Support and Efficiency of Nested Parallelism in OpenMP Implementations

Panagiotis E. Hadjidoukas and Vassilios V. Dimakopoulos
Department of Computer Science
University of Ioannina
Ioannina, Greece, GR-45110

Abstract

Nested parallelism has been a major feature of OpenMP since its very beginnings. As a programming style, it provides an elegant solution for a wide class of parallel applications, with the potential to achieve substantial utilization of the available computational resources, in situations where outer-loop parallelism simply can not. Notwithstanding its significance, nested parallelism support was slow to find its way into OpenMP implementations, commercial and research ones alike. Even nowadays, the level of support is varying greatly among compilers and runtime systems.

In this work, we take a closer look at OpenMP implementations with respect to their level of support for nested parallelism. We classify them into three broad categories: those that provide full support, those that provide partial support and those that provide no support at all. The systems surveyed include commercial and research ones. Additionally, we proceed to quantify the efficiency of the implementation. With a representative set of compilers that provide adequate support, we perform a comparative performance evaluation. We evaluate both the incurred overheads and their overall behavior, using microbenchmarks and a full-fledged application. The results are interesting because they show that full support of nested parallelism does not necessarily guarantee scalable performance. Among our findings is the fact that most compilers do not seem to handle nested parallelism in a predictable and stable way as the number of threads increases beyond the system's processor count.

1. Introduction

OpenMP [27] has become a standard paradigm for programming symmetric shared memory multiprocessors (SMP), as it offers the advantage of simple and incremental parallel program development, in a high abstraction level. Its usage is continuously increasing as small SMP machines have become the mainstream architecture even in the personal computer market. Nested parallelism, a major feature of OpenMP, has the potential of benefiting a broad class of parallel applications to achieve optimal load balance and speedup, by allowing multiple levels of parallelism to be active simultaneously. This is particularly relevant these days in emerging SMP environments with multicore processors.

For applications that have enough (and balanced) outer-loop parallelism, a small number of coarse threads is usually enough to produce satisfactory speedups. In many other cases though, including situations with multiple nested loops, or recursive and irregular parallel applications, threads should be able to create new teams of threads because only a large number of threads has the potential to achieve good utilization of the computational resources.

Although many OpenMP compilation systems provide support for nested parallelism, there has been no comprehensive study up to now regarding the level and the efficiency of such a support. It can be seen that the existing OpenMP implementations exhibit significant runtime overheads which are mainly due to their adopted kernel-level threading model. In general, there are several design issues and performance limitations that need to be addressed effectively.

This work discusses the main issues that are related to the runtime support of nested parallelism in OpenMP. We survey existing implementations and discuss the type of support they offer. Based on previous knowledge and infrastructure, we also develop a set of appropriate microbenchmarks that measure the runtime overheads of OpenMP when nested parallelism is exploited. A comparative performance evaluation assesses the advantages and limitations of several OpenMP implementations, both commercial and research ones.

The rest of this chapter is organized as follows: in Section 2. we discuss nested parallelism in general. We also present the mechanisms OpenMP provides for controlling nested parallelism. In Section 3. we survey existing OpenMP implementations and discuss the level of nested parallelism support they provide; both commercial and research compilers are examined. Section 4. is devoted to the evaluation of the performance of a number of compilers when it comes to nested parallelism, while Section 5. summarizes the results of this study.

2. Nested Parallelism

Nested parallelism is becoming increasingly important as it enables the programmer to express a wide class of parallel algorithms in a natural way and to exploit efficiently both task and loop-level parallelism. Several studies with production codes and application kernels have shown the significant performance improvements of using nested parallelism. Despite some deficiencies in the current support of nested parallelization, many OpenMP applications have benefited from it and managed to increase their scalability mainly on large SMP machines [1, 2, 5, 31].

2.1. What OpenMP Specifies

Nested parallelism in OpenMP is effected either by a nested `parallel` construct (i.e. a `parallel` region within the lexical extend of another `parallel` region) or by an *orphaned* construct, where a `parallel` region appears in a function which is called within the dynamic extend of another `parallel` region in the program.

In Fig. 1 the classic example of Fibonacci numbers is shown; the nth Fibonacci number is calculated recursively as the sum the $(n-1)$th and the $(n-2)$th, through an orphaned `parallel` construct. In each recursive call, two threads are spawned resulting in a exponential total population. Recursion-based nested parallelism is an elegant programming methodology but can easily lead to an overwhelming number of threads.

```
int fib(int n)
{
  int f1,f2;
  if (n < 2) return 1;
  #pragma omp parallel sections num_threads(2)
  {
    #pragma omp section
      f1 = fib(n-1);             /* Recursive call */
    #pragma omp section
      f2 = fib(n-2);             /* Recursive call */
  }
  return f1+f2;
}
```

Figure 1. Fibonacci numbers using nested parallelism.

The OpenMP specifications leave support for nested parallelism as optional. Implementations are considered compliant even if they don't support nested parallelism; they are allowed to execute the nested `parallel` region a thread encounters by a team of just 1 thread, i.e. nested `parallel` regions may be serialized. Because of the difficulty in handling a possibly huge number of threads, many implementations provide support for nested parallelism but with certain limitations. For example, there may exist an upper limit on the depth of nesting or on the total number of simultaneously active threads.

Nested parallelism can be enabled or disabled either during program startup through the `OMP_NESTED` environment variable or dynamically (any time at runtime) through an `omp_set_nested()` call. The `omp_get_nested()` call queries the runtime system whether nested parallelism is enabled or not. In runtime systems that do not support nested parallelism, enabling or disabling it has no effect whatsoever.

To control the number of threads that will comprise a team, the current version of OpenMP (2.5) provides the following mechanism: the default number of threads per `parallel` region is specified through the `OMP_NUM_THREADS` environment variable. In the absence of such a variable, the default is implementation dependent. The `omp_set_num_threads()` call can be used at runtime to set the number of threads that will be utilized in subsequent `parallel` regions. Finally, a `num_threads(n)` clause that appears in a `parallel` directive requests this particular region to be executed by `n` threads.

However, the actual number of threads dispatched in a `parallel` region depends also on other things. OpenMP provides a mechanism for the *dynamic adjustment* of the number of threads which, if activated, allows the implementation to spawn fewer threads than what is specified by the user. In addition to dynamic adjustment, factors that may affect the actual number of threads include the nesting level of the region, the support/activation of nested parallelism and the peculiarities of the implementation. For example, some systems maintain a fixed pool of threads, usually equal in size to the number of available processors. Nested parallelism is supported as long as free threads exist in the pool, otherwise it is dynamically disabled. As a result, a nested `parallel` region may be executed by a varying number of threads, depending on the current state of the pool.

OpenMP has been architected mainly with the first (outer) level of parallelism in mind and certain features are not thoroughly thought out. For one, in the first version of the specifications [24, 25] there was no way to control the number of threads in inner levels, since `omp_set_num_threads()` can only be called from the sequential parts of the code; the `num_threads` clause was added in the second version of OpenMP [26] to overcome this limitation. There are also a few parts in the specifications which are unclear when applied to nested levels, e.g. the exact meaning of the persistence of threadprivate variable values between `parallel` regions. Finally, there are some features that are completely lacking; ancestor-descendant interaction and intra-team shared variables are only two of them.

The upcoming version of the OpenMP specifications is expected to clarify some ambiguities and provide a richer functional API for nested parallelism; [7] discloses the following self-explanatory calls:

```
omp_get_nested_level()
omp_set_max_nested_levels()
omp_get_max_nested_levels()
omp_set_thread_limit()
omp_get_thread_limit()
```

while it will be possible to call `omp_set_num_threads()` from within `parallel` regions, so as to control the number of threads for the next nesting level. There seem to also exist provisions for identifying one's parent (i.e. the master thread of a team), as well as any other ancestor.

3. Support in Compilers

According to the OpenMP specifications, an implementation which serializes nested `parallel` regions, even if nested parallelism is enabled through the `OMP_NESTED` environment variable or the `omp_set_nested()` call, is considered *compliant*. An implementation can claim *support* of nested parallelism if nested `parallel` regions (i.e. at levels greater than 1) may be executed by more than 1 thread. Nowadays, several commercial and research compilers support nested parallelism, either fully or partially. Partial support implies that there exists some kind of limit imposed by the implementation. For example, there exist systems that support a fixed number of nesting levels; some others allow an unlimited number of nested levels but have a fixed number of simultaneously active threads. In the latter case, a nested `parallel` region may be executed by a smaller number of threads than the one requested, if there are not enough free threads. The decision of the team size is made at the beginning of a `parallel` region. If later, however, some threads become idle, they will not be able to participate in the parallel execution of that region.

Notice that a nested `parallel` region which includes a `num_threads(n)` clause must be executed by exactly `n` threads, as requested, unless dynamic adjustment of the number of threads is turned on. Thus, those systems that limit the number of simultaneously active threads cannot be used with a disabled dynamic adjustment. Consequently, we consider an implementation as providing *full support* if it imposes no limit on the number of nesting levels or the number of simultaneously active threads and does not serialize nested regions when dynamic adjustment of threads is disabled.

The majority of OpenMP implementations instantiate their OpenMP threads with kernel-level threads, utilizing either the POSIX-threads API or the native threads provided by the operating system. The utilization of kernel threads limits the size of the thread pool and consequently the maximum number of OpenMP threads. In addition, it introduces significant overheads in the runtime library, especially if multiple levels of parallelism are exploited. When the number of threads that compete for hardware recourses significantly exceeds the number of available processors, the system is overloaded and the parallelization overheads outweigh any performance benefits. Finally, it becomes quite difficult for the runtime system to decide the distribution of inner-level threads to specific processors in order to favor computation and data locality.

3.1. Proprietary Compilers

Not all proprietary compilers support nested parallelism and some support it only in part. Below, we provide a summary of the level of support in various known systems.

- **Fujitsu**: The Fujitsu PRIMEPOWER compilers in the Parallelnavi software package [19] support nesting of `parallel` regions. Moreover, a high performance OpenMP runtime library is available for OpenMP applications with single-level parallelism.

- **HP**: The **HP** compilers for the HP-UX 11i operating system support dynamically nested parallelism [16]. When nested parallelism is enabled, the number of threads used to execute nested `parallel` regions is determined at runtime by the underlying OpenMP runtime library. The maximum number of threads is dependent upon the load on the system, the amount of memory allocated by the program and the amount of implementation dependent stack space allocated to each thread. The latest releases of HP-UX contain new thread functionality, providing the possibility for multiple POSIX threads of a process to map to a smaller number of kernel threads. Therefore, the OpenMP runtime library takes advantage of the hybrid (M:N) threading model.

- **Intel**: The basic mechanism for threading support in the Intel compilers [36] 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. Additional threads are created as needed by subsequent `parallel` regions. Threads that are created by the OpenMP runtime library are not destroyed. Instead, they join the thread pool until they are called upon to join a team and are released by the master thread of the subsequent team. Since the Intel compiler maps OpenMP threads to kernel threads, its runtime library uses the `KMP_MAX_THREADS` environment variable to set the maximum number of threads that it will use. This gives the user the freedom to utilize any number of threads or to limit them to the number of physical processors, so that an application or a library used by an application does not oversubscribe the system with OpenMP threads. The library will attempt to use as many threads as requested at every level, until the `KMP_MAX_THREADS` limit is reached.

- **Microsoft**: Visual C++ 2005 provides a new compiler switch that enables the compiler to understand OpenMP directives [11]. Visual C++ allows nested `parallel` regions, where each thread of the original `parallel` region becomes the master of its own thread team. Nested parallelism can continue to further nest other `parallel` regions. This process of creating threads for each nested `parallel` region can continue until the program runs out of stack space.

- **Sun**: The OpenMP runtime library of the Sun Studio compilers [30] maintains a pool of threads that can be used as slave threads in `parallel` regions. When a thread encounters a `parallel` construct and needs to create a team of more than one thread, the thread will check the pool and grab idle threads from the pool, making them slave threads of the team. The master thread might get fewer slave threads than it needs if there is not a sufficient number of idle threads in the pool. When the team finishes executing the `parallel` region, the slave threads return to the pool. The user can control both the number of threads in the pool and the maximum depth of nested `parallel` regions that may utilize more than one thread. This is performed through the `SUNW_MP_MAX_POOL_THREADS` and `SUNW_MP_MAX_NESTED_LEVELS` environment variables respectively.

Full support for nested parallelism is also provided in the latest version of the well known open-source GNU Compiler Collection, **GCC 4.2**. libGOMP [23], the runtime library of the system is designed as a wrapper around the POSIX threads library, with some target-specific optimizations for systems that provide lighter weight implementation of certain primitives. The GOMP runtime library allows the reuse of idle threads from a pre-built pool only for non-nested `parallel` regions, while threads are created dynamically for inner levels.

In contrast to the aforementioned cases, several other OpenMP compiler vendors do not currently support nested parallelism. The **MIPSpro** compiler and runtime environment on the SGI Origin does not support nested parallelism. Instead, it supports multi-loop parallelization for loops that are perfectly nested [35]. The current implementation of the **IBM XL** compiler does not provide true nested parallelism. Instead of creating a new team of threads for nested `parallel` regions, the OpenMP threads that are currently available are re-used [17,18,22]. The **PathScale** Compiler Suite is a family of compilers for the AMD64 processor family, compatible with the latest specification of the OpenMP programming model (2.5). Its Fortran version supports nested parallelism [28], as long as the number of threads does not exceed 256 and `parallel` regions are not lexically nested; nested regions should be orphaned (i.e. appear in different subroutines). Finally, the **Portland Group** (PGI) compilers do not support and thus ignore nested OpenMP `parallel` constructs.

The `omp_set_nested()` function, which allows enabling and disabling of nested `parallel` regions, has currently no effect [33].

3.2. Research and Experimental Compilers

Most research OpenMP compilers are source-to-source compilation systems that transform OpenMP-annotated source code (C/Fortran) to equivalent multithreaded code in the same base language, ready to be compiled by the native compiler of the platform. The code also includes calls to a runtime library that supports and controls the execution of the program. Thread creation is based mostly on the *outlining* technique [8], where the code inside a `parallel` region is moved to a separate function or routine, which is executed by the spawned threads.

The next four experimental systems provide full support of nested parallelism through the use of the appropriate runtime libraries.

- **GOMP/Marcel**: MaGOMP [34] is a port of GOMP on top of the Marcel threading library [32] in which BubbleSched, an efficient scheduler for nested parallelism, is implemented. More specifically, a Marcel adaptation of libGOMP threads has been added to the existing abstraction layer. MaGOMP relies on Marcel 's fully POSIX compatible interface to guarantee that it will behave as well as GOMP on POSIX threads. Then, it becomes possible to run any existing OpenMP application on top of BubbleSched by simply relinking it.

- **OdinMP**: The Balder runtime library of OdinMP [20] is capable of fully handing OpenMP 2.0 including nested parallelism. Balder uses POSIX threads as its underlying thread library, provides efficient barrier and lock synchronization, and uses a pool of threads, which is expanded whenever it is necessary.

- **Omni**: The Omni compiler [29] supports full nested parallelism but requires a user-predefined fixed size for its kernel thread pool, where threads for the execution of `parallel` regions are extracted from. Specifically, an OpenMP program creates a fixed number of worker threads at the beginning of its execution and keeps a pool of idle threads. Whenever the program encounters a `parallel` construct, it is parallelized if there are idle threads at that moment. Omni/ST [31], an experimental version of Omni equipped with the StackThreads/MP library, provided an efficient though not portable implementation of nested irregular parallelism.

- **ompi**: OMPi [10] is a source-to-source translator that takes as input C source code with OpenMP directives and outputs equivalent multithreaded C code, ready to be built and executed on a multiprocessor. It has been enhanced with lightweight runtime support based on user-level multithreading. A large number of threads can be spawned for every `parallel` region and multiple levels of parallelism are supported efficiently, without introducing additional overheads to the OpenMP library. A more detailed description of OMPi is provided in section 3.3..

There also exist other experimental compilation systems, some of them being quite advanced, which either provide a limited form of nested parallelism or do not support it at all.

- **CCRG**: CCRG OpenMP Compiler [9] aims to create a freely available, fully functional and portable set of implementations of the OpenMP Fortran specification for a variety of different platforms, such as SMPs as well as Software Distributed Shared Memory (sDSM) systems. CCRG uses the approach of the source-to-source translation and runtime support to implement OpenMP. The CCRG OpenMP Compiler fully implemented OpenMP 1.0 and in part features of OpenMP 2.0 Fortran API. Its runtime library for SMP is based on the standard POSIX threads interface.

- **NanosCompiler**: The NanosCompiler [3] for Fortran does not fully support nested parallelism. Instead, it supports multilevel parallelization based on the concept of thread groups, without allowing the total number of threads to exceed that of available processors. A group of threads is composed of a subset of the total number of threads available in the team to run a `parallel` construct. In a `parallel` construct, the programmer may define the number of groups and the composition of each one. When a `parallel` directive defining groups is encountered, a new team of threads is created. The new team is composed of as many threads as the number of groups. The rest of the threads are used to support the execution of nested `parallel` constructs. In other words, the definition of groups establishes an allocation strategy for the inner levels of parallelism. To define groups of threads, NanosCompiler supports the GROUPS clause extension to the `parallel` directive. The NanosCompiler has been recently replaced by its successor, the Nanos Mercurium compiler.

- **Nanos Mercurium**: The objective of the Nanos Mercurium compiler [4] is to offer a compilation platform that OpenMP researchers can use to test new language features. It is build on top of an existing compilation platform, the Open64 compiler, and uses templates of code for specifying the transformations of the OpenMP directives. The compiler implements most of OpenMP 2.0 along with extensions such as dynamic sections, a relaxation of the current definition of SECTIONS, to allow parallelization of programs that use iterative structures (such as while loops) or recursion. Since it is based on the runtime library of the NanosCompiler, Nanos Mercurium does not fully support nested parallelism.

- **OpenUH**: OpenUH [21] is a portable OpenMP compiler based on the Open64 compiler infrastructure with a unique hybrid design that combines a state-of-the-art optimizing infrastructure with a source-to-source approach. OpenUH is open source, supports C/C++/Fortran 90, includes numerous analysis and optimization components, and is a complete implementation of OpenMP 2.5. The thread creation transformations used in OpenUH are different from the standard outlining approach; the approach is similar to the MET (Multi-Entry Threading) technique employed in the Intel OpenMP compiler. The compiler generates a microtask to encapsulate the code lexically contained within a `parallel` region and the microtask is nested into the original function containing that `parallel` region. OpenUH does not support nested parallelism.

The above discussion is summarized in Table 1, where we list all the aforementioned compilers, their present status and the level of support for nested parallelism they provide.

Table 1. OpenMP implementations.

Entries marked with a star (⋆) represent compiler projects that seem to be dormant.

OpenMP compiler	**Nested parallelism**	**Availability**
CCRG 1.0 (Fortran)	no	freeware (⋆)
GCC 4.2.0	yes	freeware
Fujitsu	yes	commercial
HP (for HP-UX 11i)	yes	commercial
IBM XL (C v9.0, F v11.1)	no	commercial
Intel 10.0	yes	commercial, free
MaGOMP (GCC 4.2.0)	yes	freeware
Microsoft Visual C++ 2005	yes	commercial
Nanos Mercurium 1.2	limited	freeware
NanosCompiler (Fortran)	limited	freeware (⋆)
OdinMP 0.287.2 (C only)	yes	freeware (⋆)
Omni 1.6	yes	freeware (⋆)
OMPi 0.9.0 (C only)	yes	freeware
OpenUH alpha (Fortran)	no	freeware
PathScale 3.0	limited	commercial
PGI 7.0.6	no	commercial
SGI MIPSpro 7.4	no	commercial
Sun Studio 12	yes	commercial, free

3.3. OMPi

OMPi's runtime system has been architected with an internal threading interface that facilitates the integration of arbitrary thread libraries. It comes with two core libraries that are based on POSIX threads; one is optimized for single-level (non-nested) parallelism, while the other provides limited nested parallelism support through a fixed pool of threads (the size of which is determined at startup through the `OMP_NUM_THREADS` environment variable). In order to efficiently support unlimited nested parallelism, an additional library based on user-level threads has been developed [15], named `psthreads`.

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. 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 multicore architectures.

Although user-level multithreading has traditionally implied machine dependence, the `psthreads` library is completely 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 coexistence of OpenMP and POSIX threads in the same program. The primary user-level thread operations are provided by UthLib (Underlying Threads Library), a platform-independent 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.

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 a priori 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, whereby a finished thread re-initializes its own state by replacing its descriptor with the subsequent thread's descriptor and resumes its execution.

In the `psthreads` library, an idle virtual processor extracts threads from the *front* of its local ready queue but steals from the *back* of remote queues. This allows the OpenMP runtime library to employ an adaptive work distribution scheme for the management of nested parallelism. In particular, 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. The adopted scheme favors the execution of inner threads on a single processor and improves data locality.

4. A Comparative Evaluation

In this section, we evaluate an implementation's efficiency with respect to nested parallelism. We measure both the incurred overheads and the overall behavior of a representative set of compilers, using microbenchmarks and an application that makes substantial use of nested parallelism. We performed all 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 two free commercial and three freeware OpenMP C compilers that fully support nested parallelism. The commercial compilers are the Intel C++ 10.0 compiler (ICC) and Sun Studio 12 (SUNCC) for Linux. The freeware ones are GNU GCC 4.2.0, Omni 1.6 and OMPi 0.9.0. We have used GCC as native back-end compiler for both OMPi and Omni.

4.1. Overheads

The EPCC microbenchmark suite [6] is the most commonly used tool for measuring runtime overheads of individual OpenMP constructs. This section describes the extensions we have introduced to the EPCC microbenchmark suite for the evaluation of OpenMP runtime support for nested parallelism.

Synchronization and loop scheduling operations can all be significant sources of overhead in shared memory parallel programs. The technique used by the EPCC microbenchmarks to measure the overhead of OpenMP directives, is to compare the time taken for a section of code executed sequentially with the time taken for the same code executed in parallel enclosed in a given directive. A full description of this method is given in [6]. To obtain statistically meaningful results, each overhead measurement is repeated several times and the mean and standard deviation are computed over all measurements. This way, the microbenchmark suite neither requires exclusive access to a given machine nor is seriously affected by background processes in the system.

To study how efficiently OpenMP implementations support nested parallelism, we have extended both the synchronization and the scheduling microbenchmarks. According to our approach, the core benchmark routine for a given construct is represented by a task. Each task has a unique identifier and utilizes its own memory space for storing the runtime measurements, i.e. its mean overhead time and standard deviation. When all tasks finish, we measure their total execution time and compute the global mean of all measured runtime overheads. Our approach, as applied to the synchronization benchmark, is outlined in Fig. 2. The loop that issues the tasks expresses the outer level of parallelism, while each benchmark routine includes the inner one.

If the outer loop is not parallelized, the tasks are executed in sequential order. This actually corresponds to the original version of the microbenchmarks, having each core benchmark repeated more than once. On the other hand, if nested parallelism is evaluated, the loop is parallelized and the tasks are executed in parallel. The number of simultaneously active tasks is bound by the number of OpenMP threads that constitute the team of the first level of parallelism. To ensure that the OpenMP runtime library does not assign fewer threads to inner levels than in the outer one, dynamic adjustment of threads is disabled.

By measuring the aggregated execution time of the tasks, we use the microbenchmark as an individual application. This time includes not only the parallel portion of the tasks, i.e. the time the tasks spend on measuring the runtime overhead, but also their sequential portion. This means that even if the mean overhead increases when tasks are executed in

```
void test_nested_bench(char *name, func_t f)
{
  int task_id;
  double t0, t1;

  t0 = getclock();
  #ifdef NESTED_PARALLELISM
  #pragma omp parallel for schedule(static,1)
  #endif
  for (task_id = 0; task_id < NTASKS; task_id++) {
    (*f)(task_id);
  }
  t1 = getclock();

  <compute global mean time and standard deviation>
  <print construct name, elapsed time (t1-t0) and mean values>
}

main ()
{
  <compute reference time>
  test_nested_bench("PARALLEL", testpr);
  test_nested_bench("FOR", testfor);
  ...
}
```

Figure 2. Outline of the extended EPCC microbenchmarks for nested parallelism.

parallel, as expected due to the higher number of running threads, the overall execution time may decrease.

In OpenMP implementations that provide full nested parallelism support, inner levels spawn more threads than the number of physical processors, which are mostly kernel-level threads. Thus, measurements exhibit higher variations than in the case of single-level parallelism. To resolve this issue, we increase the number of internal repetitions for each microbenchmark, so as to achieve the same confidence levels.

4.2. A Data clustering Application

Except overheads, we evaluate nested parallelism using a full application. For our purpose we have chosen PCURE (Parallel Clustering Using REpresentatives) [14], the OpenMP implementation of a well-known hierarchical data clustering algorithm (CURE). Data clustering is one of the fundamental techniques in scientific data analysis and data mining. The problem of clustering is to partition a data set into a number of segments (called clusters) that contain similar data. CURE [12] is a very efficient clustering algorithm with respect to the quality of clusters because it identifies arbitrary-shaped clusters and handles high-dimensional data. However, its worst-case time complexity is $O(n^2 logn)$, where n is the number of data points to be clustered. The OpenMP parallelization of CURE copes with the quadratic time complexity of the algorithm and allows for efficient clustering of very large data sets.

```
1. Initialization: Compute distances and find nearest neighbors
   pairs for all clusters

2. Clustering: Perform hierarchical clustering until the
   predefined number of clusters k has been computed

   while (number of remaining clusters > k) {
     a.  Find the pair of clusters with the minimum distance
     b.  Merge them
           i. new_size = size1 + size2
          ii. new_centroid = a1*centroid1 + a2*centroid2,
                where a1 = size1/new_size and a2 = size2/new_size
         iii. find r new representative points
     c.  Update the nearest neighbors pairs of the clusters
     d.  Reduce the number of remaining clusters
     e.  If conditions are satisfied, apply pruning of clusters
   }

3. Output the representative points of each cluster
```

Figure 3. Outline of the CURE data clustering algorithm.

Fig. 3 outlines the main clustering algorithm: since CURE is a hierarchical agglomerative algorithm, every data point is initially considered as a separate cluster with one representative, the point itself. The algorithm initially computes the closest cluster for each cluster. Next, it starts the agglomerative clustering, merging the closest pair of clusters until only k clusters remain. According to the merge procedure, the centroid of the new cluster is the weighted mean of the two merged clusters. Moreover, the new r representative points are chosen between the $2r$ points of the two merged clusters.

Fig. 4 presents using pseudocode the most computation demanding routine in the clustering phase, which includes the update of the nearest neighbors (`update_nnbs()`, lines 1–10). The parallelism in PCURE is expressed with the two parallel nested loops, found at lines 3 and 17 respectively. For a given cluster with index i, the algorithm finds its closest cluster among those with smaller index ($j < i$). Moreover, as the algorithm evolves, the number of valid clusters gradually decreases. Therefore, the computational cost of loop iterations cannot be estimated in advance.

The efficiency of PCURE strongly depends on the even distribution of computations to processors. Due to the highly irregular clustering algorithm, such distribution is not straightforward though. As shown in [13], the algorithm scales efficiently only if nested parallelism is exploited.

4.3. Experimental Results

Synchronization overheads Our first experiment uses the extended synchronization benchmark to measure the overhead incurred by the `parallel` and `for` constructs. Both

```
void update_nnbs(int pair_low, int pair_high)
{
  for (i=pair_low+1; i<npat; i++)
  {
    if (entry #i has been invalidated) continue;
    if (entry #i had neighbor pair_low or pair_high)
      find_nnb (i, &nnb[i].index, &nnb[i].dist);
    else if (pair_high < i)
      if ((dist = compute_distance(pair_high, i))) < nnb_dist[i])
        nnb[i].index = pair_high, nnb[i].dist = dist;
  }
}

void find_nnb(int i, int *index, double *distance)
{
  min_dist = +inf, min_index = -1;
  for (j=0; j<i; j++)
  {
    if (entry #j has been invalidated) continue;
    if ((dist = compute_distance(i, j)) < min_dist)
      min_dist = dist, min_index = j;
  }
  *index = min_index; *distance = min_dist;
}
```

Figure 4. Pseudocode for the update of the nearest neighbors.

the `OMP_NUM_THREADS` environment variable and the number of tasks are equal to the number of physical processors in the system (4). Fig. 5 and 6 present the measured overhead for both constructs respectively. As the number of active threads increases when nested parallelism is enabled, the overheads are expected to increase accordingly. We observe, however, that the `parallel` construct does not scale well for the Intel, GCC and Omni compilers. For all three of them, the runtime overhead is an order of magnitude higher in the case of nested parallelism. On the other hand, both OMPi and SUNCCclearly scale better and their overheads increase linearly. SUNCC, however, exhibits higher overheads than OMPi for both single level and nested parallelism. The `for` construct (Fig. 6) behaves similarly bad except for the case of GCC, shows significant but not excessive increase; this is attributed to the platform-specific atomic primitives that GCC uses. Although OMPi does not currently use atomic operations, it manages to deliver the best performance for both microbenchmarks mainly due to its lower contention between OpenMP threads.

Scheduling overheads In the second experiment, we use the loop scheduling benchmark to study the efficiency of OpenMP when several independent parallel loops are executed concurrently. We provide measurements for the dynamic and guided scheduling policies using their default chunk size, which is equal to one. This chunk size was chosen in order to measure the highest possible scheduling overhead. As shown in Fig. 7, the overhead of the dynamic scheduling policy increases substantially for the Intel and Omni compilers and

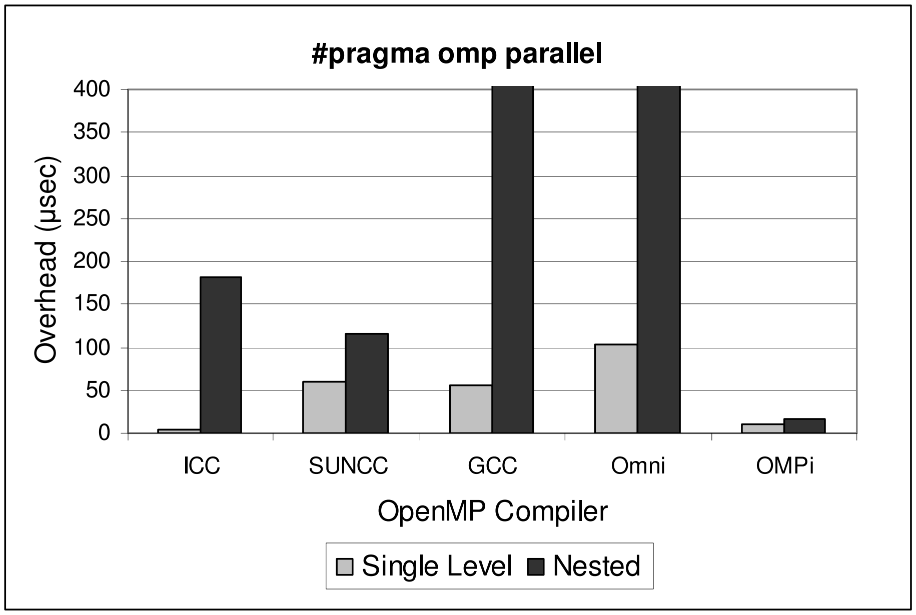

Figure 5. Runtime overhead of the OpenMP `parallel` construct (μsec).

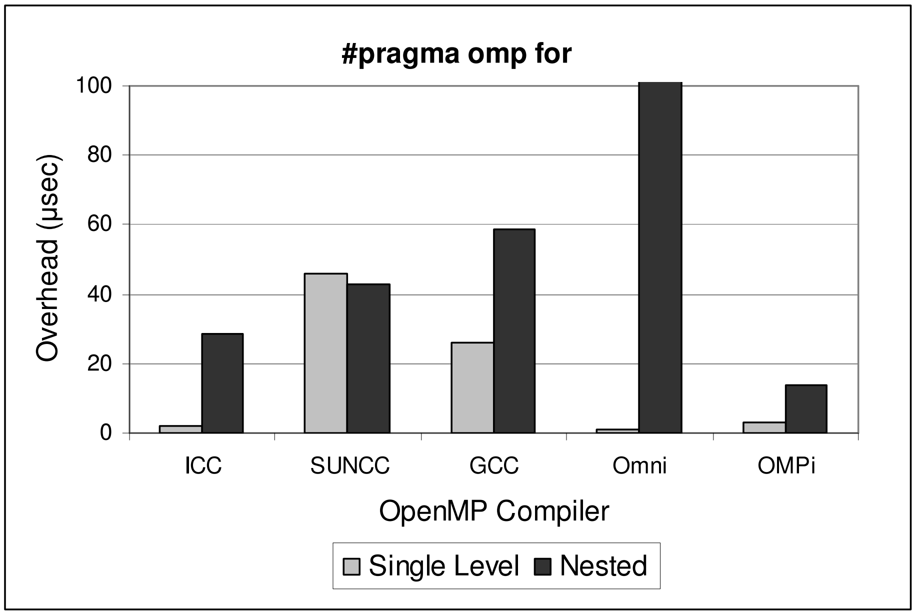

Figure 6. Runtime overhead of the OpenMP `for` construct (μsec).

decreases for SUNCC, GCC, and OMPi. The scheduling overhead depends strongly on the mechanism that OpenMP threads use to get the next chunk of loop iterations. Appropriate use of atomic primitives and processor yielding can significantly reduce thread contention

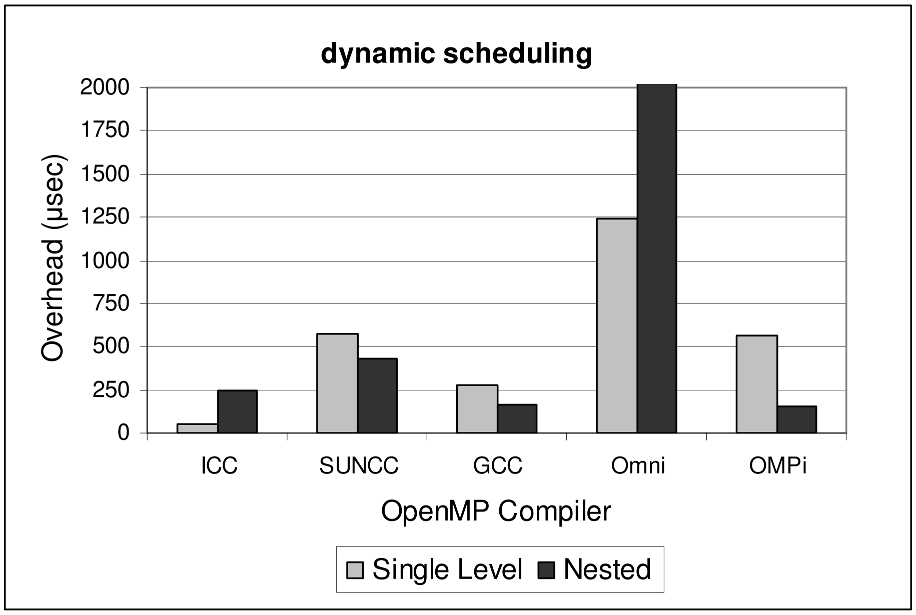

Figure 7. Runtime behavior of the dynamic scheduling policy (μsec).

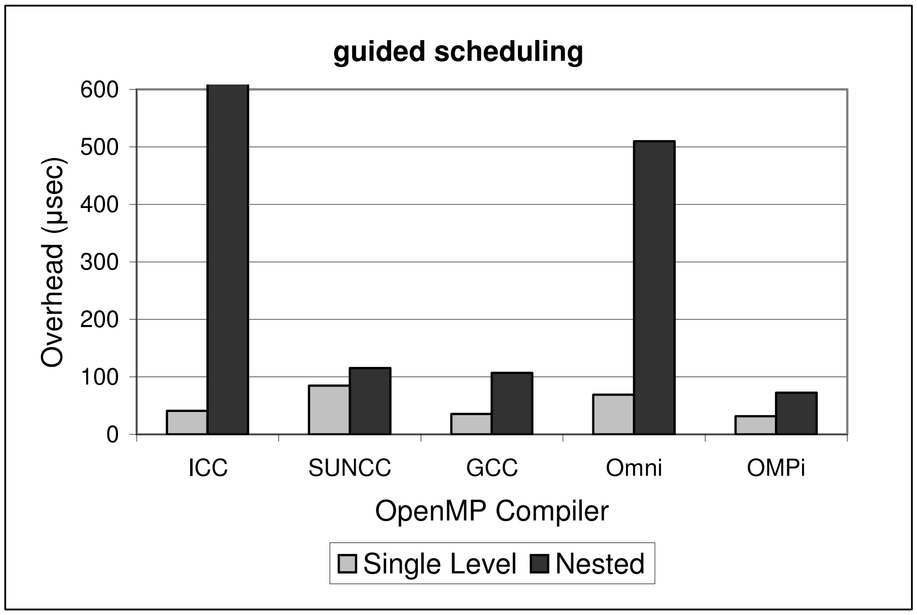

Figure 8. Runtime behavior of the guided scheduling policy (μsec).

during the dynamic assignment of loop iterations. This appears to be the case for the Sun Studio and GCC compilers, for which the dynamic scheduling overhead decreases when nested parallelism is exploited. OMPi with user-level threading achieves the same goal because it is able to assign each independent loop to a team of non-preemptive user-level OpenMP threads that mainly run on the same virtual processor. The overhead of the guided scheduling policy, as depicted in Fig. 8, is lower than that of the dynamic policy and increases for all the OpenMP implementations when nested parallelism is exploited.

Parallel data clustering The last experiment evaluates the runtime support of nested parallelism running PCURE on a data set that contains 5000 points with 24 features. The guided scheduling policy is used for both loops of the update procedure. We provide performance results after 1000 and 4000 clustering steps, depicted in Fig. 9 and 10 respectively. With the exception of OMPi, all OpenMP compilers utilize kernel threads. If `OMP_NUM_THREADS` equals T, these compilers use T^2 total kernel threads. On the other hand, OMPi initializes its user-level threads library with T virtual processors and creates T^2 user-level threads. The Intel and Omni compilers perform best when T=2 because the actual number of kernel threads from both levels of parallelism is equal to the system's processor count (4). For T=4, where 16 threads are created in total, the speedup drops mostly due to the higher threading overhead and the increased memory traffic and contention. For both SUNCC and GCC, the speedup is improved slightly on 4 threads, although this improvement declines as the number of steps increases. On the other hand, PCURE scales well in both experiments for the OMPi compiler. 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.

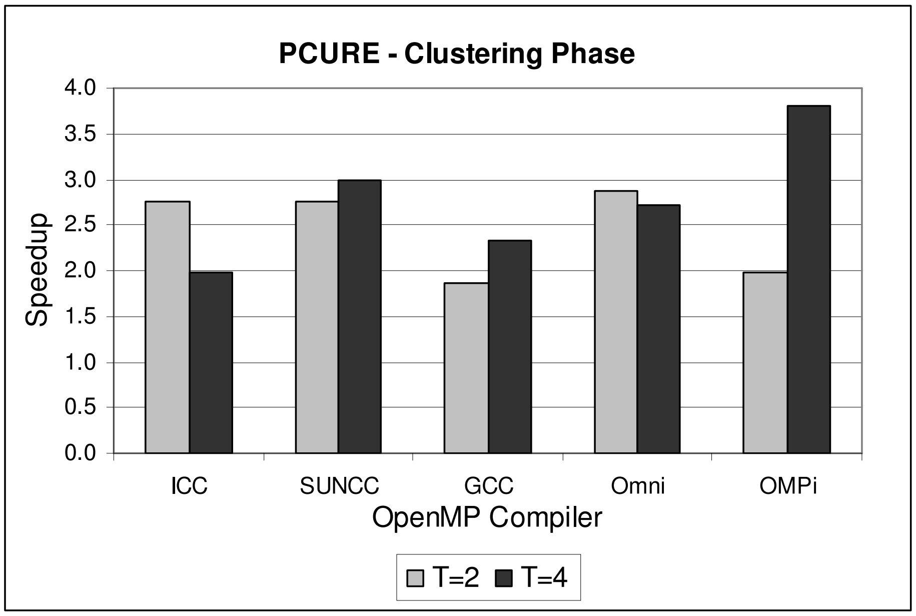

Figure 9. Speedups for 1000 steps of the parallel data clustering algorithm.

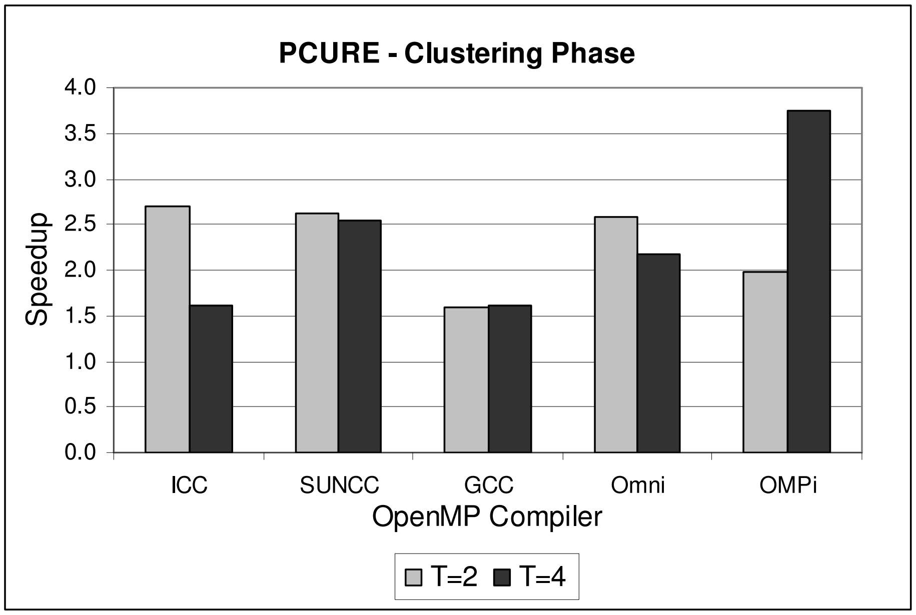

Figure 10. Speedups for 4000 steps of the parallel data clustering algorithm.

5. Conclusion

In this chapter we performed an in-depth analysis of nested parallelism in OpenMP. We examined both what the OpenMP specifications provide to the programmer for controlling nested parallelism and what the known implementations support. We surveyed commercial as well as research / experimental systems and categorized them according to the level of support they offer. After a decade since the first version of the OpenMP specifications came out, nested parallelism support is still at its infancy. There exist compilation systems that provide no support at all, but fortunately we found many systems that provide either partial or full support (i.e. the implementation does not limit dynamically the number of threads created).

However, we discovered that most implementations have scalability problems when nested parallelism is enabled and the number of threads increases beyond the number of available processors. Through specially designed microbenchmarks it was shown that the overheads increase dramatically even when moving merely to the second nesting level, in all but the OMPi compiler. In order to see the effect on the overall performance, a hierarchical data clustering application was also employed. The overheads manifested themselves as significantly lower-than-optimal attainable speedups, which got worse as the number of threads increased. It is clear that there are several design issues and performance limitations related to nested parallelism support that implementations have to address in an efficient way. The most important seems to be the kernel-level threading model which all but the OMPi compiler have adopted.

Last but not least, OpenMP still has a long way to go with respect to nested parallelism.

A number of issues in the specifications have to be clarified and a richer functional API for the application programmers must be provided.

References

[1] D. an Mey, S. Sarholz, and C. Terboven. Nested Parallelization with OpenMP. *International Journal of Parallel Programming*, **35**(5):459–476, October 2007.

[2] E. Ayguade, M. Gonzalez, X. Martorell, and G. Jost. Employing Nested OpenMP for the Parallelization of Multi-zone Computational Fluid Dynamics Applications. In *Proc. of the 18th Int'l Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, USA, April 2004.

[3] 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.

[4] J. Balart, A. Duran, M. Gonzalez, X. Martorell, E. Ayguade, and J. Labarta. Nanos Mercurium: A Research Compiler for OpenMP. In *Proc. of the 6th European Workshop on OpenMP (EWOMP '04)*, Stockholm, Sweden, October 2004.

[5] R. Blikberg and T. Sorevik. Nested Parallelism: Allocation of Processors to Tasks and OpenMP Implementation. In *Proc. of the 28th Int'l Conference on Parallel Processing (ICCP '99)*, Fukushima, Japan, September 1999.

[6] 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.

[7] J. M. Bull. Towards OpenMP V3.0. In *Proc. of the Int'l Conference on Parallel Computing: Architectures, Algorithms and Applications (PARCO '07)*, Aachen, Germany, September 2007.

[8] Jyh-Herng Chow, L. E. Lyon, and Vivek Sarkar. Automatic parallelization for symmetric shared-memory multiprocessors. In *Proc. of the 1996 conference of the Centre for Advanced Studies on Collaborative Research (CASCON'96)*, Toronto, Canada, November 1996.

[9] H. Chun and Y. Xuejun. CCRG OpenMP Compiler: Experiments and Improvements. In *Proc. of the 1st Int'l Workshop on OpenMP*, Eugene, Oregon, USA, June 2005.

[10] 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.

[11] K. S. Gatlin and P. Isensee. OpenMP and C++: Reap the Benefits of Multithreading without All the Work. In *MSDN Magazine*, October 2005.

[12] S. Guha, R. Rastogi, and K. Shim. CURE: An Efficient Clustering Algorithm for Large DataBases. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, 1998.

[13] P. E. Hadjidoukas and L. Amsaleg. Portable Support and Exploitation of Nested Parallelism in OpenMP. In *Proc. the 6th European Workshop on OpenMP (EWOMP '04)*, Stockholm, Sweden, October 2004.

[14] P. E. Hadjidoukas and L. Amsaleg. Parallelization of a Hierarchical Data Clustering Algorithm Using OpenMP. In *Proc. the 2nd Int'l Workshop on OpenMP (IWOMP '06)*, Reims, France, June 2006.

[15] P.E. Hadjidoukas and V.V. Dimakopoulos. Nested Parallelism in the OMPi OpenMP C Compiler. In *Proc. of the European Conference on Parallel Computing (EUROPAR '07)*, Rennes, France, August 2007.

[16] Hewlett-Packard Development Company. *Parallel Programming Guide for HP-UX Systems, 8th Edition.* 2007.

[17] International Business Machines (IBM) Corporation. *IBM XL C/C++ Enterprise Edition for AIX, V9.0: Compiler Reference.* 2007.

[18] International Business Machines (IBM) Corporation. *IBM XL Fortran Enterprise Edition for AIX, V11.1: Compiler Reference.* 2007.

[19] 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 Int'l Workshop on OpenMP: Experiences and Implementations (WOMPEI '03)*, Tokyo, Japan, November 2003.

[20] 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.

[21] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An Optimizing, Portable OpenMP Compiler. In *Proc. the 12th Workshop on Compilers for Parallel Computers*, A Coruna, Spain, January 2006.

[22] K. Matsubara, E. Kwok, I. Rodriguez, and M. Paramasivam. Developing and Porting C and C++ Applications on AIX. In *SG24-5674-01, IBM Redbooks*, July 2003.

[23] D. Novillo. OpenMP and automatic parallelization in GCC. In *Proc. of the 2006 GCC Summit*, Ottawa, Canada, June 2006.

[24] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface, Version 1.0.* October 1997.

[25] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface, Version 1.0.* October 1998.

[26] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface, Version 2.0.* March 2002.

[27] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface, Version 2.5.* May 2005.

[28] QLogic. *PathScale Compiler Suite User Guide, V.3.0.* 2007.

[29] 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.

[30] Sun Microsystems. Sun Studio 12: OpenMP API User's Guide, 2007. P.N. 819-5270.

[31] 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.

[32] Team RUNTIME INRIA. Marcel: A POSIX-Compliant Thread Library for Hierarchical Multiprocessor Machines. Available at: `http://runtime.futurs.inria.fr/marcel`.

[33] The Portland Group, STMicroelectronics, Inc. *PGI User's Guide: Parallel Fortran, C and C++ for Scientists and Engineers, 13th printing.* 2007.

[34] S. Thibault, F. Broquedis, B. Goglin, R. Namyst, and P-A Wacrenier. An Efficient OpenMP Runtime System for Hierarchical Architectures. In *Proc. of the 3rd Int'l Workshop on OpenMP (IWOMP '07)*, Beijing, China, June 2007.

[35] X. Tian, M. Girkar, S. Shah, D. Armstrong, E. Su, and P. Petersen. Compiler and runtime support for running openmp programs on pentium- and itanium-architectures. In *Proc. of the 17th Int'l Parallel and Distributed Processing Symposium (IPDPS '03)*, Washington, DC, USA, 2003.

[36] 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.