# Nested Parallelism in the OMPi OpenMP/C compiler

Panagiotis E. Hadjidoukas and Vassilios V. Dimakopoulos

Department of Computer Science
University of Ioannina, Ioannina, Greece
{phadjido,dimako}@cs.uoi.gr

**Abstract.** This paper presents a new version of the OMPi OpenMP C compiler, enhanced by lightweight runtime support based on user-level multithreading. A large number of threads can be spawned for a parallel region and multiple levels of parallelism are supported efficiently, without introducing additional overheads to the OpenMP library. Management of nested parallelism is based on an adaptive distribution scheme with hierarchical work stealing that not only favors computation and data locality but also maps directly to recent architectural developments in shared memory multiprocessors. A comparative performance evaluation of several OpenMP implementations demonstrates the efficiency of our approach.

## 1 Introduction

Although nested parallelism was defined from the initial version of OpenMP, several implementation and performance issues still remain open and need to be answered. This necessity stems from the fact that both applications and end-users require such functionality, and is further augmented as multi-core technology tends to increase the number of available processors. Nowadays, several research and commercial OpenMP compilers support more than one levels of parallelism. With a few exceptions, however, most OpenMP implementations have translated "OpenMP threads" to "kernel threads", which are internally mapped to system-scope POSIX threads or native OS threads. These implementations support nested parallelism by extending their highly-optimized and fine-tuned OpenMP runtime library for single-level parallelism.

Despite the fact that the above approach fulfills the requirements for the common case, namely single-level parallelism with dynamic threads (where the runtime system may adjust the number of working threads at will), it can cause serious performance degradation when the number of threads is explicitly requested by the user and exceeds the number of available processors. This excess is quite common in the case of nested parallelism or in multiprogramming (non-dedicated) environments.

This paper presents a new version of the OMPi OpenMP C compiler [3], enhanced by a lightweight thread library that provides efficient runtime support

for multiple levels of parallelism. The whole configuration results in significantly low parallelization overheads, especially when the number of OpenMP threads increases, and provides efficient support of nested parallelism. In addition, it simplifies thread management, favors computation and data locality and eliminates the disadvantages of time sharing, including the unavoidable synchronization overheads at the end of inner parallel regions.

The rest of this paper is organized as follows: Section 2 discusses related work. Sections 3 and 4 present the OMPi OpenMP compiler and the lightweight thread library respectively. Section 5 describes the management of nested parallelism. Experimental results are included in Section 6. Finally, Section 7 discusses our ongoing work.

## 2 Related Work

Several research efforts and compiler vendors support nested parallelism, creating a new team of OpenMP threads to execute nested parallel regions.

The NANOS compiler [1] supports nested regions and OpenMP extensions for processor groups. The runtime support of the NANOS Compiler is provided by an efficient user-level threads library (NthLib), which assumes that dynamic parallelism is always enabled and thus the number of spawned threads never exceeds that of available processors. The Omni compiler [13] supports a limited form of nested parallelism, requiring a user-predefined fixed size for the kernel thread pool, from where threads will be used for the execution of parallel regions. Omni/ST [14], an experimental version of Omni equipped with the StackThreads/MP library, provided an efficient though not portable implementation of nested irregular parallelism. The Balder runtime library of OdinMP [8] is capable of fully handing OpenMP 2.0 including nested parallelism. Balder uses POSIX threads as underlying thread library, provides efficient barrier and lock synchronization and uses a pool of threads which is expanded whenever it is necessary.

All vendors that support nested parallelism implement it by maintaining a pool of kernel threads. GOMP [11], the OpenMP implementation for GCC, implements its runtime library (libgomp) as a wrapper around the POSIX threads library, with some target-specific optimizations for systems that support lighter weight implementation of certain primitives. The GOMP runtime library allows the reuse of idle threads from their pool only for non-nested parallel regions, while threads are created dynamically for inner levels.

The OpenMP runtime library of the Sun Studio compiler [9] maintains a pool of threads that can be used as slave threads in parallel regions. The user can control both the number of threads in the pool and the maximum depth of nested parallel regions that require more than one thread. Similarly, the basic mechanism for threading support in the Intel compiler [16] is the thread pool. The threads are not created until the first parallel region is executed, and only as many as needed by that parallel region are created. Further threads are created as needed by subsequent parallel regions. Threads that are created by the OpenMP

runtime library are not destroyed but join the thread pool until they are called upon to join a team and are released by the master thread of the subsequent team.

The Fujitsu PRIMEPOWER Fortran compiler [7] also supports nested parallelism. Moreover, if the OpenMP application has only a single level of parallelism then a high performance OpenMP runtime library is used. Finally, the IBM XLC compilers support the execution of nested parallel loops [4].

## 3 The OMPi Compiler

OMPi is a source-to-source translator that takes as input C source code with OpenMP V.2.0 directives and outputs equivalent multithreaded C code, ready to be built and executed on a multiprocessor. The current version is fully V.2.0 compliant, can target different thread libraries through a unified thread abstraction, and includes many architecture-dependent as well as higher-level optimizations. Finally, the compiler and the runtime libraries include support for the POMP performance monitoring interface. OMPi is implemented entirely in C and has been ported effortlessly on many different platforms, including Intel / Linux, Sun / Solaris and SGI / Irix systems.

OMPi produces multithreaded C code. Its architecture is such that any specific thread library can be supported through a well-defined generic interface, making OMPi quite extensible. It currently supports a number of thread libraries through the generic thread interface. They include a POSIX threads based library (the default thread library target mainly for portability reasons) which is highly tuned for single-level parallelism. Another library can be used in Sun / Solaris machines, where the user has the option of producing code with Solaris threads calls. It should be noted that the generic interface provides for a transparent choice of the underlying thread library. That is, the user source code is not affected in any way—the actual thread library that will be used is only included at linking time.

## 4 Lightweight Runtime Support

The new internal threading interface in the runtime library of OMPi facilitates the integration of arbitrary thread libraries. In order to efficiently support nested parallelism, a user-level thread library, named `psthreads`, has been developed.

The `psthreads` library implements a two-level thread model, where user-level threads are executed on top of kernel-level threads that act as *virtual processors*. Each virtual processor runs a dispatch loop, selecting the next-to-run user-level thread from a set of ready queues, where threads are submitted for execution. An idle virtual processor extracts threads from the *front* of its local ready queue but steals from the *back* of remote queues. The queue architecture allows the runtime library to represent the layout of physical processors. For instance, a hierarchy can be defined in order to map the coupling of processing elements in current multi-core architectures [10].

Despite the user-level multithreading, the `psthreads` library is fully portable because its implementation is based entirely on the POSIX standard. Its virtual processors are mapped to POSIX threads, permitting the interoperability of OMPi with third-party libraries and the co-existence of OpenMP and POSIX threads in the same program. The primary user-level thread operations are provided by UthLib (Underlying Threads Library), a portable thread package. An underlying thread is actually the stack that a `psthread` uses during its execution. Synchronization is based on the POSIX threads interface. Locks are internally mapped to POSIX mutexes or spinlocks, taking into account the non-preemptive threads of the library. In addition, platform-dependent spin locks are utilized.

The application programming interface of `psthreads` is similar to that of POSIX threads. Its usage simplifies the OpenMP runtime library since spawning of threads is performed explicitly, while thread pooling is provided by the thread library. The thread creation routine of `psthreads` allows the user to specify the queue where the thread will be submitted for execution and whether it will be inserted in the front or in the back of the specified queue. Moreover, there exists a variant of the creation routine that accepts an already allocated thread descriptor. This is useful for cases where the user implements its own management of thread descriptors.

Efficient thread and stack management is essential for nested parallelism because a thread with a private stack should always be created since the runtime library cannot know whether the running application will spawn a new level of parallelism. An important feature of `psthreads` is the utilization of a lazy stack allocation policy. According to this policy, the stack of a user-level thread is allocated just before its execution. This results in minimal memory consumption and simplified thread migrations. Lazy stack allocation is further improved with stack handoff. In peer-to-peer scheduling, a finished thread picks the next descriptor, creates a stack for that thread and switches to it. Usually, this stack is extracted from a reuse queue. Using stack handoff, a finished thread re-initializes its own state, by replacing its descriptor with the following thread's descriptor, and resumes its execution. By allocating the descriptors from the stack of the parent thread (i.e. master), the activation of their recycling mechanism is also avoided.

If native POSIX thread libraries followed a hybrid (two-level or M:N) implementation, the runtime overheads would be reduced, allowing the creation of several threads without additional performance cost, as shown in [12]. However, all vendors have dropped the hybrid model and use a 1:1 mapping of POSIX threads to kernel threads. Fortunately, Marcel [6] is a two-level thread library that provides similar functionality and application programming interface to POSIX threads. Marcel binds one kernel-level thread on each processor and then performs fast user-level context-switches between user-level threads, hence getting complete control of thread scheduling in user-land without any further help from the kernel. We have successfully built and integrated a Marcel-based module into the OMPi compiler, by replacing the corresponding threading calls of the `psthreads` module.

## 5 Management of Nested Parallelism

According to OpenMP, when an application encounters the first of two nested parallel loops, a new team of threads is created and the loop iterations are distributed among these worker threads. Eventually, each thread will become the master of a new team that will be created for the execution of the inner loop. Assuming that the number of threads spawned in a parallel region is equal to the number ($P$) of processors, a kernel thread model will result in $P \times P$ threads that compete for hardware resources. Time-sharing can significantly increase implicit synchronization overheads that are related to thread management and dynamic loop scheduling. Even if static loop schedules are used, it is difficult for the runtime library to decide how to bind inner threads to specific processors in order to favor locality. A common approach that handles these problems uses a fixed size pool of threads, limiting thus the number of created threads. Another approach does not create additional threads but partitions the available threads into groups, based on information provided by the programmer [1] or extracted from the loop characteristics [4]. Despite the good locality and the low overheads of grouping, it is hard to determine the number of groups and their size and this can easily cause load imbalance.

We propose a straightforward approach able to handle general unstructured nested parallelism. Due to the lightweight runtime support of `psthreads`, OMPi can support efficiently a large number of threads and multiple levels of parallelism. Moreover, the utilization of non-preemptive threads allows the runtime library to manage parallelism explicitly, which is not possible for the case of kernel threads. Specifically, the OpenMP runtime library utilizes a variant of the all-to-all scheme in order to distribute work across the processors. Threads that are spawned at the first level of parallelism are distributed cyclically and inserted at the back of the ready queues. For inner levels, the threads are inserted in the front of the ready queue that belongs to the virtual processor they were created on. Since an idle virtual processor extracts threads from the front of its local queue and the back of the remote ones, this scheme favors the exploitation of data locality of inner levels of parallelism.

Our approach can be easily generalized to include the latest developments of shared memory architecture, like multi-core and SMT processors. The work stealing mechanism has been designed to work hierarchically, assuming the existence of thread groups, as shown in Fig. 1. Specifically, the virtual processors are organized into hierarchical groups of size that is equal to a power of 2 (i.e. 2, 4, 8, etc), according to the level of hierarchy. Thus, an idle virtual processor first examines the ready queue of its adjacent virtual processor in the two-processor group (level 1, size 2) where it belongs to, then it tests the queues of the rest two processors in the quad-processor group (level 2, size 4), etc. Moreover, due to our two-level thread model, there is a 1:1 mapping between virtual and physical processors and, thus, the queue hierarchy of the runtime library can be mapped directly to the hardware architecture.

The number of groups and their size can be set explicitly using an OpenMP extension (e.g. [1]) or appropriate machine description. In contrast to NANOS

```
GroupSize = 2;
while (GroupSize <= Virtual Processors) {
   GroupId = MyID / GroupSize;
   GroupBaseVP = GroupId*GroupSize;
   vp = (MyID + 1) % GroupSize + GroupBaseVP;
   while ((thread == NULL) && (vp != MyID)) {
      if (!visited[vp]) {
         visited[vp] = 1;
         thread = DequeueWork(vp);
      }
      vp = (vp + 1) % GroupSize + GroupBaseVP;
   }
   GroupSize *= 2;
}
/* execute thread */
```

**Fig. 1.** Hierarchical work-stealing algorithm for uniform thread groups

groups, our approach supports inter-group stealing and, based on these groups, determines how idle processors access ready queues. Although an idle processor in Omni/ST was able to issue work-stealing requests, the mechanism was heavyweight and lacked portability due to the required intervention of the remote processor. The dependence of Omni/ST on the StackThreads compiler, a patched version of GNU C, did not allow its further usage. On the contrary, work stealing in OMPi/`psthreads` is performed asynchronously and virtual processors access remote queues in a uniform way. Furthermore, our software configuration allows the integration of any native compiler. Similarly, the Marcel package utilizes the BubbleSched scheduler [15], a framework that allows the distribution of threads over the hierarchy of the computer so as to benefit from cache effects and to avoid NUMA factor penalties as much as possible. The integration of Marcel threads into OMPi will allow the exploitation of the above framework in an OpenMP environment. A further discussion on this issue is beyond the scope of this paper.

## 6 Experimental Results

We performed our experiments on a Compaq Proliant ML570 server with 4 Intel Xeon III CPUs running Debian Linux (2.6.6). We provide comparative performance results for the Intel C++ compiler (v. 9.1), GNU GCC 4.2, Omni 1.6 and the new version of OMPi. The native compiler for both OMPi and Omni is GNU GCC.

Our first experiment demonstrates our lightweight runtime support as the number of OpenMP threads increases. Specifically, we use the EPCC microbenchmarks [2] to measure the OpenMP runtime overheads using from 2 up to 32 threads on a dedicated machine.

Figures 2 presents the overheads of the `for` and lock/unlock OpenMP constructs. The overhead of `for` is lower if user-level thread libraries (marcel and

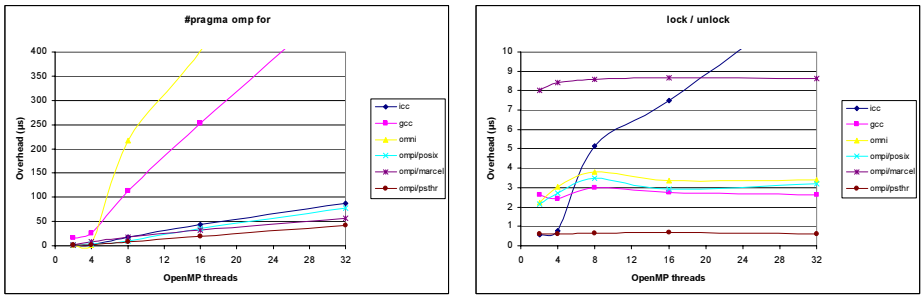


**Fig. 2.** Runtime overheads for large number of threads

`psthreads`) are used. The Intel compiler and the OMPi compiler with the native POSIX threads library also achieve good performance. On the contrary, both Omni and GCC exhibit significant runtime overhead as the number of OpenMP threads increases. The overhead of OpenMP locks does not depend on the number of threads for all the cases but the Intel compiler. This is attributed to the spinlock-based synchronization of the Intel compiler, which results in the lowest overhead for 2 and 4 OpenMP threads but fails to maintain stability when the number of threads exceeds the number of physical processors. To avoid this problem for Omni, its mutex-based lock primitives are used in all the experiments.

Our second experiment focuses on the evaluation of OpenMP runtime support for nested parallelism. For this purpose, we have appropriately extended the EPCC microbenchmarks. Specifically, the core benchmark routine for a given construct (e.g. `parallel`) is called several times within a parallel loop. If nested parallelism is tested, the loop is parallelized and these tasks are executed in parallel, otherwise they are executed sequentially as in the original version of the microbenchmarks. We measure the total execution time of the tasks and we compute the mean of the measured runtime overhead for each individual task.

```
omp_set_dynamic(0);
omp_set_nested(1);
t0 = omp_get_wtime();
#if defined(TEST_NESTED_PARALLELISM)
#pragma omp parallel for schedule(dynamic,1)
#endif
for (i = 0; i < ntasks; i++)
   testpr(i);

t1 = omp_get_wtime();
```

**Fig. 3.** Extended micro-benchmarks for nested parallelism

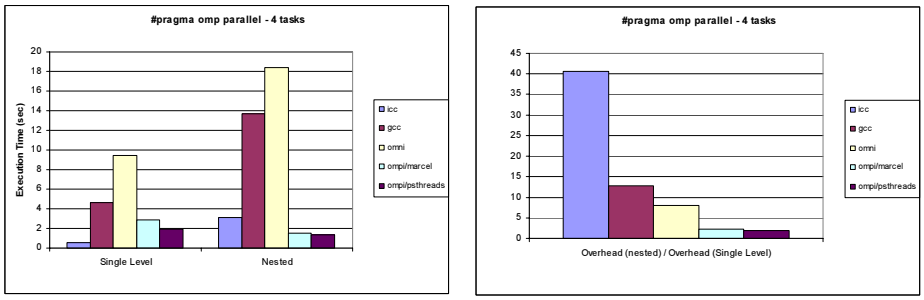


**Fig. 4.** Runtime behavior of the OpenMP `parallel` construct

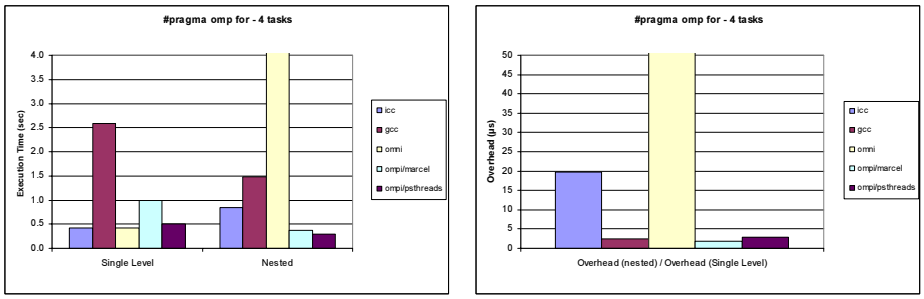


**Fig. 5.** Runtime behavior of the OpenMP `for` construct

Figures 4 and 5 present the runtime behavior of the `parallel` and `for` constructs respectively. The left figures show the total execution time and the right ones how many times the runtime overhead is increased when nested parallelism is enabled. For the `parallel` construct, both the execution time and the corresponding overhead are significantly increased for the Intel, GCC and Omni compilers. On the other hand, for both configurations of OMPi the total execution time is slightly decreased and the runtime overhead remains almost stable. This is expected because OMPi exploits better the task parallelism of the benchmark and restrains the contention between OpenMP threads. The `for` construct exhibits similar behavior except for the case of GCC, which does not suffer from high contention possibly due to its platform-specific atomic primitives.

The last experiment demonstrates our efficient support of nested parallelism using PCURE [5], an OpenMP implementation of a hierarchical data clustering algorithm. PCURE consists of an initialization phase and the core clustering algorithm, which is repeated until the requested number of clusters has been computed. The asymmetry and non-determinism of the algorithm requires the exploitation of nested parallelism, which is expressed with two parallel nested loops. The inner loop contains a reduction on a pair of data, an operation that is not directly supported by OpenMP. Therefore, a team of threads is spawned and each thread keeps the partial results (minimum distance and index) in its

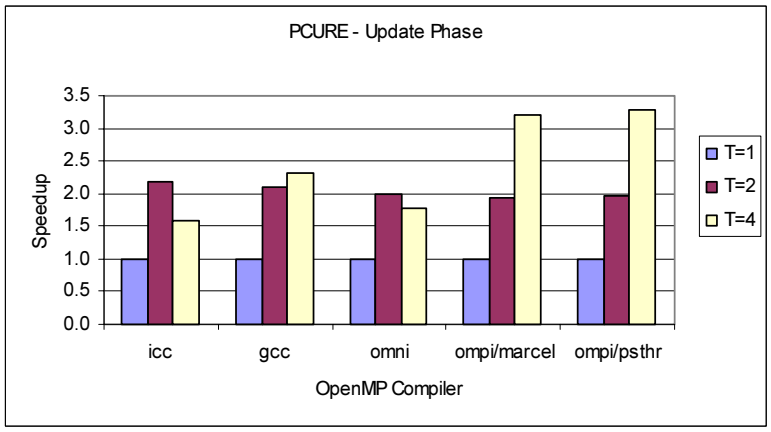


**Fig. 6.** Speedups of the parallel data clustering algorithm

local memory. When the iterations of the inner loop have been exhausted, the reduction operation takes place, with each thread checking and updating the global result within a critical section.

For both loops of the update phase, the guided scheduling policy is used. The input dataset contains 5000 records with 24 features and the clustering stops after 4000 steps. The performance speedups obtained for the update phase of PCURE are depicted in Figure 6. The Intel and Omni compilers perform best when 2 threads are used because the actual number of kernel threads from both levels of parallelism is equal to the number of physical processors (4). For 4 threads per level, the speedup drops mostly due to the higher contention of threads and the increased overheads of the runtime libraries. For GCC, however, the speedup is improved slightly when 4 threads are used. On the other hand, PCURE scales well for both OMPi configurations (marcel and `psthreads`). Let us note here that PCURE is a data intensive application so its scalability is limited by the low memory bandwidth of the bus-based SMP machine.

## 7 Conclusions and Future Work

This paper presented an OpenMP implementation based on user-level multithreading and its advantages on the exploitation of nested parallelism compared to the traditional kernel thread based approaches. Our ongoing work includes the implementation of the workqueuing model in the OMPi compiler and the evaluation of our approach for nested parallelism on large scale multiprocessor systems. The `psthreads` library has also been integrated into the Omni and GCC compilers, while our future plans include open-source OpenMP Fortran compilers. The source code of OMPi is available at `http://www.cs.uoi.gr/~ompi`.

## References


1. E. Ayguade, M. Gonzalez, X. Martorell, J. Labarta, N. Navarro, and J. Oliver. NanosCompiler: Supporting Flexible Multilevel Parallelism in OpenMP. *Concurrency: Practice and Experience*, 12(12):1205–1218, October 2000.
2. J. M. Bull. Measuring Synchronization and Scheduling Overheads in OpenMP. In *Proc. of the 1st European Workshop on OpenMP (EWOMP '99)*, Lund, Sweden, September 1999.
3. V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas. A Portable C Compiler for OpenMP V.2.0. In *Proc. of the 5th European Workshop on OpenMP (EWOMP '03)*, Aachen, Germany, October 2003.
4. A. Duran, R. Silvera, J. Corbalan, and J. Labart. Runtime Adjustment of Parallel Nested Loops. In *Proc. of the International Workshop on OpenMP Applications and Tools (WOMPAT '04)*, Houston, TX, USA, May 2004.
5. P. E. Hadjidoukas and L. Amsaleg. Parallelization of a Hierarchical Data Clustering Algorithm Using OpenMP. In *Proc. the 2nd International Workshop on OpenMP (IWOMP '06)*, Reims, France, June 2006.
6. Team RUNTIME INRIA. Marcel: A POSIX-compliant thread library for hierarchical multiprocessor machines. Available at: `http://runtime.futurs.inria.fr/marcel`.
7. H. Iwashita, M. Kaneko, M. Aoki, K. Hotta, and M. van Waveren. On the Implementation of OpenMP 2.0 Extensions in the Fujitsu PRIMEPOWER compiler. In *Proc. of the International Workshop on OpenMP: Experiences and Implementations (WOMPEI '03)*, Tokyo, Japan, November 2003.
8. S. Karlsson. A Portable and Efficient Thread Library for OpenMP. In *Proc. of the 6th European Workshop on OpenMP (EWOMP '04)*, Stockholm, Sweden, October 2004.
9. Sun Microsystems. Sun Studio 10: OpenMP API User's Guide. Available at: `http://docs.sun.com/app/docs/doc/819-0501`.
10. D. S. Nikolopoulos, E. D. Polychronopoulos, and T. S. Papatheodorou. Efficient Runtime Thread Management for the Nanothreads Programming Model. In *Proc. of the 2nd IEEE IPPS/SPDP Workshop on Runtime Systems for Parallel Programming*, volume 1388, pages 183–194, Orlando, FL, USA, April 1998.
11. D. Novillo. OpenMP and automatic parallelization in GCC. In *Proc. of the 2006 GCC Summit*, Ottawa, Canada, June 2006.
12. R. Rufai, M. Bozyigit, J. Alghamdi, and M. Ahmed. Multithreaded Parallelism with OpenMP. *Parallel Processing Letters*, 15(4):367–378, 2005.
13. M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *Proc. of the 1st European Workshop on OpenMP (EWOMP '99)*, Lund, Sweden, September 1999.
14. Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance Evaluation of OpenMP Applications with Nested Parallelism. In *Proc. of the Fifth Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR '00)*, Rochester, NY, USA, May 2000.
15. S. Thibault. A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines. In *Proc. of the 2nd International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, Cambridge, USA, June 2005.
16. X. Tian, J. P. Hoeflinger, G. Haab, Y-K Chen, M. Girkar, and S. Shah. A compiler for exploiting nested parallelism in OpenMP programs. *Parallel Computing*, 31:960–983, 2005.