A Microbenchmark Study of OpenMP Overheads Under Nested Parallelism

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

Abstract

In this work we present a microbenchmark methodology for assessing the overheads associated with nested parallelism in OpenMP. Our techniques are based on extensions to the well known EPCC microbenchmark suite that allow measuring the overheads of OpenMP constructs when they are effected in inner levels of parallelism. The methodology is simple but powerful enough and has enabled us to gain interesting insight into problems related to implementing and supporting nested parallelism. We measure and compare a number of commercial and freeware compilation systems. Our general conclusion is that while nested parallelism is fortunately supported by many current implementations, the performance of this support is rather problematic. There seem to exist issues which have not yet been addressed effectively, as most OpenMP systems do not exhibit a graceful reaction when made to execute inner levels of concurrency.

1 Introduction

OpenMP [1] 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. Nested parallelism has been a major feature of OpenMP since its very beginning. As a programming style, it provides an elegant solution for a wide class of parallel applications, with the potential to achieve substantial processor utilization, in situations where outer-loop parallelism simply can not. Despite 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.

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 contemporary OpenMP compilation systems provide some kind of support for nested parallelism, there has been no evaluation of the overheads incurred by such a support. The well known EPCC microbenchmark suite [2, 3] is a valuable tool with the ability to reveal various synchronization and scheduling overheads, but only for single-level parallelism.

In this work, we present a set of benchmarks that are based on extensions to the EPCC microbenchmarks and allow us to measure the overheads of OpenMP systems when nested parallelism is in effect. To the best of our knowledge this is the first study of its kind as all others have been based on application speedups [4, 5, 6, 7] which give overall performance indications but do not reveal potential construct-specific problems.

The paper is organized as follows. In Section 2 we give an overview of OpenMP specification and the current status of various implementations with respect to nested parallelism. In Section 3 we present the microbenchmarks in detail. Section 4 reports on the performance of several OpenMP compilation systems when used to execute our benchmarks. The section also includes a discussion of our findings. Finally, Section 5 concludes this work.

2 Nested Parallelism in OpenMP

The OpenMP specification leaves support for nested parallelism as optional, allowing an implementation to serialize the nested parallel region, i.e. execute it by only 1 thread. In implementations that support nested parallelism, the user can choose to enable or disable it either during program startup through the `OMP_NESTED` environmental variable or dynamically at runtime through an `omp_set_nested()` call. The number of threads that will comprise a team can be controlled by the `omp_set_num_threads()` call. Because this is allowed to appear only in sequential regions of the code, there is no way to specify a different number of threads for inner levels through this call; to overcome this, the current version of OpenMP (2.5) provides the `num_threads(n)` clause. Such a clause can appear in a (nested) `parallel` directive and request that this particular region be executed by exactly `n` threads.

However, the actual number of threads dispatched in a (nested) `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.

In general, it is a recognized fact that the current version of OpenMP has a number of shortcomings when it comes to nested parallelism [7], and there exist issues which

need clarification. Some of them are settled in the upcoming version of the API (3.0), which will also offer a richer functional API for the application programmer.

According to the OpenMP specification, an implementation which serializes nested `parallel` regions, even if nested parallelism is enabled by the user, is considered *compliant*. An implementation can claim *support* of nested parallelism if nested `parallel` regions may be executed by more than 1 thread. Because of the difficulty in handling efficiently a possibly large number of threads, many implementations provide support for nested parallelism but with certain limitations. For example, there exist systems that support a fixed number of nesting levels; some others allow an unlimited number of nesting levels but have a fixed number of simultaneously active threads.

Regarding proprietary compilers, not all of them support nested parallelism and some support it only in part. Among the ones that provide unlimited support in their recent releases are the Fujitsu PRIMEPOWER compilers, the HP compilers for the HP-UX 11i operating system, the Intel compilers [8] and the Sun Studio compilers [9]. Full support for nested parallelism is also provided in the latest version of the well known open-source GNU Compiler Collection, GCC 4.2, through the libGOMP [10] runtime library.

Research/experimental OpenMP compilers and runtime systems that support nested parallelism include MaGOMP, a port of libGOMP on top of the Marcel threading library [11], the Omni compiler [12, 6] and OMPi [13, 14].

3 The microbenchmark methodology

The EPCC microbenchmark suite [2, 3] is the most commonly used tool for measuring runtime overheads of individual OpenMP constructs. However, it is only applicable to single-level parallelism. This section describes the extensions we have introduced to this microbenchmark suite for the evaluation of OpenMP runtime support under nested parallelism.

The technique used 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. Let T_p be the execution time of a program on p processors and T_1 be the execution time of its sequential version. The overhead of the parallel execution is defined as the total time spent collectively by the p processors over and above T_1, the time required to do the "real" work, i.e. $T_{ovh} = pT_p - T_1$. The per-processor overhead is then $T_o = T_p - T_1/p$. The EPCC microbenchmarks [2] measure T_o for the case of single-level parallelism using the method described below.

A reference time, T_r, is first fixed, which represents the time needed for a call to a particular function named `delay()`. To avoid measuring times that are smaller than the clock resolution, T_r is actually calculated by calling the `delay()` function sufficiently many times:

```
for (j = 0; j < innerreps; j++)
  delay(delaylength);
```

and dividing the total time by `innerreps`.

Then, the same function call is surrounded by the OpenMP construct under measurement, which in turn is enclosed within a `parallel` directive. For example, the `testfor()` routine that measures the `for` directive overheads, actually measures the portion shown in Fig. 1 and then divides it by `innerreps`, obtaining T_p. Notice, that because the measurement includes the time taken by the `parallel` directive, `innerreps` is large enough so that the overhead of the enclosing `parallel` directive can be ignored. The overhead is derived as $T_p - T_r$, since the total work done needs actually pT_r sequential time. Of course, 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.

```
testfor() {
  ...
  <start measurement>
    #pragma omp parallel private(j)
    {
      for (j = 0; j < innerreps; j++)
        #pragma omp for
          for (i = 0; i < p; i++)
            delay(delaylength);
    }
  <stop measurement>
  ...
}
```

Figure 1: Portion of the `testfor()` microbenchmark routine.

3.1 Extensions for nested parallelism

To study how efficiently OpenMP implementations support nested parallelism, we have extended both the synchronization and the scheduling microbenchmarks of the EPCC suite. According to our approach, the core benchmark routine for a given construct (e.g. the `testfor()` discussed above) is represented by a *task*. 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 measure their total execution time and compute the global mean of all measured runtime overheads. Our approach is outlined in Fig. 2. 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. 2, if the outer loop (lines 5–7) 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 2–4) 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

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

1       t0 = getclock();
2       #ifdef NESTED_PARALLELISM
3       #pragma omp parallel for schedule(static,1)
4       #endif
5       for (task_id = 0; task_id < p; task_id++) {
6         (*originalfunc)(task_id);
7       }
8       t1 = getclock();

        <compute global statistics>
        <print construct name, elapsed time (t1-t0), 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 2: Extended microbenchmarks for nested parallelism overhead measurements

each member of the team executes exactly one task, a static schedule with chunksize of 1 was chosen at line 3. 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)`.

By measuring the aggregated execution time of the tasks, we use the microbenchmark as an individual application. This time does not only include 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 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. 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 (see Fig. 1). To resolve these issues, we increase the number of internal repetitions (`innerreps`) for each microbenchmark, so as to be able to reach the same confidence levels (95%). A final subtle point is that when the machine is oversubscribed, each processor will be timeshared among multiple threads. This leads to an overestimation of the overheads because the microbenchmarks account for the sequential work (T_r) multiple times. We overcame

this by decreasing `delaylength` so that T_r becomes negligible with respect to the measured overhead.

4 Results

All our measurements were taken on a Compaq Proliant ML570 server with 4 Intel Xeon III single-core CPUs running Debian Linux (2.6.6). Although this is a relatively small SMP machine, size is not an issue here. Our purpose is to create a significant number of threads; as long as a lot more threads than the available processors are active, the desired effect is achieved. We provide performance results for two free commercial and three 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 GCC 4.2.0, Omni 1.6 and OMPi 0.9.0. As Omni and OMPi are source-to-source compilers, we have used GCC as the native back-end compiler for both of them. In addition, because OMPi is available with a multitude of threading libraries, we have used two different configurations for it, namely OMPi+POSIXand OMPi+PSTHREADS. The first one uses the default runtime library, based on POSIX threads which, although optimized for single-level parallelism, provides basic support for nested parallelism. The second one uses a high-performance runtime library based on POSIX threads and portable user-level threads [14].

Most implementations start by creating an initial pool of threads, usually equal in size to the number of available processors, which is 4 in our case. Because the number of threads in the second level is implementation dependent, in all our experiments we have explicitly set it to 4 through an `omp_set_num_threads(4)` call and we have disabled the dynamic adjustment of the number of threads. I.e., when executing the second level of parallelism, there are in total $4 \times 4 = 16$ active threads. However, some implementations cannot handle this situation. In particular, the Omni compiler and OMPi+POSIX cannot create more threads on the fly, even if needed; they support nested parallelism as long as the initial pool has idle threads, otherwise nested parallel regions get serialized. To overcome this problem, for those two implementations we set the `OMP_NUM_THREADS` environmental variable equal to 16 before executing the benchmarks, so that the initial pool is forced to have 16 threads; the `omp_set_num_threads(4)` call then limits the outer level to exactly 4 threads, while all 16 threads are utilized in the inner level. We have, however, been careful not to give those two implementations the advantage of zero thread creation overhead (since with the above trick the 16 threads are pre-created), by including a dummy nested parallel region at the top of the code. This way, all implementations get a chance to create 16 threads before the actual measurements commence.

A selection of the obtained results is given in Figs 3–4, for the synchronization and scheduling microbenchmarks. Fig. 3 includes the overheads of all six systems for the `parallel`, `for`, `single` and `critical` constructs. Each plot includes the single-level overheads of each system for reference.

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, al-

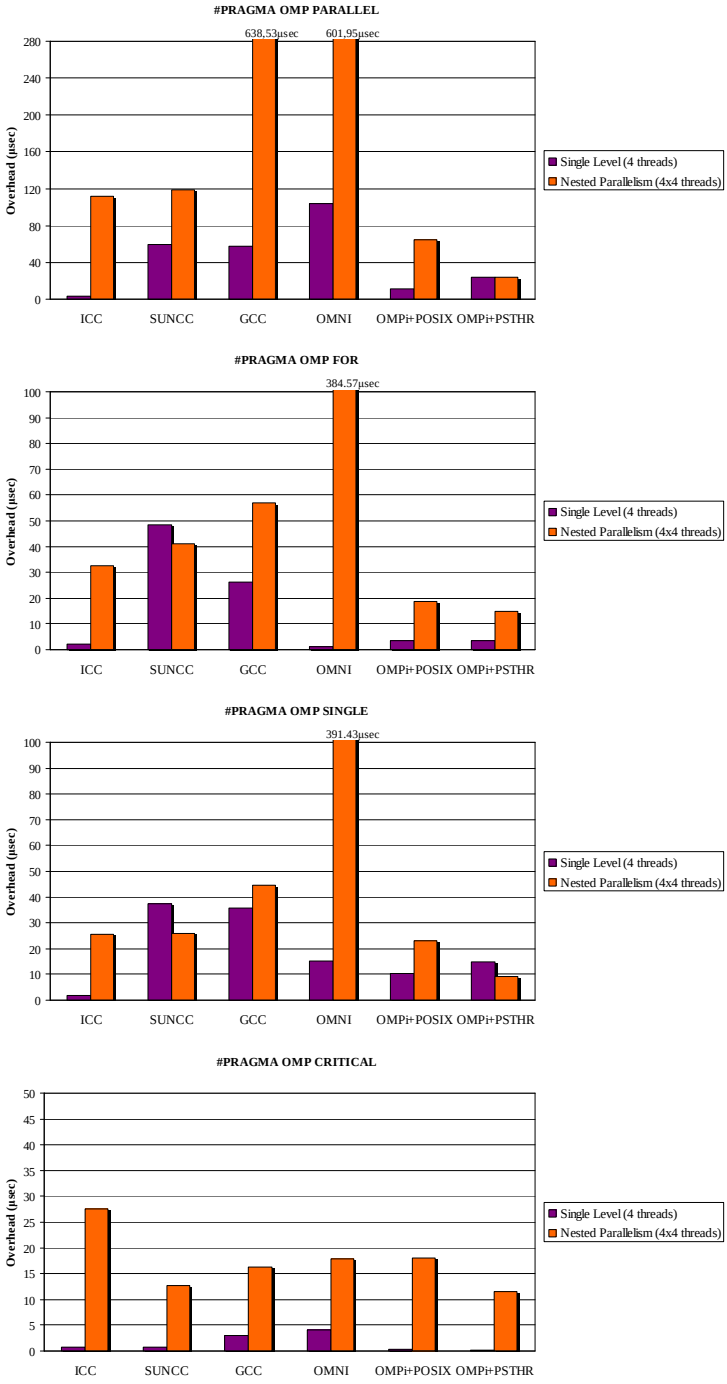

Figure 3: Overheads for `parallel`, `for`, `single` and `critical`

though ICC remains quite fast. For all three 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 [8]. 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.

Similar behavior is seen for the `for` and `single` constructs, except that GCC shows significant but not excessive increase. The Sun compiler seems to handle loop scheduling quite well showing a decrease in the actual overheads. This, combined with the decrease in the `single` overheads, reveals efficient team management since both constructs incur mostly inter-team contention. Especially in the `single` construct, OMPi+PSTHREADS shows the advantage of user-level threading: inner levels are executed by user-level threads, which mostly live in the processor where the parent thread is, eliminating most inter-team contention and the associated overheads. In contrast, the (unnamed) `critical` construct incurs global contention since all threads from all teams must compete for a single lock protecting the critical code section. Overheads are increased significantly in all systems, suggesting that *unnamed* `critical` constructs should be avoided when nested parallelism is required.

Fig. 4 includes results from the scheduling microbenchmarks. For presentation clarity, we avoided reporting curves for a wide range of chunksizes; instead we include only results for static, dynamic and guided schedules with a chunksize of 1, which represent the worst cases, with the highest possible scheduling overhead. Scheduling overheads increase, as expected, for the static and guided schedules in the case of nested parallelism. However, the overheads of the dynamic scheduling policy seem to increase at a slower rate and in some cases (SUNCC, GCC and OMPi+PSTHREADS) actually decrease, which seems rather surprising. This can be explained by the fact that for this particular scheduling strategy and with this particular chunk size, the overheads are dominated by the excessive contention among the participating threads. With locality-biased team management, which groups all team threads onto the same CPU, and efficient locking mechanisms, which avoid busy waiting, the contention has the potential to drop sharply, yielding lower overheads than in the single-level case. This appears to be the case for the Sun Studio and GCC compilers. 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 processor. However, as the chunksize increases, assigned jobs become coarser and any gains due to contention avoidance vanish. This is confirmed in the last plot of Fig. 4; with a chunksize of 8 all implementations show increased overheads with respect to the single-level case.

In Fig. 5 we present the results of our next experimentation: we delved into discovering how the behavior of our subjects changes for different populations of threads. We fixed the number of first-level threads to 4 but changed the second-level teams to consist of 2, 4 and 8 threads, yielding in total 8, 16 and 32 threads on the 4 processors. Because this was only possible using the `num_threads()` clause (an OpenMP V.2.0 addition), Omni was not included, as it is only V.1.0 compliant. Fig. 5 contains one plot per compiler, including curves for most synchronization microbenchmarks. The results confirmed what we expected to see: increasing the number of threads in the

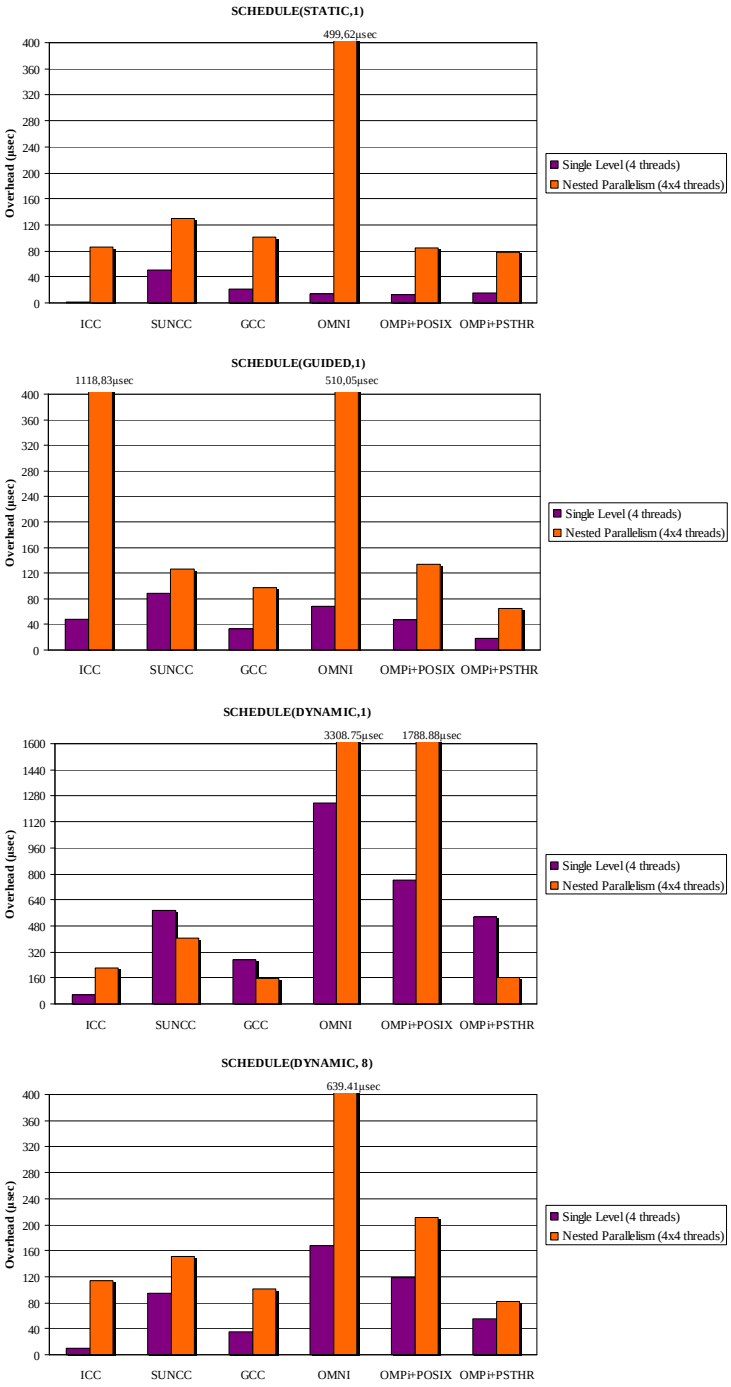

Figure 4: Scheduling overheads for `static`, `guided` and `dynamic`.

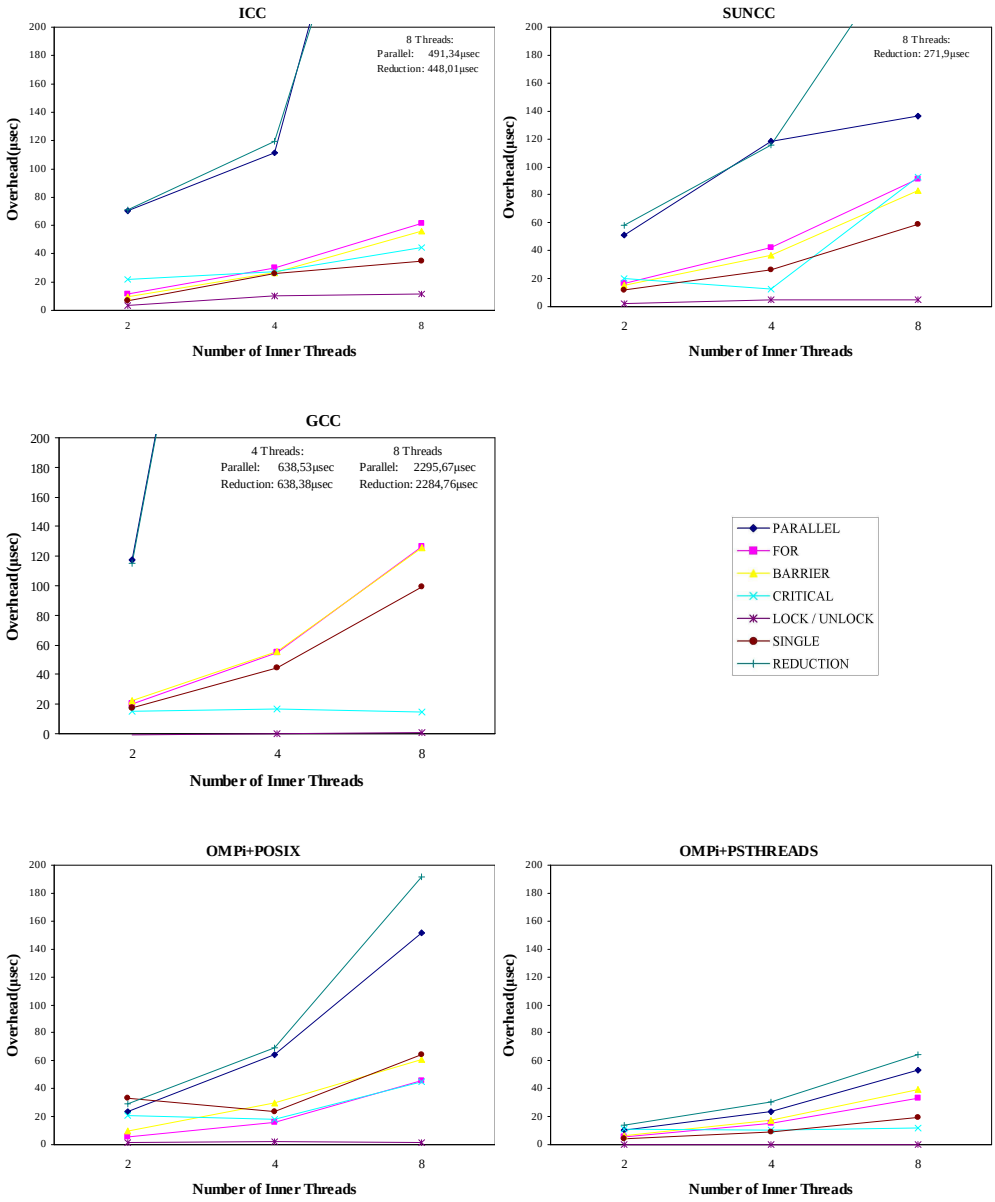

Figure 5: Overheads per compiler, for increasing team sizes at the second level of parallelism

second level leads to increased overheads. Due to space limitations, we cannot comment on every aspect of the plots but we believe that they present the situation very vividly. It is enough to say that for some implementations things seems to get out of control, especially for `parallel` and `reduction`. By far, the most scalable behavior is exhibited by the OMPi+PSTHREADS setup, although in absolute numbers the Intel compiler is in many cases the fastest.

5 Conclusion

In this paper we presented an extension to the EPCC microbenchmark suite that allows the measurement of OpenMP construct overheads under nested parallelism. Using this extension we studied the behavior of various OpenMP compilation and runtime systems when forced into inner parallel regions. We discovered that many implementations have scalability problems when nested parallelism is enabled and the number of threads increases well beyond the number of available processors. This is most probably due to the kernel-level thread model the majority of the implementations use. The utilization of kernel threads introduces significant overheads in the runtime library. 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.

Although our study was limited to two nesting levels, it became clear that studying deeper levels would only reveal worse behavior. It is evident that there are several design issues and performance limitations related to nested parallelism support that implementations have to address in an efficient way. In the near future we plan to expand the microbenchmark suite appropriately so as to be able to study the overheads at any arbitrary nesting level.

Acknowledgement

This work was co-funded by the European Union in the framework of the project "Support of Computer Science Studies in the University of Ioannina" of the "Operational Program for Education and Initial Vocational Training" of the 3rd Community Support Framework of the Hellenic Ministry of Education, funded by national sources and by the European Social Fund (ESF).

References

[1] OpenMP Architecture Review Board: OpenMP C and C++ Application Program Interface, Version 2.5. (May 2005)

[2] Bull, J.M.: Measuring Synchronization and Scheduling Overheads in OpenMP. In: Proc. of the 1st EWOMP, European Workshop on OpenMP, Lund, Sweden (1999)

[3] Bull, J.M., O'Neill, D.: A Microbenchmark Suite for OpenMP 2.0. In: Proc. of the 3rd EWOMP, European Workshop on OpenMP, Barcelona, Spainn (2001)

[4] Ayguade, E., Gonzalez, M., Martorell, X., Jost, G.: 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 (2004)

[5] Blikberg, R., Sorevik, T.: 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 (1999)

[6] Tanaka, Y., Taura, K., Sato, M., Yonezawa, A.: 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)

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

[8] Tian, X., Hoeflinger, J.P., Haab, G., Chen, Y.K., Girkar, M., Shah, S.: A compiler for exploiting nested parallelism in OpenMP programs. Parallel Computing **31** (2005) 960–983

[9] Sun Microsystems: Sun Studio 12: OpenMP API User's Guide (2007) PN819-5270.

[10] Novillo, D.: OpenMP and automatic parallelization in GCC. In: Proc. of the 2006 GCC Summit, Ottawa, Canada (2006)

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

[12] Sato, M., Satoh, S., Kusano, K., Tanaka, Y.: Design of OpenMP Compiler for an SMP Cluster. In: Proc. of the 1st EWOMP, European Workshop on OpenMP, Lund, Sweden (1999)

[13] Dimakopoulos, V.V., Leontiadis, E., Tzoumas, G.: A Portable C Compiler for OpenMP V.2.0. In: Proc. of the 5th EWOMP, European Workshop on OpenMP, Aachen, Germany (2003)

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