Task-based Execution of Nested OpenMP Loops*

Spiros N. Agathos† Panagiotis E. Hadjidoukas
Vassilios V. Dimakopoulos
Department of Computer Science, University of Ioannina
P.O. Box 1186, Ioannina, Greece, GR-45110
{sagathos,phadjido,dimako}cs.uoi.gr

Abstract

In this work we propose a novel technique to reduce the overheads related to nested parallel loops in OpenMP programs. In particular we show that in many cases it is possible to replace the code of a nested parallel-for loop with equivalent code that creates tasks instead of threads, thereby limiting parallelism levels while allowing more opportunities for runtime load balancing. In addition we present the details of an implementation of this technique that is able to perform the whole procedure completely transparently. We have experimented extensively to determine the effectiveness of our methods. The results show the actual performance gains we obtain (up to 25% in a particular application) as compared to other OpenMP implementations that are forced to suffer nested parallelism overheads.

1 Introduction

OpenMP has become one of the most popular models for programming shared-memory platforms and this is not without good reasons; just to name a few, the base language (C/C++/Fortran) does not change, high-level abstractions are provided, most low-level threading details need not be dealt with and all these lead to ease of use and higher productivity. At the same time significant performance benefits are possible. While the initial target of OpenMP was mostly loop-level parallelism, its expressiveness expanded significantly with the addition of tasks in V3.0 of the specifications [8], making it now suitable for a quite large class of parallel applications.

Among the important features included from the very beginnings of OpenMP was nested parallelism, that is the ability of any running thread to create its

*This work has been supported in part by the General Secretariat for Research and Technology and the European Commission (ERDF) through the Artemisia SMECY project (grant 100230)

†S.N. Agathos is supported by the Greek State Scholarships Foundation (IKY)

own team of child threads. Although actual support for nested parallelism was slow to appear in implementations, nowadays most of them support it in some way. However, it is well known that nested parallelism, while desirable, is quite difficult to handle efficiently in practice, as it easily leads to processor oversubscription, which may cause significant performance degradation.

The addition of the `collapse` clause in V3.0 of the specifications can be seen as a way to avoid the overheads of spawning nested parallelism for certain nested loops. However, it is not always possible to use the `collapse` clause since:

- the loops may not be perfectly nested
- the bounds of an inner loop may be dependent on the index of the outer loop
- the inner loop may be within the extend of a general parallel region, not a parallel-loop region.

The nature of OpenMP loops is relatively simple; they are basically DO-ALL structures with independent iterations, similar to what is available in other programming systems and languages (e.g., the FORALL construct in Fortran 95, the `parallel_for` template in Intel TBB [10], or `cilk_for` in Cilk++ [7]). What is interesting is that some of these systems implement DO-ALL loops without spawning threads; they are mostly creating some kind of task set to perform the job. Can an OpenMP implementation do the same? While this seems rather useless for first-level parallel-for loops (since there is no team of threads to execute the tasks; only the initial thread is active), it may be worthwhile in a nested level.

What we propose here is a novel way of avoiding nested parallel loop overheads through the use of tasks. In particular, as our first contribution, we show that it is possible to replace a second-level loop by code that creates tasks which perform equivalent computations; the tasks are executed by the first-level team of threads, completely avoiding the overheads of creating second-level teams of threads and oversubscribing the system. We use the proposed method to show experimentally the performance improvement potential.

At the same time we observe that our techniques require sizable code changes to be performed by the application programmer, while they are not always applicable for arbitrary loop bodies. Our second contribution is then to automate the whole procedure and provide transparent tasking from the loop nests, which, except the obvious usability advantages, does not have the limitations of the manual approach. We present the implementation details of our proposal in the context of the OMPi [4] compiler. Finally, we perform a performance study using synthetic benchmarks as well as a face-detection application that utilizes nested parallel loops; all experimental results depict the performance gains attainable by our techniques.

The rest of the paper is organized as follows: in Section 2 we present the necessary code transformations that a programmer must perform in order to

```
#pragma omp parallel num_threads(M)
{
  #pragma omp parallel for\
    schedule(static) num_threads(N)
  for (i=LB; i<UB; i++) {
    <body>
  }
}
```

Figure 1: Nested parallel loop example

```
#pragma omp parallel num_threads(M)
{
  for (t=0; t<N; t++)
    #pragma omp task
    {
      calculate(N,LB,UB,&lb,&ub);
      for (i=lb; i<ub; i++)
          <body>
    }
  #pragma omp taskwait
}
```

Figure 2: Transformation outline of Fig.1.

produce tasking code equivalent to a nested loop region, for cases where this is indeed possible. In Section 3 we discuss the transparent implementation of the proposed methodology, which is applicable for general loops and schedules. Section 4 contains our performance experiments and finally Section 5 concludes this paper.

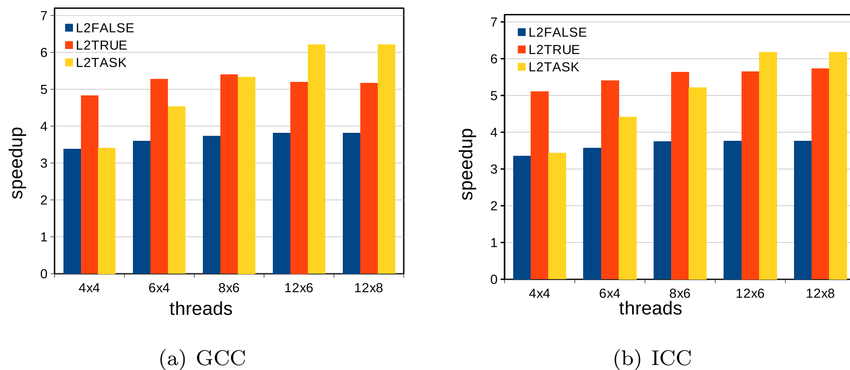
2 Proof of Concept: Re-Writing Loop Code Manually

Consider the sample OpenMP code shown in Fig. 1. There exists a nested parallel for-loop which will normally spawn a team of N threads for each of the M (first-level) threads that participate in the outer `parallel` region[1]; a total of $M \times N$ threads may be simultaneously active executing second-level parallelism, leading to potentially excessive system oversubscription.

Fig. 2 illustrates how the code of Fig. 1 can be re-written so as to make use of OpenMP tasks instead of nested parallelism. The idea is conceptually simple: each first-level thread creates N tasks (i.e. equal in number to the original code's second-level threads), and then waits until all tasks are executed. The code of each task contains the original loop body; in order to perform the same work the corresponding thread would perform, it is necessary to calculate the exact iterations that should be executed, hence the `calculate()` call. In essence, the user code must perform the same calculations that the runtime system would perform for the case of Fig. 1 when handing out the iterations for the static schedule.

Why does it work? The answer is that the original code creates *implicit* tasks, according to OpenMP terminology, while the code in Fig. 2 emulates them through the use of *explicit* tasking. Also, while implicit tasks may contain barriers (which are not allowed within explicit tasks), there is no such a

[1] Even if `num_threads(M)` is absent from the outer parallel construct, the standard practice is to produce as many threads as the number of available processors. Assume, without loss of generality, that they are equal to M.



(a) GCC (b) ICC

Figure 3: Performance of the proposed technique; speedup for a face detection algorithm applied on an test image with 57 faces.

possibility here since the implicit tasks in Fig. 1 only execute independent loop iterations, and within the loop body there can not exist a barrier closely nested. As a result, the programs in Figs. 1 and 2 are equivalent, and no other changes are required[2], (we will return to this in the next section).

The important difference is that the code in Fig. 2 does *not* generate a second level of parallelism. It utilizes the tasking subsystem of the compiler and uses only the available M threads to execute the $M \times N$ tasks generated in total, allowing for improved load balance opportunities. While task creation and execution is not without overheads, it remains mostly in the realm of the OpenMP implementation to deal with it efficiently. On the other hand controlling a large amount of threads resulting from nested parallel regions may not be possible, especially when the OpenMP runtime relies on kernel-level threads (such as POSIX threads, which is a usual case).

We applied the above technique to the parallel version of a high-performance face detection application. The application itself, the 16-core system as well as the software configuration we used for our experiments are described in more detail in Section 4 so we will avoid repeating it here. The important issue is that the application contains a first-level parallel loop with unbalanced iteration load. The number of iterations depends on the image size and is usually less than 14. Inside this loop, there exist multiple second-level parallel for-loops, which clearly need to be exploited in order to increase performance. We re-wrote these second-level loops according to the method described above. We compiled the code with various OpenMP compilers and in Fig. 3 we show the execution results obtained using GNU GCC and Intel ICC on a particular image containing 57 faces; similar results were observed with other compilers, too. In the figure, the

[2] Actually one more change may be need. Because in task regions the data sharing attributes of referenced variables default to `firstprivate` while in parallel regions they default to `shared` the user must explicitly set the data sharing attributes of all referenced variables in the new task-based code.

new application code is designated as L2TASK. The original code which utilized nested parallelism is the L2TRUE part. For comparison, we include the L2FALSE bars which represent the original code executed with nested parallelism disabled (i.e. the environmental variable `OMP_NESTED` was set to false).

In the plots we vary the number of participating threads per level using up to $M = 12$ threads for the first level and up to $N = 8$ threads in the second level. For both compilers nested parallelism (L2TRUE) boosts performance as long as processors are not heavily oversubscribed. It can be seen that GCC's performance drops for large number of threads, while ICC seems to handle the situation much better, although its performance approximately levels off after the 8×6 configuration. Our approach results in better speedups for more than 8 first-level threads in both cases, confirming the validity of our approach. The lower performance shown in smaller configurations is expected since we only rely on the few first-level threads while nested parallelism is able to utilize all the 16 processors in the system. Finally, in the larger configurations, notice that while the L2TRUE code utilizes all the 16 available processors (albeit with increased overheads), we obtain better speedups with only 12 threads.

3 Overcoming Limitations by Automatic Transformation

In the previous section we presented the core idea behind our method. The proposed code transformation was exemplified using a loop with a `static` schedule. A similar approach can be used for any schedule type, e.g. `dynamic` or `guided`. In these cases, however, the new code does not execute just one chunk of iterations; it should rather be enclosed within another loop that asks continuously for chunks of iterations. Calculating the iteration bounds becomes considerably more complicated as it has to take into account the competition / synchronization among tasks and keep some kind of state in order to hand out the iterations in accordance to the loop schedule. In essence, the user has to re-implement a mini worksharing runtime subsystem in order to cover all possible schedule configurations. This is clearly both undesirable for the user and redundant as far as the compiler is concerned, since all this functionality is already present in its OpenMP runtime library.

Another important issue is that even if the user is determined to do all this work, this will not be enough to make it applicable to all possible cases. The reason is that within the loop body there may exist references to thread-specific quantities, for example,

- the loop body may contain calls to `omp_get_thread_num()` and utilize the thread's ID in computations, or,
- the loop body may access `threadprivate` variables.

The above makes it almost impossible to move the loop's body to independent tasks, as there is no guarantee as to which threads will execute what tasks.

In conclusion, the manual code transformations need extensive programmer involvement and are not applicable in the general case. On the other hand, all the required functionality is already implemented within the runtime library of the OpenMP system. Additionally, the runtime system has access to all the stored thread-specific quantities. It should thus be in position to support the required transformations seamlessly. In this section we describe the actual implementation of this idea in the runtime system of the OMPi compiler.

3.1 The OMPi Compiler

OMPi is an experimental, lightweight OpenMP infrastructure for C. It consists of a source-to-source compiler and a runtime library. The compiler takes as input C code with OpenMP directives and outputs multithreaded C code augmented with calls to its runtime library, ready to be compiled by any standard C compiler. It conforms to V3.0 of the specifications while also supporting parts of the recently announced V3.1 [9].

Here we provide a brief description of portions of OMPi and its tasking implementation that are necessary for our discussion. A more detailed description was given by Agathos et al [1]. The compiler uses *outlining* to move the code residing within a `parallel` or a `task` region to a new function and then, depending on the construct, inserts calls to create a team of threads or a task to execute the code of the new function.

The runtime system of OMPi has a modular architecture in order to facilitate experimentation with different threading libraries. In particular, it is composed of two largely independent layers. The upper layer (ORT) carries all required OpenMP functionality by controlling a number of abstract *execution entities* (EEs). The lower layer (EELIB) is responsible for actually providing the EEs, along with some synchronization primitives. A significant number of EELIBs is available. The default one is built on top of POSIX threads, while there also exists a library which is based on high-performance user-level threads [6].

OMPi provides a tasking layer within ORT which can be used with any EELIB, although the runtime design allows for the latter to provide its own tasking functionality, if desired. Each execution entity (thread) is equipped with a queue (TASK_QUEUE) which is used to store all the pending tasks it has created. OMPi's task scheduler is based on work stealing [2], whereby a thread that has finished executing its own tasks tries to steal tasks from other threads' TASK_QUEUEs. After a new task is created, it is placed in the thread's queue until some thread decides to execute it. Task queues have fixed length, which means that they can store up to a certain number of pending tasks. This number is one of OMPi's runtime parameters, controlled through an environment variable (`OMPI_TASKQ_SIZE`). The manipulation of task queues is based on a highly efficient lock-free algorithm.

When a thread is about to execute its implicit task (parallel region), a new task descriptor is allocated and the task code is executed immediately. Whenever a thread reaches an explicit `task` construct, it can either allocate a new task node and submit the corresponding task for deferred execution, or it can suspend

the execution of the current task and execute the new task immediately; OMPi's default behavior is to choose the former. That is, it implements a *breadth-first* task scheduling policy. It resorts to the second alternative (*depth-first* task execution) when the task queue is full. In that case the thread enters *throttling* mode, where every encountered task is executed immediately to completion. Notice that in this case the current task (although temporarily suspended in favor of the new task) does not enter the task queue, so it can never be resumed by another thread. In effect, all tasks are *tied*. Throttling mode is disabled when 30% of the task queue capacity becomes again available.

3.2 Automating the Process

In order to apply our technique we had to modify the code produced by the OMPi compiler as well as add new functionality to the runtime system. The actual changes in the compiler were rather minimal and limited to the case where a combined `parallel for` construct is encountered. An (identical) outlined function is still created which includes all the code needed for sharing the loop iterations among threads. However, the call to create the team of threads now includes a new parameter to let the runtime know that this is a combined loop construct. This covers nested and orphaned construct cases alike.

The changes in the runtime system (ORT) were more extensive. Whenever a team of threads needs to be created, if the team is going to operate in nesting level > 1 and the parallel region is actually coming from a combined `parallel for` construct[3], then, instead of threads, an equal number of explicit tasks are created. However, as noted previously this is not enough to cover the cases where the user code accesses thread-specific data.

OMPi associates a control block (EECB) with every execution entity it manages. The EECB contains everything ORT needs in order to schedule the thread, including the size of the team, the thread ID within the team, its parallel level etc. The only thread-specific data not actually stored in a threads EECB are `threadprivate` variables. These are are allocated at the team's parent control block (in order to guarantee persistence across parallel regions, as required by the OpenMP rules). The EECB makes them available through a pointer to the parent EECB (thus a tree of EECBs is formed at runtime). In conclusion, everything a running thread requires is serviced through its control block. Whenever a thread starts the execution of a parallel region, ORT assigns a new EECB to it, which is later freed when the team is disbanded.

Based on the above, the main idea behind our implementation is that the produced tasks try to mimic threads. Every task produced (instead of a thread) when a nested combined parallel loop is encountered, carries a special flag along with the ID number the corresponding thread would have. The tasks are inserted as normal in the TASK_QUEUE of the outer-level thread that encountered the nested construct. When such a task is scheduled for execution (either by the same thread or a thief), the flag will cause the following actions:

[3](and if the user allows; a new environmental variable lets the user decide whether the new technique should be applied or not)

- A new EECB is created, as would be done if a new nested thread was created in the first place, updating the tree of EECBs correspondingly.
- The outer-level thread that is about to execute the task assumes temporarily the new EECB and sets its thread ID equal to the ID stored within the task.
- The task becomes tied to this thread.

In essence, an outer level thread while executing the task in question, obtains all the characteristics of the inner level thread that would be created normally. As such it is able to handle thread-specific data accesses, overcoming all the previously mentioned limitations. Notice for example that because the old control block of the thread remains intact in the tree, all information needed to service runtime calls such as `omp_get_level()`, `omp_get_active_level()`, etc, is readily available. When the task execution is finished, the temporary EECB is freed and the thread resumes its original control block, continuing with its normal operation.

3.3 Ordered

The above implementation is able to substitute a nested team of threads by an equivalent set of tasks, for any OpenMP schedule type. However, one of our initial concerns was the possible presence of the `ordered` directive. This particular directive forces ordering dependencies among the iteration executors; when the executors are threads there is no problem whatsoever but what about tasks? Is there a possibility that particular task scheduling sequences lead to deadlock? In all cases but one, the answer is no. This is because even if there is only one thread available to execute the generated tasks, there will always be at least one task active, advancing the iteration count and obtaining the next chunk of iterations. For example, consider the case of `dynamic` schedules; if there is a thread (task) blocked at an ordered directive then there must exist at least one other thread that obtained the (sequentially) previous chunk; eventually the latter will be executed and the turn of the former will come.

The single problematic case is the `static` schedule with specified chunk size. Although it is a matter of implementation, the straightforward way of executing it is by using a double loop; the outer loop iterates over the series of chunks while the inner loop goes over the actual iterations of a particular chunk. As the loops bounds are *pre-calculated* (since for this particular schedule they are not subject to competition among the executors), imposing an `ordered` directive may lead to a deadlocked situation, depending on how tasks are implemented / scheduled.

To see this consider the case of having M (level-1) threads to execute $N > M$ tasks generated by the level-2 parallel loop. When all threads have gone through their first chunk of iterations, they will be blocked at an `ordered` region waiting for their next chunk's turn. However, if tasks are executed on a run-to-

```
delay() {
  volatile i, a;
  for (i=0; i < TASK_LOAD; i++)
    a += i;
}
testpfor() {
  for(i=0; i <= REPS; i++)
    #pragma omp parallel for num_threads(N)
    for (j=0; j < N; j++)
      delay();
}
main() {
  #pragma omp parallel for num_threads(16)
  for (i=0; i < 16;t++)
    testpfor();
}
```

Figure 4: Code for synthetic benchmark

completion basis, the remaining $N - M$ tasks will never be given a change to run and advance the iteration count, resulting in a deadlock.

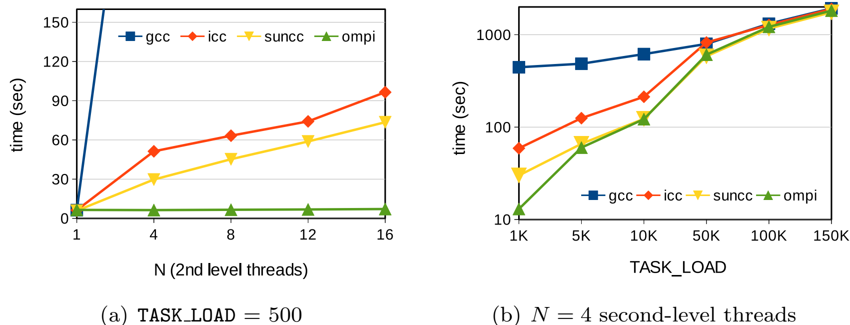
OMPi by default executes tasks to completion and is thus susceptible to this problem. The engineering solution we currently follow is to avoid the problem altogether: if the loop schedule is `static` and an explicit chunk size is given and an `ordered` clause is present, nested parallelism is generated as usual, instead of tasks. We are currently working on the support for OpenMP V3.1 which includes a new `taskyield` directive. Yielding upon an imminent `ordered` block should allow the possibility of other tasks to be executed an thus make progress.

4 Evaluation

We have run several experiments in order to evaluate the performance gains of our implementation. We report here the results obtained on a server with two 8-core AMD Opteron 6128 CPUs operating at 2GHz and a total of 16GB of main memory. The operating system is Debian Squeeze based on the 2.6.32.5 Linux kernel. In our experiments, apart from OMPi, we had the following compilers available: GNU GCC (version 4.4.5-8), Intel ICC (version 12.1.0) Oracle SUNCC (version 12.2). We used "-O3 -fopenmp" flags for GCC, "-fast -openmp" flags for ICC and "-fast -xopenmp=parallel" flags for SUNCC. GCC with the "-O3" flag was used as a back-end compiler for OMPi. For all compilers, the default runtime settings were used. These settings also happened to produce the best results.

4.1 Synthetic Benchmark

Our first experiments aim at showing directly the performance gains possible with our methodology in the given system. A synthetic benchmark is used, measuring the time taken to execute the code shown in Fig. 4. This code is



(a) `TASK_LOAD` = 500 (b) $N = 4$ second-level threads

Figure 5: Synthetic benchmark execution times

based on the EPCC microbenchmarks [3] which are used to estimate OpenMP construct overheads. We instead measure the total execution time. In the main function a team of 16 threads is created and each thread calls the `testpfor()` function once. In there a thread executes REPS times a combined `parallel for` directive, creating N second-level threads, each one performing work, the granularity of which is controlled by the `TASK_LOAD` parameter in the `delay()` function. We used REPS= 100000 and varied the `TASK_LOAD` value.

We present the results in Fig. 5. In Fig. 5(a) we consider fine grain work (`TASK_LOAD` = 500) and vary the number of second-level threads in order to stress the runtime system. The growing number of threads results in considerable overheads that are clearly depicted in the total execution time. Because OMPi avoids creating nested parallelism, it exhibits remarkable stability in its performance, which is only very slightly affected by an increasing number of generated tasks.

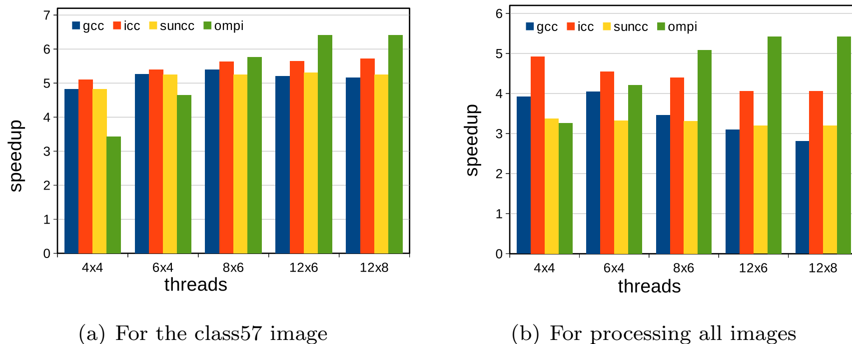
In figure 5(b) we fixed the number of second-level threads to $N = 4$ and varied the work granularity, with `TASK_LOAD` values in the range of 1K to 150K. We use a logarithmic scale due to the wide range of timing results. As expected, for finer grain work our methodology results in significantly faster execution as compared to other compilers. As the work gets coarser, all compilers tend to exhibit similar performance since the task or thread manipulation stops being the performance bottleneck and execution time is dominated by the actual computation. For the coarser load, all compilers execute the benchmark in about 2000 sec.

4.2 Face Detection

As already mentioned in Section 2, we also experimented with a full face detection application, which has been described in detail by Hadjidoukas et al [5]. It takes as input an image and discovers the number of faces depicted in it, along with their position in the image. The code has been parallelized with OpenMP, utilizing nested parallelism in order to obtain better performance than what is

```
for each scale {        /* level 1 */
   for i=1 to 4 {
     <body1>
   }
   for i=1 to 14 {
     <body2>
   }
   for i=1 to 14 {
     <body3>
   }
}
```

Figure 6: Structure of the main computational loop



(a) For the class57 image (b) For processing all images

Figure 7: Face detection results (for each compiler the speedups are calculated in comparison to its own sequential execution time).

possible with only single-level loop parallelization.

In Fig. 6 we outline the structure of the main loop nest of the application. Initially the image is subsampled repeatedly to create a pyramid of different scales, the number of which is dependent on the images size and is usually less than 14. For each scale (this is the first-level loop) a series of convolutional filters and non-linear subsamplings are applied through the 3 nested for-loops. Because of the load imbalance between the different image scales, the level-1 loop is parallelized through a `parallel for` directive with `dynamic` schedule, while for the inner loops a `parallel for` directive with a `static` schedule is applied.

In our experiments we vary the number of participating threads per parallelism level; a configuration of $M \times N$ threads uses M (≤ 12) threads in the first level and N (≤ 8) threads for the second level. In Fig. 7 we show the performance obtained when each of the available compilers was used. We do not include results for single-level parallelization ($N = 1$) as they were inferior to what we obtained when $N > 1$. For these plots the speedups for each compiler are calculated in relation to the sequential execution time obtained by the same compiler so that we can show how it behaves under nested parallelism. In

Table 1: Best execution times and comparison with OMPi when processing all images (speedup is calculated in comparison to the best sequential time overall).

Compiler	Sequential time (sec)	Best configuration	Parallel time (sec)	Speedup	OMPi improvement
GCC	37.329	6x4	9.210	3.219	25.5%
ICC	37.282	12x8	9.163	3.236	25.2%
SUNCC	29.656	4x4	8.778	3.378	21.9%
OMPi	37.329	16x8	6.853	4.327	–

Fig. 7(a) the application used as input a particularly demanding image which contains 57 faces (the 'class57' image from the CMU test set [11]). For obtaining the results in Fig. 7(b) we processed a series of 161 images with varying sizes and faces, one after the other.

All compilers, except OMPi are using nested parallelism, spawning $M \times N$ threads, while OMPi uses only M threads that execute $M \times N$ tasks in total. The results lead to similar conclusions in both plots. For the 4×4 configuration OMPi exhibits the lowest speedup due to the few (4) available threads while all other compilers employ 16 threads in total, potentially exploiting all the 16 cores of the system. On the other hand, when 8 or more threads are used in the first level, OMPi exhibits the highest speedups. ICC exhibits the second best performance and when processing image class57 it attains stable speedups for all thread configurations. For the set of all images ICC exhibits its best behavior when 4×4 threads are used, while for more threads synchronization overheads cause poorer speedups. GCC get its best speedup for image class57 for 8×6 threads, whereas for set of all images maximum speedup is shown for 6×4 threads. SUNCC exhibits similar execution times compared to ICC for both inputs in all thread configurations. The lower speedups shown for SUNCC are due to its shorter sequential execution times compared to all other compilers.

For completeness, in Table 1 we report the best performance attained by each compiler based on absolute execution times. For each compiler, we include the time required for a sequential run, the best observed configuration and the parallel execution time for that configuration. Speedups are then calculated in relation to the lowest sequential execution time, which is achieved using the SUNCC compiler. The last column demonstrates the performance improvement OMPi achieves in comparison to each compiler, based on the parallel execution times. Notice that for a fair comparison we also considered the $16 \times N$ configuration, which, although not advantageous for the rest of the compilers, it gives OMPi the chance to utilize all the available processors. It should be clear that our task-based technique outperforms the conventional implementations which utilize nested thread teams.

5 Conclusion

We have proposed a novel technique for executing nested `parallel for` loops using tasks instead of threads, thereby avoiding the overheads associated with

nested parallelism in such cases. The technique we present is potentially applicable to any OpenMP runtime system that supports tasking and requires almost no changes in the compiler-produced code. It has been implemented in the framework of the OMPi compiler and has been shown to offer significant performance gains. While in this work we were mostly interested in showing the performance potential, as a future work we envisage an adaptive application of our technique. In particular, we believe that appropriate decisions can be made at runtime, depending on the number of active threads; if the active threads are much less than the available system processors it may be more appropriate to create nested threads instead of tasks.

Our technique can be applied to nested `parallel sections` regions without any alterations, as well, and this is currently under implementation in OMPi. It is not applicable, though, to general nested parallel regions. This is because parallel regions produce threads that may contain barrier synchronizations, which are not allowed within tasks. Nevertheless, it seems plausible to investigate this further and we are actually working on this possibility within our PSTHREAD library [6, 1]; this library is based on user-level threads that are used to instantiate both OpenMP threads and OpenMP tasks.

References

[1] Agathos, S.N., Hadjidoukas, P.E., Dimakopoulos, V.V.: Design and Implementation of OpenMP Tasks in the OMPi Compiler. In: Proc. PCI '11, 15th Panhellenic Conference on Informatics. pp. 265–269. IEEE, Kastoria, Greece (Sept 2011)

[2] Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. J. Parallel Distrib. Comput. 37(1), 55–69 (1996)

[3] Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In: Proc. of 1st EWOMP, European Workshop on OpenMP. pp. 99–105. Lund, Sweden (Sept 1999)

[4] Dimakopoulos, V.V., Leontiadis, E., Tzoumas, G.: A portable C compiler for OpenMP V.2.0. In: roc. EWOMP 2003, 5th European Workshop on OpenMP. pp. 5–11. Aachen, Germany (Sept 2003)

[5] Hadjidoukas, P.E., Dimakopoulos, V.V., Delakis, M., Garcia, C.: A high-performance face detection system using OpenMP. Concurrency and Computation: Practice and Experience 21, 1819–1837 (October 2009)

[6] Hadjidoukas, P.E., Dimakopoulos, V.V.: Nested Parallelism in the OMPi OpenMP/C Compiler. In: Proc. Euro-Par '07, 13th Int'l Euro-Par Conference on Parallel Processing. pp. 662–671. Rennes, France (Aug 2007)

[7] Leiserson, C.E.: The Cilk++ concurrency platform. J. of Supercomputing 51, 244–257 (2012)

[8] OpenMP ARB: OpenMP Application Program Interface V3.0 (May 2008)

[9] OpenMP ARB: OpenMP Application Program Interface V3.1 (July 2011)

[10] Reinders, J.: Intel threading building blocks. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edn. (2007)

[11] Rowley, H., Baluja, S., Kanade, T.: Neural network-based face detection. IEEE Trans. on Pattern Analysis and Machine Intelligence 20, 23–28 (1998)