Exploiting Fine-Grain Thread Parallelism on Multicore Architectures

P. E. Hadjidoukas *G. Ch. Philos*
V. V. Dimakopoulos
Department of Computer Science
University of Ioannina
Ioannina, Greece, GR-45110
{phadjido,gfilos,dimako}@cs.uoi.gr

Abstract

In this work we present a runtime threading system which provides an efficient substrate for fine-grain parallelism, suitable for deployment in multicore platforms. Its architecture encompasses a number of optimizations that make it particularly effective in managing a large number of threads and with low overheads. The runtime system has been integrated into an OpenMP implementation to allow for transparent usage under a high level programming paradigm. We evaluate our implementation on two multicore systems using synthetic microbenchmarks and a real-time face detection application.

1 Introduction

Multicore architectures (MCAs) have become ubiquitous, and systems based on MCAs are these days a commodity, offering a very accessible means of achieving increased application performance. While CPUs with 2-8 cores have been already available in the last few years, the day when there will exist tens or hundreds of cores on a single die does not seem to be very far away [15].

Programming efficiently such systems is a necessity that in many cases becomes a headache. The reason is that because MCAs are the mainstream architecture, mainstream programmers (with years of sequential programming experience) will be called to program them. These programmers will be faced with problems and challenges that up to now were exclusive to the high-performance computing (HPC) community. However, traditional non-HPC programmers are more oriented towards productivity (and easiness of development) than performance. As such, programming models and supporting system software for MCAs have to be relatively easy to use and at the same time able to produce significant performance figures.

MCAs can be viewed as full SMP systems on a chip. In fact, this is the view most current operating systems have, hence not differentiating between SMPs and MCAs. While the similarities are indeed many, there are some subtle differences that are crucial for application performance. To start with, the cores in MCAs have a deeper hierarchy of memory sharing than the CPUs in an SMP. While the latter share only main memory, MCA cores typically share both memory and L2 caches. This favors computational locality and results in high core-to-core communication speeds. On the other hand, L2 caches in MCAs are much smaller (they are expected to one or two orders of magnitude smaller) than the L2 caches collectively be present in a similarly sized SMP system. Sharing such small caches will easily lead to cache conflicts in MCAs, and may have a negative impact on application performance.

MCAs are becoming non-uniform memory access (NUMA) machines. There is a significant difference when an access hits on a shared L2 cache and when the access has to go all the way to main memory. The NUMA factor will be even bigger as tens or hundreds of cores will have to be interconnected by either an on-chip network or a very deep hierarchy of shared caches. SMPs on the other hand are mostly considered UMA machines and are treated as such by both the programmer and the runtime system.

MCAs will provide many cores; the Intel 80-core prototype is already almost 2 years old [15]. In SMPs, the CPUs were typically limited to single-digit numbers, reaching 16 or 32 in some high-end systems. While applications for a limited number of CPUs can be coarser, for MCAs they will have to be finer in order to utilize efficiently the available computational power.

In conclusion, while MCAs are similar to SMPs, there exist enough and important differences to justify reconsidering the design of all software levels, from system software up to the application level.

In this paper we consider a runtime threading system capable of leveraging MCAs, producing high performance execution while hidden under an easy-to-use programming model. In particular, we make the case that OpenMP [18] is a very appropriate programming model for today's and most probably tommorow's multicore systems. This stems from the fact that it provides high-level abstractions and incremental program development without altering the base programming language, making it thus accessible to traditional non-HPC programmers. At the same time we have found that it allows for advanced runtime systems to take full advantage of the underlying processing hardware.

The rest of the paper is organized as follows. Section 2 presents the design of our runtime system. In Section 3 we discuss the integration of our runtime system into an OpenMP compiler so as to make its usage transparent to the application programmer. Section 4 assesses the effectiveness of our design; we present experimental results which include both synthetic microbenchmarks and a full-fledged application. Section 5 discusses related work and, finally, Section 6 concludes the paper.

2 The threading runtime system

We consider runtime libraries that will be called to harness the multiplicity of processing cores through multiple threads. Threads is the natural choice for parallel execution. Even in the case of new application-domain languages, where the programmer is not directly exposed to threads, but uses a higher-level abstraction, the underlying runtime system is based on threads (e.g. Cilk [4]).

While threading systems always strived for performance, we have identified a number of characteristics that threading systems for MCA platforms will be called to provide and handle efficiently:

- *Load balancing* is a prerequisite for high-performance execution. Although, this will be a very serious issue when a larger number of cores is available, it is also important today. For example, nested parallelism may actually lead to increased execution times if the workload is not balanced appropriately.

- *A large number of threads.* This is a necessity, not only because multicore CPUs will consist of many cores but also because, even with a relatively small number of cores, a large number of low-overhead threads may help towards balancing the load. A few coarse threads on a few cores can easily lead to load imbalance and underutilization of the system. Large numbers of *fine-grain* threads will be required, along with an efficient management of their execution.

- *Effective scheduling.* Traditional SMP-style thread scheduling (e.g. based on processor affinity) will not suffice. NUMA factors will have to be taken into account, especially when a larger number of cores becomes available. Some form of hierarchical scheduling seems appropriate, so as to match the hierarchy in architecture and memory sharing.

We have designed PSTHREADS, a high-performance threads library architected to meet the above requirements in a multicore environment. It exports a POSIX threads-like interface and implements a two-level thread model, where non-preemptive user-level threads are executed on top of kernel-level threads that act as *virtual processors.*

2.1 Core design

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, while the primary user-level thread operations, i.e. creation and context-switching, are provided by UTHLIB (Underlying Threads Library), a platform-independent thread package. These operations are based on the management routines of the `jmpbuf` or `ucontext_t` structures, although they can also be emulated using exclusively POSIX threads. UTHLIB utilizes a queue-based recycling mechanism for the underlying threads. The routines required for multiprocessor synchronization

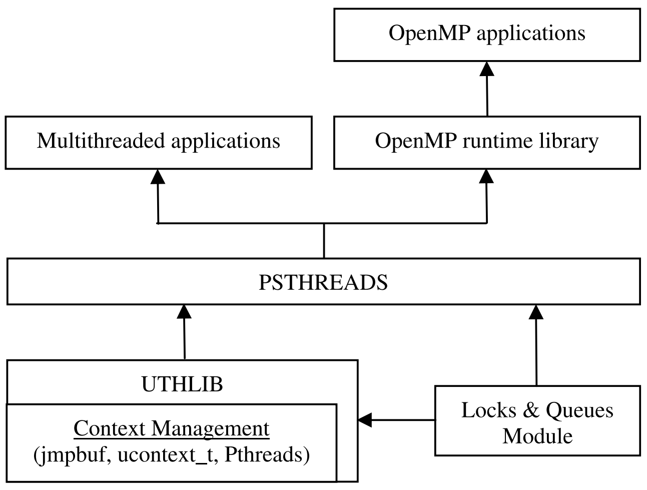


Figure 1: General design of the PSTHREADS library

and queue management are implemented in a separate module. Locks are internally mapped to POSIX mutexes or spinlocks, taking into account the non-preemptive threads of the library. The routines and the exported application programming interface of UTHLIB are both utilized by the PSTHREADS runtime library, which actually implements the two-level thread model (Figure 1).

Two important features of PSTHREADS are (a) the utilization of different data structures for the `psthread` descriptor and the underlying thread (i.e. stack) and (b) the adoption of a lazy stack allocation policy. Thus, the stack of a `psthread` is allocated right before the first context-switch to it, which means that a `psthread` binds an underlying thread at that time. This results in minimal memory consumption and actual thread migrations between processors. Specifically, a large number of threads can be spawned without having to allocate an equal number of stacks. Thus, better load balancing can be achieved, with low runtime overheads due to the user-level operations. Moreover, stack size can be fixed and large enough, which simplifies and accelerates stack management.

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. There are local (per virtual processor) queues and one optional global queue that can be used for coarse-grain tasks. A second set of queues is also available for the recycling of thread descriptors. The thread creation routine of PSTHREADS, which always tries to reuse a descriptor, does not imply insertion of the thread to a ready queue. Instead, an additional routine 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 at the back of the specified queue. Moreover, a parent thread does not have to join explicitly each child thread: a wait routine suspends the execution of the current thread until all its child threads have finished. Whenever a thread is blocked, the library scheduler is invoked and another thread with its own stack can be dispatched for execution on the same virtual processor.

Load balancing is achieved through work stealing [5], according to which an idle virtual processor first checks its local ready queue and then tries to steal work from the queues of the other virtual processors. 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.

2.2 Scheduling

When many threads are spawned or nested parallelism is exploited, kernel level thread models oversubscribe the system processors and time-sharing increases significantly runtime overheads. In addition, scheduling threads that map to inner levels of parallelism becomes difficult with respect to their binding to specific processors in order to favor computation and data locality. Limiting the number of created threads avoids the excessive runtime overheads but can easily cause load imbalance. The utilization of non-preemptive user-level threads allows the runtime library to manage parallelism explicitly, which is not possible for the case of kernel threads.

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 provides support to an adaptive work distribution scheme for the management of general unstructured 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. This scheme favors the execution of inner threads on a single processor and improves data locality.

The work stealing mechanism has been designed to work hierarchically, assuming the existence of thread groups. Specifically, the virtual processors are organized as a hierarchy of groups, which can be arbitrary. Thus, an idle virtual processor first examines the ready queues of its adjacent virtual processors in the lowest level group it belongs to; if no work is found, it tests the queues of the remaining processors in the group one level higher and so on. 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. In our runtime system, we do not introduce any additional ready queues. This simplifies the implementation and avoids the overhead of moving threads across queues of different levels. Having a single queue for each higher level group can result in hot spot contention, if all processors try to access that queue. Due to our hierarchical visiting order, however, the accesses to the ready queues is performed more evenly between virtual processors. An outline of our hierarchical work stealing mechanism, with the assumption that groups of the same level have equal size, is presented in Figure 2. The group size at each level can be arbitrary. In particular, they can be defined by the user or discovered during PSTHREADS initialization to match the hierarchical organization of the system cores. Figure 3 shows an example of the work stealing algorithm and the hierarchical visiting order of the queues for a specific virtual processor.

The Cilk runtime system [4] also maintains a local ready queue for each processor (so does the Intel Threading Building Blocks, TBB, library [20]). The ready queue is an array of lists, with each list corresponding to a specific level of parallelism. An idle processor selects randomly a ready queue and, if this is nonempty, extracts the thread from the tail of the list that corresponds to the outermost level of parallelism. Due to random stealing, however, these systems do not take into account any hierarchy among the processing cores. Other approaches (e.g. [17]) utilize an hierarchical queue scheme: the ready queues are organized as a tree, having a central queue at the root and local ready queues all the way down to the leaves. A processor has access only to the queues that reside on the path between its own local ready queue and the highest-level central queue. The drawback of this scheme is that it defines a fixed partitioning of the system processors and can lead to load imbalance.

```
GroupSize[0..NLevels-1]: # VPs in each group at a given level
/* private data */
MyID: current virtual processor
visited[0..N-1]: flags initialized to zero at every call

visited[MyID] = 1; /* local queue was empty */
thread = NULL;
Level = 0;
while (Level < NLevels) {
  nvisits = 0;
  VPGroupSize = GroupSize[Level];
  VPGroupID = MyID / VPGroupSize;
  VPGroupBase = VPGroupID*VPGroupSize;
  if (Level == 0)
    vp = MyID;
  else
    vp = (MyID + GroupSize[Level-1]) % VPGroupSize + VPGroupBase;
  while ((thread == NULL) && (nvisits < VPGroupSize)) {
    if (!visited[vp]) {
      visited[vp] = 1;
      thread = DequeueWork(vp);
    }
    vp = (vp + 1) % VPGroupSize + VPGroupBase;
    nvisits++;
  }
  if (thread != NULL) break;
  Level++;
}
/* execute thread */
```

Figure 2: Outline of the hierarchical thread scheduling algorithm

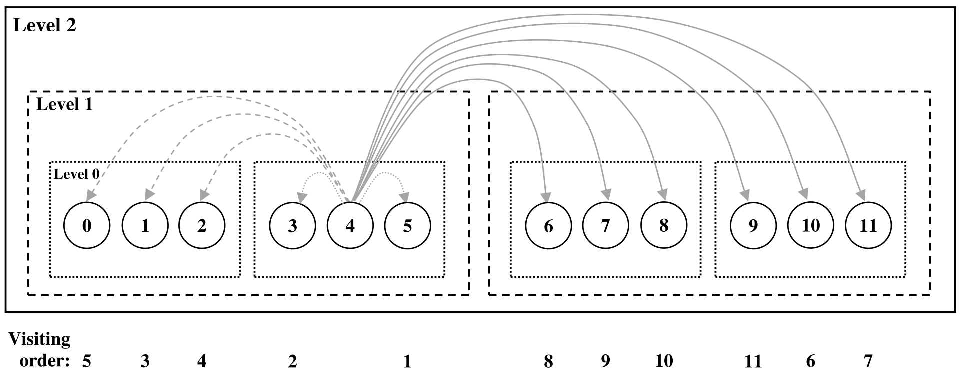


Figure 3: An example of the work stealing algorithm and the visiting order of the remote queues for virtual processor #4. The group sizes at levels 0, 1 and 2 are 3, 6 and 12 respectively.

2.3 Enhancements for fine-grain parallelism

The efficient support of a large number of threads has motivated further improvements in the library, related to memory recycling and thread barriers.

Both PSTHREADS and UTHLIB employ a queue-based recycling mechanism for the `psthread` descriptors and the underlying stacks respectively. As a central queue approach provides load balancing but suffers from high contention, recycling is performed through local queues. The thread creation routine always tries to recycle a thread from the local queue and finished threads are inserted locally too. To deliver the best performance, however, parallel programs that spawn many threads must rely on the local recycling without requiring access to the other queues. UTHLIB achieves this goal due to the lazy stack allocation policy: in the case of single-level parallelism, one underlying stack per virtual processor is adequate for any arbitrary number of threads. Moreover, each level of parallelism introduces at most one stack on each virtual processor. A minor improvement of this mechanism, which avoids the access to the local recycling queue (although it is contention-free), is the support for stack handoff. Specifically, a finished thread does not recycle its stack but instead (a) replaces its descriptor with the subsequent thread's descriptor, (b) re-initializes its own execution state and (c) resumes execution.

In contrast to underlying threads, local recycling does not imply optimal memory management of thread descriptors. Due to work stealing in PSTHREADS, threads can run and finish on any virtual processor. Therefore, local recycling queues will have available descriptors that can be consumed during thread creation. If, however, spawning of parallelism is mostly performed on the same virtual processor, there will be one producer and many consumers for the thread descriptors. To overcome this issue we implemented a direct-reuse mechanism for the descriptors. Specifically, we have introduced a variant of the thread creation routine (`psthread_create_ex`) that can accept a previously allocated descriptor. If the pointer to the descriptor is not set (i.e. has the value NULL) then the recycling mechanism is activated, otherwise the memory of the provided descriptor is used. When such a thread finishes, the library does not recycle the memory of the corresponding descriptor. It is the user's responsibility to save the pointer to the descriptor, in order to reuse it later in a subsequent thread creation call.

Figure 4 shows an example of thread creation using the direct memory reuse method. Initially, a number of threads (N) are created and submitted for execution in the queue of virtual processor 0. As the thread descriptors have not been set, the `pthread_create_ex` involves memory allocation for each descriptor. In the second loop, however, the creation routine takes as input the previously allocated descriptors and hence does not activate the

```
int i;
psthread_t t[N];
psthread_attr_t attr = PSTHREAD_ATTR_DEFAULT;

/* 1st round - thread descriptors are allocated */
for (i = 0; i < N; i++) {
  t[i] = NULL;
  psthread_create_ex(&t[i], &attr, func, arg);
  psthread_enqueue(t[i], 0);
}
psthread_waitall();    /* wait all child threads */

/* 2nd round - thread descriptors are provided */
for (i = 0; i < N; i++) {
  psthread_create_ex(&t[i], &attr, func, arg);
  psthread_enqueue(t[i], 0);
}
psthread_waitall();
```

Figure 4: An example of using psthreads with the variant of the thread creation routine. Thread descriptors are allocated only during the first loop.

recycling mechanism.

The PSTHREADS library also implements barriers between its user-level threads, exporting a set of routines similar to that of POSIX threads. The barrier initialization function takes as argument the number of the threads that will call the barrier, while the wait function suspends the calling thread until the specified number of threads reach the barrier. An efficient spinning barrier implementation cannot be used because we cannot make any assumption about the number of participating threads (i.e. there may be more than the number of processors). In a simple implementation, similar to that presented in [7], each `psthread` that reaches a barrier is inserted in the private queue of that barrier and releases the underlying virtual processor. The last thread extracts and reinserts each blocked thread in the ready queue of the virtual processor that was previously executing it. To avoid contention at the barrier queue, we have introduced an optimization to our barrier implementation. Specifically, the barrier initialization routine, does not use a single queue but allocates an array of pointers to thread descriptors. The size of the array is equal to the number of threads that will join the barrier. The barrier wait routine is also extended to include as second argument the id of the thread, which must be provided by the programmer or the software that utilizes PSTHREADS (e.g. OpenMP runtime library). Whenever a thread reaches a barrier, it registers itself in the array, at the position that is determined by its id. The last thread accesses the array and reinserts every descriptor in the appropriate ready queue.

3 A runtime system for OpenMP

OpenMP [18] has become a standard paradigm for shared memory programming, as it offers the advantage of simple and incremental parallel program development, in a high abstraction level. One of the reasons that OpenMP has been so successful is that it does not change the base language; application programmers continue to use C/C++ or Fortran, augmented with directives that take effect only if an OpenMP compiler is used. In addition, the programmer does not have to deal with threading details, as these are taken care of by an accompanying runtime library. As a result, OpenMP is a programming model quite accessible to non-HPC programmers.

OpenMP was designed with SMP architectures in mind and as such it fits the MCA model quite nicely, albeit it may not always deliver top performance as almost all implementations seem to have scalability problems when the number of threads increases significantly. For example, it has been demonstrated [11] that under fine-grain nested parallelism, where a large number of threads have to be managed, almost no implementation has a graceful behavior. However, this does not mean OpenMP is unsuitable for fine-grain parallelism. We show here that a high-performance runtime support library for OpenMP may allow applications to reach high performance levels. OpenMP is probably not ready for the many-core era as it does not currently allow for NUMA exploitation. However, there have been quite a few proposals in the open literature for NUMA extensions and it is certainly going to be one the key concepts for the next versions of OpenMP [3, 8].

We have integrated PSTHREADS into OMPi [19], a source-to-source compiler for OpenMP/C V2.5. OMPi takes as input C source code with OpenMP directives and outputs transformed but equivalent C code augmented with calls to OMPi's runtime system. The resulting program is compiled by the system's native C compiler and linked with the runtime library producing the final executable.

The runtime system of OMPi has been designed with modularity in mind and makes it particularly simple to modify its threading primitives or incorporate new ones. OMPi, through its compilation process maps OpenMP threads to abstract *execution entities* (EEs). The runtime system of OMPi provides the EEs that will carry out the work of OpenMP threads and controls their operation and synchronization. It has been architected with an internal interface that facilitates the integration of arbitrary EEs. It consists of two modules; the first module (ORT) groups EEs, coordinates them and schedules their execution within worksharing regions, but *it does not implement them.* The second module (EELIB) is the one that actually implements the execution entities. A number of EELIBs that provide thread EEs are available for OMPi, including libraries that are based on POSIX threads and Solaris threads. Finally, there is one more library that provides heavyweight processes as EEs and interfaces with arbitrary software DSM cores, providing transparent execution on clusters [19].

We have implemented a custom EELIB to interface with PSTHREADS. The EELIB that integrates PSTHREADS into OMPi is relatively straightforward because spawning of threads is performed explicitly, while thread pooling is provided by the thread library. OpenMP thread self-identification is based on thread local storage that PSTHREADS supports, and OpenMP barriers are directly mapped to PSTHREADS barriers. Upon application startup, the value of the `OMP_NUM_THREADS` environment variable determines the number of virtual processors. If this variable has not been set or its value exceeds the system's processor count, the number of virtual processors is set equal to the number of physical processors. Thus, OMPi maps the OpenMP threads to lightweight PSTHREADS and the number of kernel-level threads never exceeds the number of physical processors. This approach minimizes the OpenMP runtime overheads, especially when nested parallelism is enabled, and manages to exploit fine-grain parallelism. In addition, the internal thread scheduling scheme of the PSTHREADS library favors the execution of inner-level OpenMP threads on a single processor and improves data locality.

4 Experiments

4.1 Methodology

The integration of the PSTHREADS library into an OpenMP environment allows us to evaluate the efficiency of its threading primitives using standard microbenchmarks that have been proposed for OpenMP instead of designing our custom ones. Moreover, using various OpenMP implementations we have a straightforward comparison between different threading approaches.

For this purpose, we use the EPCC microbenchmark suite [6], which measures the overhead of the OpenMP constructs, including the costs for creating parallelism (threads), locking and barriers. However, this suite is only applicable to single-level parallelism; running the benchmarks with a large number of threads can give an overhead estimation that is not accurate. Evaluating nested parallelism based on application speedups [25, 1] gives overall performance indications but does not reveal potential construct-specific problems. To study how efficiently OpenMP implementations support nested parallelism and exploit fine-grain parallelism, we have extended both the synchronization and the scheduling microbenchmarks of the EPCC suite [11].

The technique followed in the EPCC microbechmark suite for measuring 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. According to our approach, the core benchmark routine for a given construct is represented by a *"task"* (not to be confused with the task directive introduced in OpenMP 3.0). Each "task" has a unique identifier and utilizes its own memory space for storing its table of runtime measurements. We create a team of threads, where each member of the team executes its own "task". When all "tasks" finish, we compute the global mean of all measured runtime overheads. The method is outlined in Fig. 5. The team of threads that execute the "tasks" expresses the outer level of parallelism, while each benchmark routine ("task") contains the inner level of parallelism.

In Fig. 5, if the loop (lines 4–6) is not parallelized, the "tasks" are executed in sequential order. This is equivalent to the original version of the microbenchmarks, having each core benchmark repeated more than once. On the other hand, if nested parallelism is enabled, the loop is parallelized (lines 1–3) 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 each member of the team executes exactly one "task", a static schedule with chunksize of 1 was chosen at line 2. In addition, to guarantee that the OpenMP runtime library does not assign fewer threads to inner levels than in the outer one, dynamic adjustment of threads is *disabled* through a call to `omp_set_dynamic(0)`.

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. In addition, due to the presence of more than one team parents, the overhead of the parallel directive increases in most implementations, possibly causing overestimation of other measured overheads. To resolve these issues, we increase the number of internal repetitions for each microbenchmark, so as to be able to reach the same confidence levels (95%).

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

  #ifdef NESTED_PARALLELISM
  #pragma omp parallel for schedule(static,1)
  #endif
  for (task_id = 0; task_id < p; task_id++) {
    (*originalfunc)(task_id);
  }

  <compute global statistics>
  <print construct name and statistics>
}


main() {
  <compute reference time>
  omp_set_num_threads(omp_get_num_procs());
  omp_set_dynamic(0);
  nested_benchmark("PARALLEL", testpr);
  nested_benchmark("FOR",      testfor);
  ...
}
```

Figure 5: Extended microbenchmarks for nested parallelism overhead measurements

4.2 Microbenchmark results

We conducted our experiments on a server with 4 dual-core Intel Xeon Paxville 3.0GHz CPUs running Linux 2.6. We provide results for two free commercial and two freeware OpenMP C compilers that 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 and OMPi 1.0.0. We have used the default settings of the OpenMP runtime libraries and the -O3 optimization flag in all experiments.

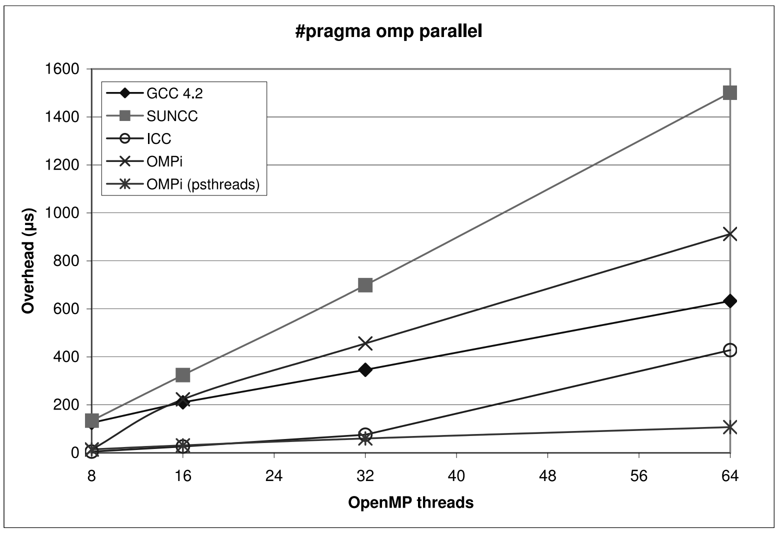
Our first experiment demonstrates our lightweight runtime support as the number of OpenMP threads increases. Figure 6 presents the overheads of the `parallel` (thread creation) and `barrier` OpenMP constructs, increasing the number of threads from 8 up to 64 on a dedicated machine and having a single-level of parallelism. We observe that GCC, SUNCC, and OMPi with POSIX threads (OMPi) exhibit significant overheads while both ICC and OMPi+PSTHREADS achieve good performance for up to 32 OpenMP threads. For more threads, however, the Intel compiler fails to maintain stability and only the overheads of the OMPi+PSTHREADS compiler increase linearly with the number of threads (i.e. proportionally to the number of user-level context switches). This is attributed to the lower contention of user-level threads on the processing cores.

In the second experiment we focus on the evaluation of OpenMP runtime support for nested parallelism, using the extended EPCC microbenchmarks. A selection of the obtained results is given in Figures 7–8, for the synchronization and scheduling microbenchmarks. Each plot includes the single-level overheads of each system for reference. We have chosen a logarithmic scale for the y-axis for clarity. Both the `OMP_NUM_THREADS` environment variable and the number of "tasks" are equal to the number of processing cores in the system (8).

Fig. 7 includes the overheads for the `parallel` and `barrier` constructs. 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 GCC, Intel and OMPi compilers. For both of them, the runtime overhead is more than an order of magnitude higher in the case of nested parallelism. For ICC this could be attributed, in part, to the fact that threads join a unique central pool before getting grouped to teams [27]. On the other hand, both OMPi+PSTHREADS and SUNCC clearly scale better and their overheads increase linearly, with SUNCC, however, exhibiting higher overheads than OMPi for both single level and nested parallelism.

While ICC exhibits similar behavior for the `barrier` construct, both GCC and OMPi show significant but not excessive increase. The Sun compiler seems to handle inter-team barriers quite well showing a decrease in the actual overheads. OMPi+PSTHREADS manages to deliver the best performance, showing the advantage of user-level threading: inner levels are executed by lightweight threads, which mostly live in the processor where the parent thread is, eliminating most intra-team contention and the associated overheads.

Fig. 8 includes representative results from the scheduling microbenchmarks and specifically for static and guided schedules with a chunksize of 1. As before, we observe that the overhead of both scheduling policies increases substantially for the Intel compiler and considerably for GCC and OMPi. In contrast, the overheads of the guided scheduling policy actually decrease for both SUNCC and OMPi+PSTHREADS. For the Sun Studio compiler, this is



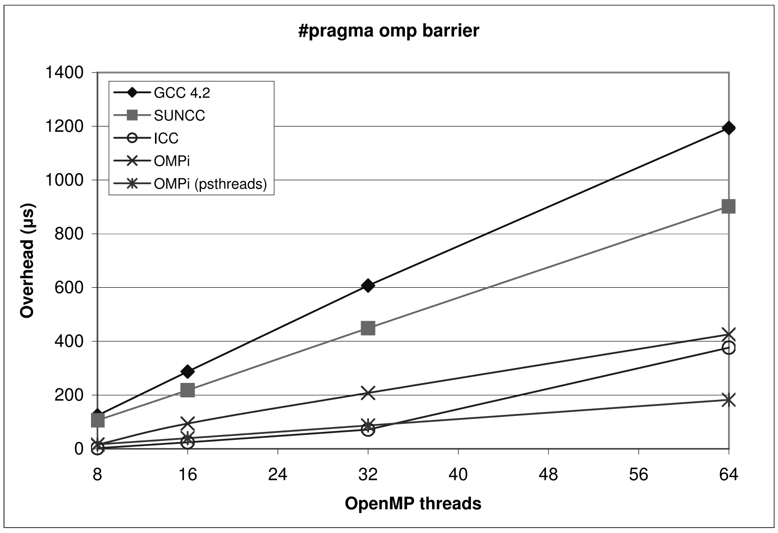
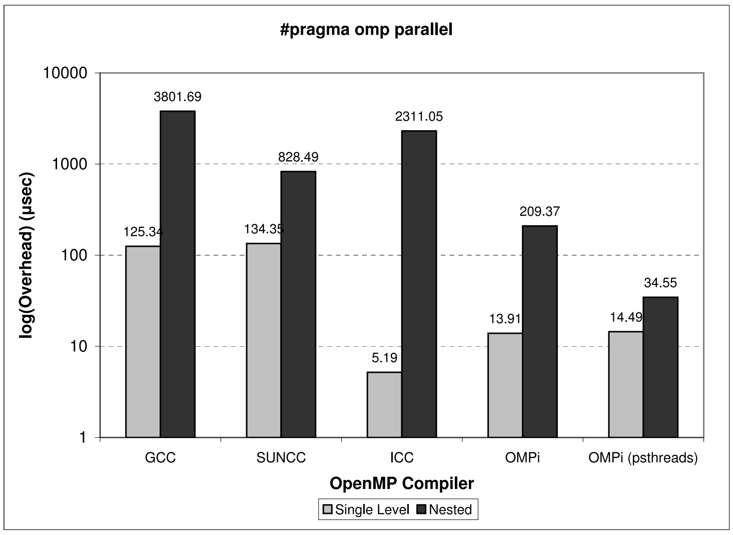


Figure 6: Single-level runtime overheads for large number of threads



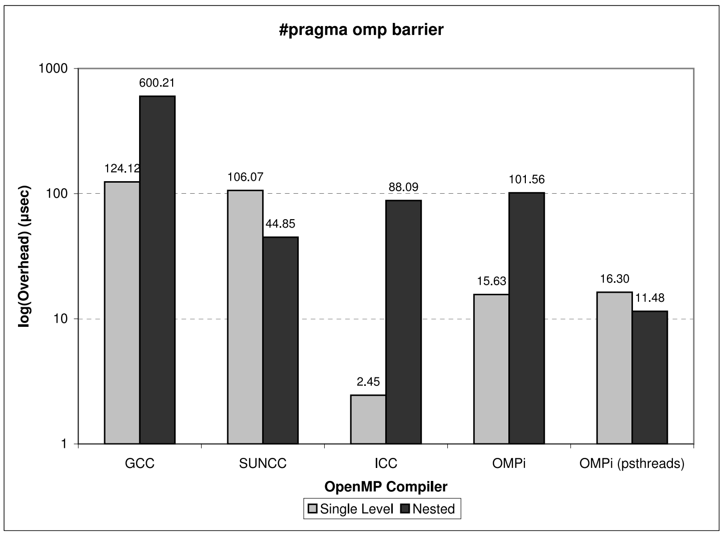


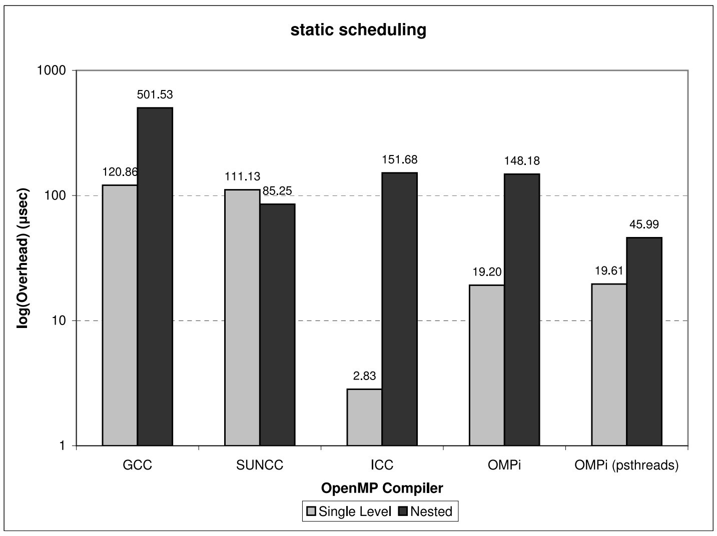
Figure 7: Overheads for `parallel` and `barrier` when one or two levels of parallelism are exploited

attributed to the appropriate use of atomic primitives and processor yielding, which can significantly reduce thread contention during the dynamic assignment of loop iterations. 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.

4.3 A fine-grain application

In this section, we evaluate our runtime system using a fine-grain parallel face detection application. The system is based on a special highly-structured neural network topology and provides the best so far, in terms of accuracy, face detection. The face detection system utilizes a fast pipeline method that is highly parallelizable due to its simplicity. Details can be found in [14].

As shown in [14], when processing a single image the performance of the face detection system does not scale linearly as the processor count increases. This is attributed to the inherent load imbalance, as the algorithm includes a small number of parallel iterations and the computational load cannot be distributed evenly to the system processors/cores. The exploitation of nested parallelism can provide an effective solution to the above and thus ameliorate the performance of the face detection system because additional fine-grain parallelism, in terms of inner parallel loops, is extracted from the application. The inner loops belong to the second level of parallelism, which is executed by new teams of OpenMP threads. Lightweight runtime support is crucial for better performance of the face detection system, considering the requirement for real-time processing of a single image.



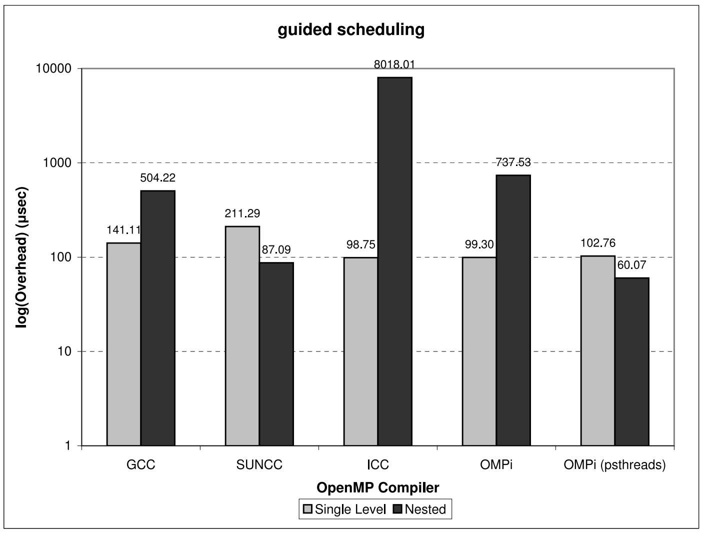


Figure 8: Overheads of static and guided loop scheduling when one or two levels of parallelism are exploited

Table 1: Nested parallelization overheads on the quad-core system (μs)

OpenMP compiler	**parallel**	**for**
GCC 4.2	553.24	21.39
SUNCC	38.63	14.86
ICC	60.81	8.35
OMPi (psthreads)	3.51	3.65

All our experiments we conducted on a server equipped with an Intel Xeon Quad-core X5355 processor (2.66GHz, 4MB L2 cache) and 2GB of main memory. The operating system was Debian Linux 2.6.

Table 1 presents a sample of the runtime overheads for the `parallel` and `for` OpenMP constructs under nested parallelism, measured using the microbenchmarks described in Section 4.1. In this experiment, both the number of "tasks" and OpenMP threads are equal to 4, resulting in 16 OpenMP threads that compete for computational resources. Comparing with the results on the eight cores, the performance of ICC is significantly improved on the quad-core system, mainly because this experiment generates less contention per processor. We also observe that OMPi continues to exhibit the lowest runtime overheads of all OpenMP implementations.

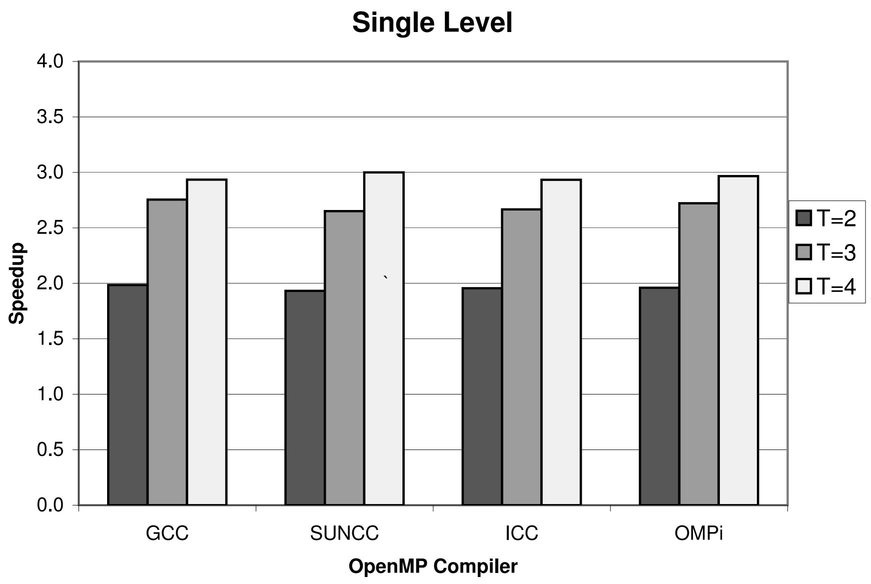
Figure 9 depicts the speedups of the parallel face detection system for a standard single-face test image (355×237 pixels wide), exploiting either a single or two levels of parallelism. For the case of single level parallelism, we observe that all configurations have similar performance and manage to improve the face detection responsiveness. OpenMP parallelization is the key factor that provides real-time performance (i.e. ≥25 images/sec) to the face detection system. We observe that the scalability of the face detection system is higher when nested parallelism is exploited using the

OMPi compiler and 4 OpenMP threads. In this case, OMPi+PSTHREADS attains the maximum speedup (3.66x), which is significantly improved, by up to 23%, compared to the corresponding 2.97x speedup for the single-level parallelism case, increasing thus the image processing rate.

When two OpenMP threads are used (T=2) for each parallel region, the three OpenMP compilers exhibit speedups that are better than that of OMPi and higher than the corresponding number of threads. This is reasonable because these configurations utilize 4 (2×2) total kernel threads, which run on all system processing cores. On the other hand, OMPi utilizes only two kernel threads. Using 3 OpenMP threads (T=3), we observe that all the compilers exhibit similar performance, with slightly higher speedup for the Intel compiler. PSTHREADS, however, uses one less processor than the others, which means that the overheads of the other OpenMP systems are significantly higher. As the number of OpenMP threads increases, more fine grain parallelism is exploited. Thus, the contention of the OpenMP kernel-level threads, which become more than the processing elements, is also increased.

5 Related work

Hybrid thread models provide a combination of parallelism and low overhead, having the advantages of both user-level and kernel-level models. With the exception of few Unix vendors (HP-UX and IBM/AIX), most POSIX threads libraries nowadays follow the kernel-level model, in which user threads are associated one-to-one to kernel entities. NGPT (Next Generation POSIX Threads) was a hybrid model implementation of POSIX threads for Linux, which has abandoned in favor of NPTL (Native POSIX Threads



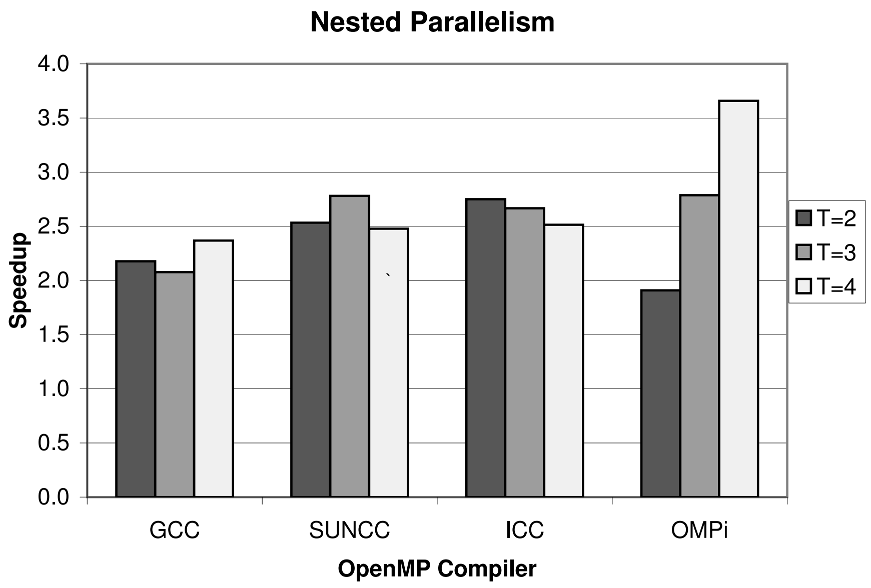


Figure 9: Speedups of face detection, exploiting a single level and two levels of parallelism. T is the value of the OMP_NUM_THREADS environment variable, and represents the number of threads at each nesting level.

Library) [12]. If POSIX thread libraries followed a hybrid implementation, however, the runtime overheads would be reduced, allowing the creation of several threads without additional performance cost, as shown in [21].

Only few runtime systems implement an efficient and portable two-level threads model for multiprocessor and multicore systems. Some of them have been used to provide runtime support to an OpenMP compiler. StackThreads/MP is a fine-grain thread library that provided efficient but not portable support for dynamic nested parallelism as runtime module of an experimental version of the Omni OpenMP compiler [25]. Marcel [26] is a portable thread library that features a two-level thread scheduler, provides a POSIX-compliant interface and runtime support to a modified version of the GNU OpenMP compiler. We have experimented with integrating Marcel into OMPi [13]. We observed slightly higher overheads than PSTHREADS, mainly due to its more complex implementation and preemptive thread scheduling.

McRT (Many-Core RunTime) [22] is a runtime environment for terascale chip multiprocessors that supports fine-grain parallelism, concurrency abstractions for easier parallel programming and supports platform and application heterogeneity. McRT includes an OpenMP adaptor that translates the API used by the Intel C compiler to the core McRT API.

NthLib is an efficient user-level threads library that provides runtime support to the NANOS OpenMP compiler [2]. Nested parallelization is based on the concept of thread groups, which are determined through appropriate OpenMP extensions. A group of threads is composed of a subset of the total number of available threads, while the rest threads support the execution of nested parallel constructs. Therefore, the total number of threads never exceeds that of available processors and, hence, the runtime overheads of nested parallelism are equal to those of single level. This approach, however, can lead to load imbalance because it does not fully exploit fine-grain parallelism and some processors may remain idle.

Tiny threads (TNT) [10] is a thread model for the Cyclops64 architecture and has been proposed as the first component of a Thread Virtual Machine targeted to cellular architectures. TNT is part of a microkernel for the C64 that runs directly on top of the C64 architecture aimed to high efficiency at the expense of portability. A dispatched thread will run until completion, without releasing the underlying hardware thread even if it is sleeping. The integration of TNT into the Omni runtime library [23] provides an OpenMP infrastructure for the C64, without considering multiple levels of parallelism though [9].

Besides OpenMP, thread libraries have provided runtime support to other parallel programming approaches too. Intel Threading Building Blocks [20] is a C++ library for multi core processors that does not require a special runtime or compiler. The library maps user tasks into threads which can be run in parallel and relieves the programmer from the overhead of manually optimized thread design when conventional threads are used. Factory [24] is a similar object-oriented parallel programming substrate which allows programmers to express multigrain parallelism without having to manage it.

Cilk [4] is a parallel programming language that does not use explicit threading but Cilk frames, which are generated by its cilk2c compiler. The Cilk runtime system maintains a local ready queue for each processor and deploys an efficient work-stealing scheduler. The EARTH programming model [16] follows a two-level hierarchy formed by threaded functions and fibers. Fibers are lightweight threads that are scheduled using a dataflow approach and executed in a non-preemptive manner. Hence, a fiber is never interrupted and must never block.

The above runtime libraries do not support global barriers between their work units. Whenever a work unit is blocked, the execution vehicle invokes the runtime scheduler and runs the selected work unit on the same stack. Therefore, these runtime libraries can not be used in an OpenMP implementation. On the contrary, PSTHREADS is a runtime framework that provides a lightweight implementation of nested OpenMP parallelism.

6 Conclusion

We presented the runtime architecture of PSTHREADS, a high-performance user-level threads library for efficient exploitation of fine-grain parallelism on multicore architectures. The library has been integrated into the runtime library of the OMPi OpenMP compiler, resulting in lightweight nested parallelism support. We have developed a methodology for measuring OpenMP construct overheads under nested parallelism. Using the developed microbenchmarks we evaluate the runtime overheads and demonstrate the advantages of user-level multithreading compared to the traditional kernel thread based approaches.

PSTHREADS allow for effective exploitation of fine grain parallelism with large numbers of threads, through hierarchical scheduling and work stealing techniques. Our current research targets support for heterogeneous cores and NUMA-aware thread scheduling.

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, J. Labarta, N. Navarro, and J. Oliver. NanosCompiler: Supporting Flexible Multilevel Parallelism in OpenMP. *Concurrency: Practice and Experience*, 12(12):1205–1218, October 2000.

[3] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner. Extending OpenMP for NUMA Machines. In *Proc. of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[4] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proc. of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, 1994.

[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] David R. Butenhof. *Programming with POSIX Threads*. Addison Wesley, 1997.

[8] B. Chapman, F. Bregier, A. Patil, and A. Prabhakar. Achieving Performance Under OpenMP on ccNUMA and Software Distributed Shared Memory Systems. *Concurrency and Computation: Practice and Experience*, 14(8–9):713–739, 2002.

[9] J. Del Cuvillo, W. Zhu, and G. Gao. Landing OpenMP on Cyclops-64: An Efficient Mapping of OpenMP to a Many-Core System-on-a-Chip. In *Proc. of the 3rd Conference on Computing Frontiers*, pages 41–50, 2006.

[10] J. del Cuvillo, W. Zhu, Z. Hu, and G.R. Gao. TiNy Threads: a Thread Virtual Machine for the Cyclops64 Cellular Architecture. In *Proc. of the 5th Workshop on Massively Parallel Processing (WMPP '05)*, Denver, Colorado, April 2005.

[11] V. V. Dimakopoulos, P. E. Hadjidoukas, and G. Ch. Philos. A Microbenchmark Study of OpenMP Overheads Under Nested Parallelism. In *Proc. of the Int'l Workshop on OpenMP (IWOMP '08)*, West Lafayette, USA, May 2008.

[12] U. Drepper and I. Molnar. The Native POSIX Thread Library for Linux. In *Technical Report*, Red Hat, Inc., January 2003.

[13] 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.

[14] P. E. Hadjidoukas, V. V. Dimakopoulos, M. Delakis, and C. Garcia. A High-Performance Face Detection System. *Concurrency and Computation: Practice and Experience, to appear.*

[15] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the intel 80-core network-on-a-chip terascale processor. In *Proc. of the 2008 ACM/IEEE conference on Supercomputing (SC '08)*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[16] C. J. Morrone, G. Tremblay J. N. Amaral, and G. R. Gao. A Multi-Threaded Runtime System for a Multi-Processor/Multi-Node Cluster. In *Proc. of the 15th Annual Int'l Symposium on High Performance Computing Systems and Applications*, June 2001.

[17] 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.

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

[19] G. C. Philos, V. V. Dimakopoulos, and P. E. Hadjidoukas. A Runtime Architecture for Ubiquitous Support of OpenMP. In *Proc. of the 7th Int'l Symposium*

on Parallel and Distributed Computing (ISPDC '08), Krakow, Poland, June 2008.

[20] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O' Reilly Media, Inc., 2007.

[21] R. Rufai, M. Bozyigit, J. Alghamdi, and M. Ahmed. Multithreaded Parallelism with OpenMP. *Parallel Processing Letters*, 15(4):367–378, 2005.

[22] B. Saha, A. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R.L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling Scalability and Performance in a Large Scale CMP Environment. *SIGOPS Oper. Syst. Rev.*, 41(3):73–86, 2007.

[23] 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.

[24] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Factory: An Object-Oriented Parallel Programming Substrate for Deep Multiprocessors. In *Proc. of the 1st Int'l Conference on High Performance Computing and Communcations (HPCC 2005)*, pages 223–232, September 2005.

[25] 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.

[26] S. Thibault. A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines. In *Proc. of the 2nd Int'l Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, Cambridge, USA, June 2005.

[27] 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.