect to their affinity relations. We validate our approach using the OpenMP version of a real-life application (the MPU [OBA+03] parallel surface reconstruction algorithm) that features a highly irregular divide-and-conquer parallel structure based on a recursive refinement process. We show that the OpenMP version of this program clearly draws a substantial benefit from our approach.

2 An OpenMP Platform for Developing and Tuning NUMA-aware Thread Scheduling Policies

To deal with dynamic, irregular OpenMP applications, we claim that the key step is to transmit information extracted by the compiler to the underlying thread scheduler *in a continuous way*. Indeed, only a tight integration of application-provided meta-data and architecture description can let the underlying runtime system take appropriate decisions during the whole application run time.

Thus we have designed "forestGOMP ", an extension to the GNU OpenMP runtime system [gom] that relies on the Marcel/BubbleSched thread scheduling package. BubbleSched provides facilities for attaching various information to groups of threads called *bubbles*, together with a framework that helps to develop schedulers capable of using these metadata.

2.1 Related Work

The numerous studies and papers [MAN+99, TTSY00, DSCL04, DGC05, BS05, GSW+06, aMST07] that emphasized multilevel parallelism as a promising path toward scalability with OpenMP, gradually brought compiler researchers and vendors to put more of their efforts on OpenMP nested parallelism. Modern OpenMP compilers have some support for nested parallelism and either rely on an efficient user-level thread library (NANOS Nthlib [GOM+01], Omni/ST [TTSY00], OMPi [HD07]) or on a pool of threads avoiding useless and costly creation/destruction (Intel Compiler [TGS+03, TGBS05], OdinMP [Kar05]).

For instance, Omni/ST is based on a fine-grain thread management system that uses a fixed number of threads to execute an arbitrary number of *filaments*, as with the Cilk multithreaded system [FLR98]. The performance obtained over symmetrical multiprocessors is often very good, mostly because many tasks can be executed sequentially with hardly any overhead when all processors are busy.

To deal with hierarchical architectures, the OMPi C Compiler uses user-level non-preemptive threads that are inserted in the processor runqueues in the following way: threads that are spawned at the first level of parallelism are distributed cyclically and appended at the tail of the ready queues. Inner level threads are inserted at the head of the ready queue of the processor that created them. In order to favor data locality, an idle processor extracts threads from the head of its local queue and steals work from the tail of the remote ones. Moreover the work-stealing scheme follows the computer hierarchy. However, neither Omni/ST nor OMPi provide any support for annotating generated tasks with high level information such as memory affinity. The theft of a thread blindly ignores and breaks the affinity relation between threads that were created together. This may put a strain on the performance on hierarchical, NUMA multiprocessors.

Several OpenMP language extensions have been proposed to control the allocation of work to the participating threads. The mechanism in GOMP [GOM+01] to control the binding of threads is useful to tune an application for a given computer. Binding, however, is non-portable from the performance point of view. In order to favor affinities in a more portable manner, the NANOS compiler [DGC05, AGMJ04] allows to associate groups of threads with parallel regions in a static way. The OpenUH Compiler [CHJ+06] proposes a mecanism to accurately select the threads of a subteam, although this proposition does not involve nested parallelism.

Finally, the KAI/Intel [STH+04] and the NANOS Mercurium compilers [BDG+04] support task parallelism and a proposal for parallel tasks in OpenMP 3.0 has been written [ACD+07]. This is a major step towards natural support of MIMD applications in OpenMP. Moreover, the OpenMP task paradigm will naturally lead to the generation of structured parallelism, so we claim that the techniques presented in this paper will also be beneficial to programs featuring task parallelism.

3 A Scheduling Policy Guided by Affinity Hints

The challenge of a scheduler for the nested parallelism 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.

3.1 Assumptions

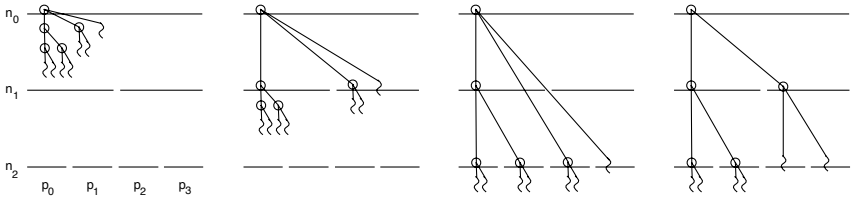
Divide and conquer algorithms generate intensively cooperating groups of threads that run smoother if they are scheduled on the same limited subset of processors. A bad distribution of these collaborating entities results in multiple expensive NUMA accesses over hierarchical architectures, that lowers the general performance of parallel applications. Alternatively, a distribution that considers those affinity relations entails a better use of cache memory, and improves local memory accesses.

The *Affinity* bubble-scheduler is specifically designed to tackle irregular applications based on a divide and conquer scheme. In this aim, we consider that each bubble contains threads and subbubbles that are heavily related, most of the time through data sharing. We assume that the best thread distribution is obtained by scheduling each entity contained in a bubble on the same processor, sometimes breaking the load balancing scheme, even if a local redistribution is needed once in a while. This scheduler provides two main algorithms, to distribute thread and bubble entities over the different processors initially, and to rebalance work if one of them becomes idle.

3.2 Initial Thread Distribution

Entities scheduled in the same bubble should not be torn apart. Nevertheless a bubble can be required to extract its contents to increase the number of executable entities in order to occupy every processor of the architecture. This bubble is then said to be exploded. The runqueue level where a bubble explodes during the distribution is crucial to determine whether affinity relations are preserved or not. For instance, if a bubble is exploded on the top level of the topology, its contents can be scheduled on any processor. Therefore the *Affinity* thread distribution algorithm delays these explosions as much as possible, to maximize locality between the released entities.

More precisely, this first scheduling step is based on a mere recursive algorithm to greedily distribute the hierarchy of bubbles and threads over the hierarchy of runqueues. Upon each call, the algorithm counts the entities available to be distributed from the considered runqueues. If there are enough entities to occupy the complete set of processors covered by the runqueues, entities are greedily distributed over the underlying lists. Otherwise, the algorithm analyzes the contents of each available bubble to determine the ones that hold enough threads or subbubbles to occupy a complete subset of processors on their own. If



(a) Initial state

(b) Greedy distribution after the root bubble explosion

(c) Final distribution

(d) Work rebalancing after processor p3 idles

Fig. 1. Threads and bubbles distribution by the *Affinity* scheduler

so, bubble explosions are delayed to a further step, thus avoiding early separation of collaborating entities.

Figure 1(a) shows the initial state of this recursive algorithm which has been developed to guarantee bubbles and threads distribution from the most general level of the topology, representing the whole computer, to the most specific ones. Figure 1(c) shows the resulting distribution, where only the main top-level bubble has exploded. This approach obviously values affinity relations over load balancing, and could not be efficient without a NUMA-aware work stealing algorithm that rearranges the thread distribution when a processor turns to be inactive.

3.3 NUMA-aware Work Stealing

The irregular behaviour of some applications prevents estimation of the load of each created thread. This lack of load hints forces the *Affinity* scheduler to equally consider every entity. As a result, a continuous thread creation and destruction scheme may unbalance the initial thread repartition, and some of the processors may become idle.

The *Affinity* scheduler implements a dedicated work stealing algorithm to prevent these processors from remaining inactive for too long. This algorithm tracks down lists to steal from, from the most local lists to the most global one if necessary, expanding the search domain as long as no eligible runqueue has been found. Entities are thus stolen as locally as possible. If several entities are usable for work stealing, the *Affinity* scheduler arbitrarily picks the most loaded one, considering the number of recursively contained threads. If only one bubble is found during the stealing process, its contents are browsed to pick a complete subtree of entities, as illustrated by figures 1(d). When a thread, or a bubble, is finally chosen, the algorithm moves its ancestors to the most internal level of the topology common to the source runqueue and the idle processor, to avoid locking convention issues.

3.4 Discussion

A call to the *Affinity*'s thread distribution algorithm generally results in assigning a tree of entities to every processor, similar to the Cilk language or OMPi approaches to deal with divide and conquer applications. Statistically, the working load left to an entity located in the upper part of the tree is bigger than the one executed by the leaves-positioned threads. The *Affinity* scheduler therefore tries to steal from the top of the entity hierarchy, but differs from Cilk implementations by (1) looking for eligible subtrees as close as possible from the idle processor, instead of randomly picking a victim runqueue, and (2) stealing a set of threads that work together rather a lonely thread (like OMPi does). This way of stealing respects the hierarchical nature of both NUMA architectures and the application parallelization scheme.

4 Implicit Surface Reconstruction Application

With *Affinity* and several features of MARCEL, it is now possible to parallelize many recursive divide and conquer algorithms, using a naive approach and simple OpenMP constructs, and yet to obtain good speedups. To back our claim, we show that an extremely irregular divide and conquer algorithm, the Multi-Level Partition of Unity algorithm (MPU) [OBA+03], can be parallelized efficiently only by adding a few lines of code to its implementation.

This surface reconstruction algorithm processes a cloud of points sampling a geometric surface, so as to compute a mathematical representation of this sampled surface. Its main use is related to 3D scanners, that is, devices that are capable of sampling the surface of a physical object by extracting a finite set of 3D points. Reconstructing the whole surface from its samples is required for many applications ranging from rendering to physical simulations.

Thanks to its divide and conquer scheme it is one of the fastest reconstruction algorithms available. Starting from a box containing the whole cloud of points, it tries to fit a simple surface (a quadric) to the points. This surface is implicit, which means that it is defined by a real valued function defined over the entire space and whose value is zero for every point of the implicit surface. If the fitted surface does

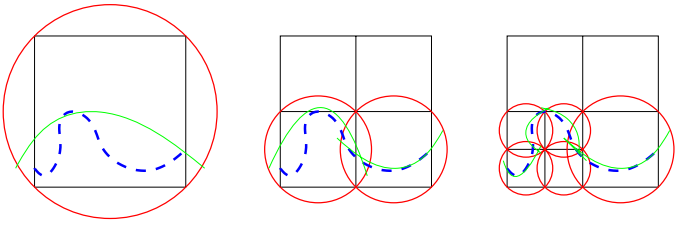


Fig. 2. Adaptive surface fitting using a recursive subdivision of space which forms a tree hierarchy. Each box is subdivided until the fitted surface is close enough to the points. The resulting surface is a weighted average of each local approximation using partition of unity functions.

```
void Node::compute() {
  computeApprox();
  if(_error > _max_error) {
    splitCell();
    for(int i=0; i<8; i++)
      _children[i]->compute();
  }
}
```

Fig. 3. Sequential MPU code to process a node. An approximation is computed and the node is subdvided if it is not precise enough. This process is then repeated recursively.

```
void Node::compute() {
  computeApprox();
  if(_error > _max_error) {
    splitCell();
#pragma omp parallel for
    for(int i=0; i<8; i++)
      _children[i]->compute();
  }
}
```

Fig. 4. Parallel MPU code to process a node. A single OpenMP directive has been added to indicate that every node can be processed concurrently.

not approximate the points closely enough, that is, if there are points too far away from the fitted surface, the box is subdivided into 8 subboxes, thus forming an octree. This process is applied recursively to each child box until the error between each approximation and the points of its box is small enough (see Figure 2).

This divide and conquer approach is made possible by the use of partition of unity functions. Indeed, using these functions makes it possible to define the global reconstructed surface as a weighted average of each function defining the local surface approximations. The weight of each local approximation in the weighted average at a given point in space is at its highest at the center of its box, and decreases as the distance to this center increases.

What makes this technique especially attractive is that there is no "stitching" involved between locally computed surfaces. This makes parallelization easier because such a step would require many synchronisations between threads working on neighbour nodes. Therefore this algorithm is well suited to parallelization since every node of the tree can be processed concurrently. The difficulty resides in balancing the work between the processors as the tree is very irregular and there is no simple way to predict where the tree is going to be refined. Ideally the programmer should be able to simply express that the function calls for processing the nodes can be executed concurrently and the runtime would be responsible for balancing this work on the processors.

OpenMP provides constructs that are very well suited to this task, and parallelizing this algorithm using OpenMP is a matter of inserting a few lines of code to indicate that each time a node is subdivided, its 8 children can be processed concurrently (see Fig. 3 and 4). Running such an application efficiently is challenging for however, because runtime systems need not only to deal with a large number of thread creations/destructions (up to tens of millions for large datasets), but also to schedule them in a way that optimizes memory locality.

5 Evaluation

We validated our approach by experimenting with the MPU application on a cloud of 437644 points, which leads to the creation of 101185 threads.

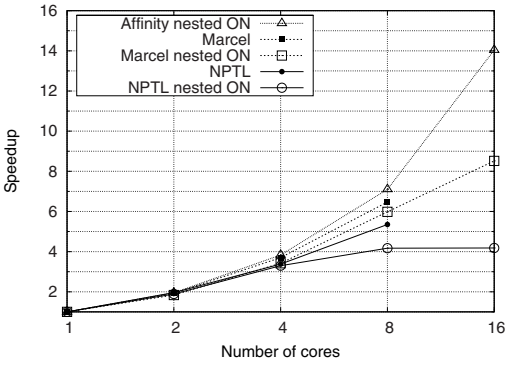


Fig. 5. Speedup of various MPU implementations

The target machine holds 8 dualcore AMD Opteron chips (hence a total of 16 cores) and 64GB of memory. The measured NUMA factor between chips varies from 1.06 (for neighbor chips) to 1.4 (for most distant chips). We tested both the Native POSIX Thread Library of Linux 2.6 (NPTL) and the MARCEL library, partitioning the set of usable cores in order to execute our tests on respectively 2, 4, 8 and 16 cores. The results can be seen on figure 5.

We first tried non-nested approaches to compare the behaviour of these two libraries. Each parallel construct generates a number of threads equal to the number of available cores. The MPU algorithm divides the computed surface in eight different subdomains, every time the refinement primitive is called. Running non-nested tests with a number of threads exceeding 8 is thus not relevant, only the first eight ones will be occupied. The MARCEL thread scheduler operates at user-level, and is less preemptive than the one used by NPTL. MPU thus runs much faster with MARCEL threads.

In the next experiments, we allowed the GOMP compiler to create extra threads when a nested parallel construct is encountered. This approach theoretically suits the MPU application divide and conquer nature. We achieved the best speedups by creating 4 threads at each parallel section. Allowing nested approaches results in creating a great number of threads, and thread creation and management primitives are more expensive in a kernel-level thread library like NPTL. Those used by MARCEL are lighter, which explains why it scales better. On the other hand, neither the runtime system of those libraries has sufficient information about threads' relations to adjust their distribution, so that related threads may be executed by cores located on different NUMA nodes, and the speedup is yet a bit limited. On the contrary, respecting affinity relations by locally scheduling groups of threads results in much better speedups, as can be seen on the *Affinity* curve.

We then evaluated the effectiveness of *Affinity*'s NUMA-aware scheduling algorithm by running two tests. In the first test, the MPU application is *unmodified* but the work stealing algorithm of *Affinity* is replaced by a *random* work stealing

algorithm: the victim is elected from a randomly chosen runqueue. The achieved speedup on 16 cores varies between 8.2 and 11.82.

In the second test, the MPU application is *modified*. A single thread is bound per processor, which is used to schedule tasks in the form of lightweight threads. An idle thread tries to steal tasks from the most local queues when necessary. The structure of the application allows this version to use a lock-free stealing strategy most of the time, in a Cilk-like manner. The result is that the Cilk-like version obtains the best speed-up, 15.05 on 16 cores, *at the cost of portability*, since the MPU application was modified to integrate this algorithm. With a speedup of 14.04 out of 16 cores, the forestGOMP results almost reach the *cilk-like* results, *without sacrificing* either portability or generality in application-specific optimizations.

6 Conclusion

To exploit nowaday's multiprocessor machines at their full potential, it is crucial to transmit affinity relationships between application threads to the underlying runtime system scheduler. Efficiently scheduling an application on top of a NUMA architecture indeed requires an accurate knowledge of both the machine and the application behaviour in order to make appropriate NUMA-aware scheduling decisions at runtime. Parallel programming languages such as OpenMP are therefore inherently promising since they are particularly fitted for transparent information gathering.

In this paper, we presented a scheduling policy called *Affinity* embedded in our GOMP-based OpenMP scheduling framework and programming environment. *Affinity* is built on the bubble concept and the rich set of manipulation primitives offered by the Marcel/BubbleSched hierarchical scheduler toolkit to let the application programmer naturally express the thread cooperation affinities and to follow these hints in the actual scheduling process. The experiments we conducted on MPU, a real-life highly irregular surface reconstruction application made a strong case in validating our approach in terms of development *easiness* for the programmer, *portability* and *performance*. Our approach is therefore a way for experts to build complex scheduling strategies that take characteristics of the application into account. Using and mixing such strategies, application programmers get a greater control on scheduling of their OpenMP programs.

In the near future, we intend to investigate two main directions. First, we are currently extending our BubbleSched platform with advanced memory management primitives in order to allocate, register and potentially migrate memory areas used within bubbles on NUMA architectures. This will enable us to take into account memory "attraction" when computing thread redistribution patterns and to operate data movements when significant thread redistributions have to be performed. Second, forestGOMP could use static code analysis in determining the groups of threads that are effectively sharing data, and estimating bubble thickness. This information would improve the way the *Affinity* scheduler distributes entities, by naturally preferring the less cooperating threads

groups when a bubble must be exploded. This attribute could even be enriched by dynamically refreshed hardware statistics on memory access frequency. Both directions will benefit from our ongoing work towards supporting OpenMP 3.0 tasks in FORESTGOMP.

References

[ACD+07] Ayguade, E., Copty, N., Duranl, A., Hoeflinger, J., Lin, Y., Massaioli, F., Su, E., Unnikrishnan, P., Zhang, G.: A proposal for task parallelism in OpenMP. In: Third International Workshop on OpenMP (IWOMP 2007), Beijing, China (2007)

[AGMJ04] Ayguade, E., Gonzalez, M., Martorell, X., Jost, G.: Employing Nested OpenMP for the Parallelization of Multi-Zone Computational Fluid Dynamics Applications. In: 18th International Parallel and Distributed Processing Symposium (IPDPS) (2004)

[aMST07] an Mey, D., Sarholz, S., Terboven, C.: Nested Parallelization with OpenMP. Parallel Computing 35(5), 459–476 (2007)

[BDG+04] Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E., Labarta, J.: Nanos mercurium: A research compiler for openmp. In: European Workshop on OpenMP (EWOMP) (October 2004)

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

[CHJ+06] Chapman, B.M., Huang, L., Jin, H., Jost, G., de Supinski, B.R.: Extending openmp worksharing directives for multithreading. In: EuroPar 2006 Parallel Processing (2006)

[DGC05] Duran, A., Gonzàles, M., Corbalán, J.: Automatic Thread Distribution for Nested Parallelism in OpenMP. In: 19th ACM International Conference on Supercomputing, Cambridge, MA, USA, June 2005, pp. 121–130 (2005)

[DSCL04] Duran, A., Silvera, R., Corbalán, J., Labarta, J.: Runtime adjustment of parallel nested loops. In: Chapman, B.M. (ed.) WOMPAT 2004. LNCS, vol. 3349, Springer, Heidelberg (2005)

[FLR98] Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada (June 1998)

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

[GOM+01] Gonzalez, M., Oliver, J., Martorell, X., Ayguade, E., Labarta, J., Navarro, N.: OpenMP Extensions for Thread Groups and Their Run-Time Support. In: Languages and Compilers for Parallel Computing, Springer, Heidelberg (2001)

[GSS+06] Gao, G.R., Sterling, T., Stevens, R., Hereld, M., Zhu, W.: Hierarchical multithreading: programming model and system software. In: 20th International Parallel and Distributed Processing Symposium (IPDPS) (April 2006)

[GSW+06] Gerndt, A., Sarholz, S., Wolter, M., an Mey, D., Bischof, C., Kuhlen, T.: Nested OpenMP for Efficient Computation of 3D Critical Points in Multi-Block CFD Datasets. In: Super Computing (November 2006)

[HD07] Hadjidoukas, P.E., Dimakopoulos, V.V.: Nested Parallelism in the OMPi OpenMP/C compiler. In: EuroPar, Rennes,France, July 2007, ACM, New York (2007)

[Kar05] Karlsson, S.: An Introduction to Balder - An OpenMP Run-time Library for Clusters of SMPs. In: International Workshop on OpenMP (IWOMP) (June 2005)

[MAN+99] Martorell, X., Ayguadé, E., Navarro, N., Corbalán, J., González, M., Labarta, J.: Thread Fork/Join Techniques for Multi-Level Parallelism Exploitation in NUMA Multiprocessors. In: International Conference on SuperComputing, pp. 294–301. ACM Press, New York (1999)

[OBA+03] Ohtake, Y., Belyaev, A., Alexa, M., Turk, G., Seidel, H.-P.: Multi-level partition of unity implicits. ACM Trans. Graph. 22(3), 463–470 (2003)

[STH+04] Su, E., Tian, X., Haab, M.G.G., Shah, S., Petersen, P.: Compiler Support of the Workqueuing Execution Model for Intel SMP Architectures. In: European Workshop on OpenMP (EWOMP) (October 2004)

[TBG+07] Thibault, S., Broquedis, F., Goglin, B., Namyst, R., Wacrenier, P.-A.: An Efficient OpenMP Runtime System for Hierarchical Architectures. In: International Workshop on OpenMP (IWOMP), Beijing,China, June 2007, pp. 148–159 (2007)

[TGBS05] Tian, X., Girkar, M., Bik, A., Saito, H.: Practical Compiler Techniques on Efficient Multithreaded Code Generation for OpenMP Programs. Comput. J. 48(5), 588–601 (2005)

[TGS+03] Tian, X., Girkar, M., Shah, S., Armstrong, D., Su, E., Petersen, P.: Compiler and Runtime Support for Running OpenMP Programs on Pentium- and Itanium-Architectures. In: Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments, April 2003, pp. 47–55 (2003)

[THH+05] Tian, X., Hoeflinger, J.P., Haab, G., Chen, Y.-K., Girkar, M., Shah, S.: A compiler for exploiting nested parallelism in OpenMP programs. Parallel Comput. 31(10-12), 960–983 (2005)

[TTSY00] Tanaka, Y., Taura, K., Sato, M., Yonezawa, A.: Performance evaluation of openmp applications with nested parallelism. In: Languages, Compilers, and Run-Time Systems for Scalable Computers, pp. 100–112 (2000)