Performance Analysis Techniques for Task-Based OpenMP Applications *

Dirk Schmidl[1], Peter Philippen[2], Daniel Lorenz[2], Christian Rössel[2], Markus Geimer[2], Dieter an Mey[1], Bernd Mohr[2], and Felix Wolf[1,2,3]

[1] RWTH Aachen University, 52056 Aachen, Germany
[2] Jülich Supercomputing Centre, 52425 Jülich, Germany
[3] German Research School for Simulation Sciences, 52062 Aachen, Germany

Abstract.

Version 3.0 of the OpenMP specification introduced the task construct for the explicit expression of dynamic task parallelism. Although automated load-balancing capabilities make it an attractive parallelization approach for programmers, the difficulty of integrating this new dimension of parallelism into traditional models of performance data has so far prevented the emergence of appropriate performance tools. Based on our earlier work, where we have introduced instrumentation for task-based programs, we present initial concepts for analyzing the data delivered by this instrumentation. We define three typical performance problems related to tasking and show how they can be visually explored using event traces. Special emphasis is placed on the event model used to capture the execution of task instances and on how the time consumed by the program is mapped onto tasks in the most meaningful way. We illustrate our approach with practical examples.

1 Introduction

To harness the available performance of today's multi-core systems, applications need to make efficient use of the available parallelism. Cores sitting idle, for example in communication calls waiting for data to arrive or in synchronization operations due to load imbalance, waste resources and reduce the overall performance of the application. However, optimizing load balance is often a non-trivial undertaking, especially since the behavior of the application may change when ported to a different architecture or executed on a different number of processor cores.

To address this situation, the tasking construct was introduced with OpenMP 3.0 [3]. Using tasks, the programmer is able to express parallelism in his code at a much finer level of detail. Instead of specifying a single command stream per thread, as with the traditional parallel and work-sharing constructs, the programmer can now decompose his program into smaller tasks and specify dependencies between creator tasks and their children. The defined tasks are assigned to the available threads by the runtime system. This approach is supposed to automatically improve load balancing, although it incurs

* This material is based upon work supported by the German Federal Ministry of Research and Education (BMBF) under Grant No. 01IS07005 and by the Department of Energy under Grant No. DE-SC0001621.

Published in "Proc. of 8th IWOMP 2012", LNCS 7312, pp. 196–209, Springer, 2012.
The original publication is available at www.springerlink.com

additional overhead in the runtime system. Moreover, it poses new challenges not only for developers, but also for performance-analysis tools used for tuning applications.

In our earlier work, we introduced a portable method to distinguish individual task instances and to track their suspension and resumption using event-based instrumentation [12]. A prerequisite for this approach is that tied tasks are used or untied tasks which are only suspended at task scheduling points. Based on this method, we present initial performance-analysis concepts in a trace-based analysis workflow. Specifically, we make the following contributions:

- We define three performance problems related to tasking.
- We describe an extension of the Open Trace Format 2 (OTF2) [6] event model to record constituents of these performance problems in event traces. This determines how task instances are represented in the control flow of individual threads.
- We show how time or other performance-related metrics are attributed to tasks and threads.
- We demonstrate our concepts using benchmarks and a real-world application.

The paper is organized as follows: We review related work in Section 2. After discussing typical performance problems in Section 3, we cover the OTF2 event model in Section 4. Next, we explain the representation of task instances and the attribution of execution time in Section 5. Practical examples follow in Section 6. Finally, in Section 7, we discuss progress and limitations, and present future work.

2 Related Work

Since OpenMP is such a commonly used parallel programming interface, there is already a body of work addressing performance analysis and optimization. Many current performance analysis tools support the measurement and analysis of performance data related to OpenMP constructs. Tools based on instrumentation, such as Scalasca [8], TAU [14], and Vampir [10], utilize the source-to-source instrumenter OPARI [13] to capture OpenMP-specific events. However, currently none of them provides support for tasks, mainly because the event stream of a thread may result in a sequence of task-instance fragments, which can only be analyzed if the overall task instance to which those fragments belong can be identified.

Fürlinger et al. [7] were the first who profiled tasks using instrumentation. However, their initial work provides no mechanism to identify task instances. Lorenz at al. [12] presented an instrumentation mechanism to identify task instances via source-code instrumentation of task-related constructs. This mechanism was demonstrated via a prototypical extension of OPARI. In the meantime, the successor OPARI2 [2] was released, which—among other improvements—contains a production version of this instrumentation mechanism. Our work builds upon the OPARI2 instrumentation and uses it as a prerequisite for performance measurements.

Instead of instrumenting the code directly, other tools such as HPCToolkit [1] apply statistical sampling. In this way, they obtain the call-stack and hardware counters in regular intervals. Moreover, Sun proposed a compiler interface [9] to obtain OpenMP-related data for performance analysis. Lin and Mazurov [11] extended this proposal

to support tasking and implemented a prototype based on the Sun Studio Performance Analyzer. However, whereas they focus on the acquisition of performance data, our work focuses on their analysis.

3 Performance Problems Related to Tasking

In task-parallel programs, typically many more task instances than compute resources exist. Consequently, we cannot expect all task instances to be executed in parallel. Tasks which have to wait at a synchronization point do therefore not necessarily indicate a performance drawback. In most HPC applications, the number of active threads is a good indication for the number of available compute resources, as most applications start one thread per core they want to use. Accordingly, all threads can be active at the same time. What needs to be investigated, even in tasking programs, is whether all threads are doing useful work all the time. Here, useful work means everything except spending time in the OpenMP runtime or doing nothing. The following three performance problems related to tasking can lead to situations where threads waste compute resources.

Too Finely Grained Task Parallelism. Overhead spent in the OpenMP runtime to create a task or to suspend and resume it should be avoided if possible. If the execution time of a task is very small, this overhead can consume more CPU cycles than the task's actual execution. In this case, it would be more efficient to execute the task's body immediately without separating it into a task. The overhead to create and manage a task, of course, depends on many different factors, such as the hardware, the compiler, the data-sharing attributes of the task, and so on. Thus, we cannot quantify precisely when it is beneficial to create a task.

Too Coarsely Grained Task Parallelism. In contrast to the previous situation, creating only a few very large tasks may result in load imbalance. For example, if 12 equally sized large tasks are created and eight threads are used, half of the threads will execute two tasks and the rest will only execute one. Even if there is a task for every thread, sometimes there might not be enough to smooth differences in the runtime of individual tasks, which can depend on dynamic conditions.

Task-Creation Bottleneck. When a lot of threads execute tasks while only a few threads create them, the creation of tasks can become the bottleneck. This can happen, for example, when tasks are created in a `single` region by just one thread. For `n` worker threads, the master thread must produce the tasks at least `(n-1)` times faster than they are executed by workers. This situation is commonly known in master-worker approaches where the master can become the performance bottleneck if the number of workers is too large. Another reason why not enough tasks are created might be a shortage of available parallelism in dynamic algorithms.

4 The OTF2 Task Event Model

Before any performance analysis of an application can be done, information about its runtime behavior has to be collected. For this purpose, the work presented in this paper leverages the Score-P [2] instrumentation and measurement system. To instrument OpenMP directives, Score-P utilizes the source-to-source instrumenter OPARI2, using the technique presented in [12] for task-related constructs. In tracing mode, which forms the basis of this work, the instrumentation hooks inserted by OPARI2 trigger the generation of events in the Open Trace Format 2 (OTF2) [6]. But before describing its task-specific details, we first give a brief overview of OTF2.

OTF2 stores concurrent events in separate event streams per thread of execution, representing its runtime behavior. Common event types are entering/leaving a function, sending/receiving a message, or creating/destroying an OpenMP thread team. Each event includes a timestamp as well as additional event-specific data, such as the source-code region being entered or the number of bytes being transferred. To avoid redundancy in the data being stored, static entities (so-called definitions) such as information about source-code regions are stored only once and referenced using numerical identifiers. In addition, OTF2 uses an efficient encoding scheme for these identifiers and other attributes to compress the event data on-the-fly.

To encode task-specific behavior, the "traditional" records provided by OTF2, for example, `Enter`/`Leave` for entering or leaving a source-code region and `OmpFork`/`OmpJoin` for creating or destroying an OpenMP thread team, do not suffice. Therefore, new event types need to be introduced. A careful analysis of the performance deficiencies presented in Section 3 reveals that two types of actions are relevant to analyze the efficiency of task parallelism: the creation as well as the execution of a task. In the following, we describe which events are generated by those actions and which event attributes are required for our analyses.

When a task is created, the OpenMP runtime system basically has two choices: the task can either be executed immediately or queued for later execution. In both cases, it is essential for a measurement system to be able to identify each task instance. That is, for each task being created, we generate a corresponding `OmpTaskCreate` event and attach a unique numerical task identifier to it. The task identifier zero is reserved for the implicit task for which no `OmpTaskCreate` event is generated.

When a task starts its execution—either immediately or when dequeued from the task queue—the measurement system needs to be notified in order to be able to map all following events onto the task which generates them. For this purpose, we use the task identifier assigned during task creation. Obviously, the same notification is required when the execution of one task is suspended and another task is resumed, that is, a task switch occurs. As the begin of a task's execution is basically also a task switch (either switching from the implicit task or from another task which was suspended or finished its execution), we use only a single event to encode this behavior. As the identifier of the task previously being executed is implicitly known, the `OmpTaskSwitch` event carries only the task ID of the task being started or resumed, respectively.

Finally, to allow the measurement system to clean up its internal task-specific data structures, the completion of a task needs to be identified. For this reason, the

```
1  Enter("OMP task", metrics, timestamp);
2  OmpTaskCreate(new_task_id, timestamp);
3
4  #pragma omp task
5  {
6      OmpTaskSwitch(new_task_id, timestamp);
7      Enter("OMP task structured block", metrics, timestamp);
8
9          // Do some useful work...
10
11     Leave("OMP task structured block", metrics, timestamp);
12     OmpTaskComplete(new_task_id, timestamp);
13 }
14
15 if (current_task_id != old_task_id)
16     OmpTaskSwitch(old_task_id, timestamp);
17 Leave("OMP task", metrics, timestamp);
```

Fig. 1. OTF2 events generated for an OpenMP `task` construct.

`OmpTaskComplete` event is introduced, also providing the task identifier of the task that has just finished its execution.

As can be seen, the identification of task instances via task identifiers is essential for our event model. However, the OpenMP standard does not yet require runtime systems to provide such identifiers. We therefore rely on the task instrumentation provided by OPARI2, which implements a portable method to track task identifiers for tied tasks, as well as untied tasks that are suspended only at implied scheduling points.

Figure 1 illustrates when the different events will be generated for an OpenMP `task` construct. As can be seen, task creation is surrounded by a conventional `Enter`/`Leave` event pair (lines 1 and 17). Inside, the task creation is recorded by the generating task, assigning a new task identifier (line 2). The `OmpTaskSwitch` event before leaving the creation region is only generated in the case one or more tasks have been executed at the implicit task scheduling point during creation (lines 15/16). The task execution itself is surrounded by an (unconditional) `OmpTaskSwitch` and an `OmpTaskComplete` event (lines 6 and 12), as well as an `Enter`/`Leave` pair for the task's structured block (lines 7 and 11).

For other task switching points (i.e., `taskwait` as well as implicit and explicit barriers) the event generation is depicted in Figure 2, using the `taskwait` directive as an example. Here, a conventional region is created for the construct itself (lines 1 and 7), and optionally an `OmpTaskSwitch` event is generated in case another task was executed in between (lines 5/6).

Note that time spent in either a `task` creation, `barrier`, or `taskwait` region is not necessarily a bottleneck, as these regions can also include the execution of tasks. Therefore, the time spent executing other tasks needs to be subtracted from the total time spent in these regions to compute the real waiting time.

5 Task Interruption

During the analysis of tasks, special care has to be taken when tasks are suspended and resumed. In this section, we discuss how analysis tools can handle `OmpTaskSwitch` events. The example code in Figure 3 illustrates problems regarding task suspension and resumption. Note that the `taskwait` statements in Figure 3 serve only as additional task scheduling points.

Both functions `f1` and `f2` in the example do exactly the same, they call `do_work` and run into a `taskwait` statement. A thread executing this code could create both tasks and push them into the task queue. At the barrier it might execute them in the order shown in Figure 3. First, it starts the execution of `task1` and suspends it in the `taskwait` statement, then completely executes `task2` before it resumes `task1`.

The rectangles in Figure 3 illustrate the times spent in every function. The length of the rectangles is directly proportional to the time spent in the region. Although both tasks actually do the same, the execution of `task1` and `f1` takes much longer than the execution of `task2` and `f2`, because `task1` was suspended in between. This is misleading to the programmer. Actually, the suspension of `task1` also suspended the execution of `f1`, so the time for `f1` should not include the execution of `task2`. We decided to virtually suspend all functions and regions in the task when the task is suspended and resume them later along with the task. Figure 4 shows the resulting event stream. `Task1` is split in two intervals by the suspension. This clearly shows that `task2` is not part of `task1`.

As a proof of concept, we implemented a post-processing tool to apply this approach to OTF2 traces. The tool duplicates the trace and inserts at task switch regions corresponding leave events for the suspended task and region enter events for the resumed task. Of course, rewriting the trace is too much overhead for traces of realistic size but it is sufficient to further investigate the concept. Later on, an analysis tool can generate the events on-the-fly when reading the trace.

6 Evaluation

As mentioned earlier, the tracing capabilities described in this paper were implemented as part of Score-P, while the handling of task switches was implemented in a post-processing tool for OTF2 traces generated by Score-P. Here, we demonstrate that our

```
1  Enter("OMP taskwait", metrics, timestamp);
2
3  #pragma omp taskwait
4
5  if (current_task_id != old_task_id)
6      OmpTaskSwitch(old_task_id, timestamp);
7  Leave("OMP taskwait", metrics, timestamp);
```

Fig. 2. OTF2 events generated for an OpenMP `taskwait` construct.

event model is adequate to allow the identification of the performance problems introduced in Section 3. Our evaluation is based on both kernel benchmarks as well as a real-world application.

Kernel-Benchmarks To show that the performance problems outlined ealier can be detected, we wrote artificial test programs for all three performance problems:

- A program that creates 10 very large tasks and represents the problem of coarsely grained tasks.
- A program that creates many finely grained tasks.
- A program that uses a master-worker approach. Here the master produces tasks sufficiently fast for a few threads but becomes a performance bottleneck for a larger number of threads.

After instrumenting these kernels and measuring their execution behavior using Score-P configured in tracing mode, we applied our post-processing tool to the gen-

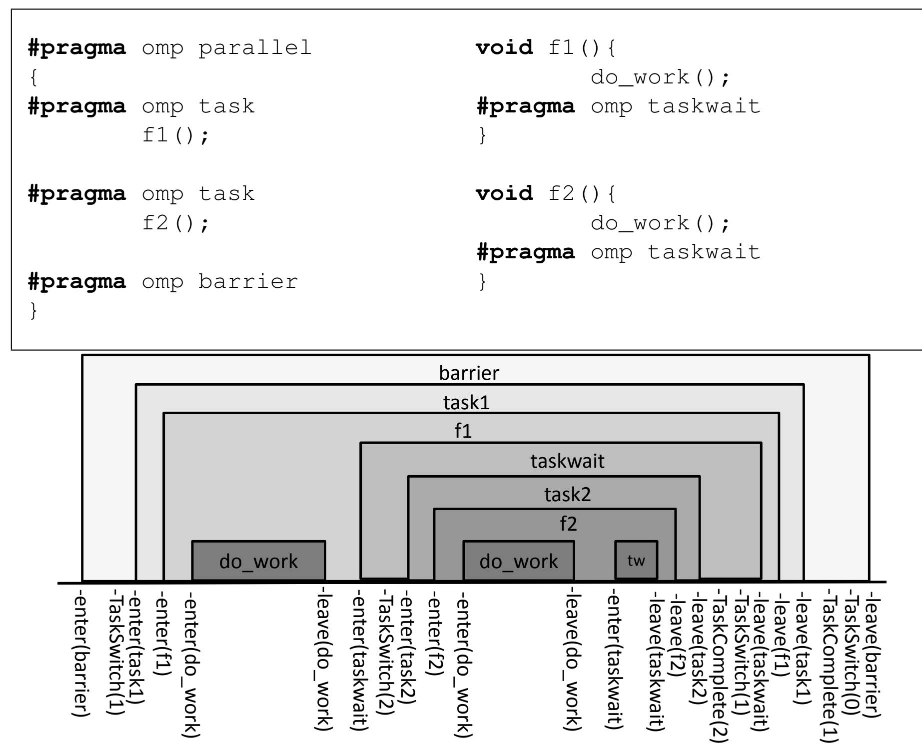

Fig. 3. Top: Code to generate two tasks, one calling `f1` and the other one calling `f2`. Both functions do exactly the same. Bottom: Example execution sequence for this code with active functions shown as rectangles. `Task1` is interrupted when `task2` begins. The rectangles indicate, that `task1` and `f1` have a much longer execution time than `task2` and `f2`.

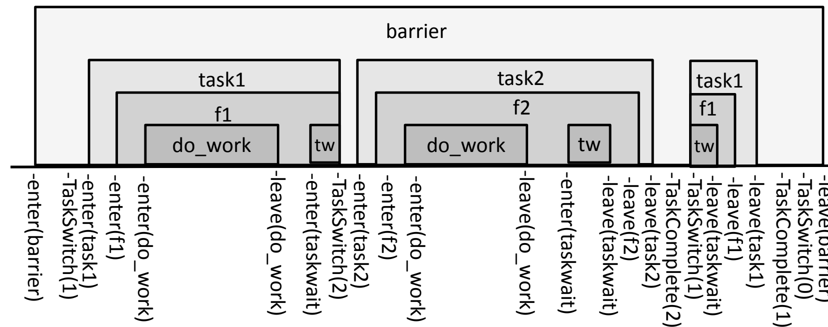

Fig. 4. Execution sequence of the above-mentioned example code with one thread only. Here all functions are interrupted when `task1` is interrupted. `Task1` and `task2` seem to take the same time, now.

erated OTF2 traces. The resulting modified trace files were then visualized using the graphical trace browser Vampir [10]. In the following, regions called `task_X_Y` indicate a task that was created by thread `X` and whose identifier is `Y`. Regions named `!$omp task` indicate task creation overhead.

Figure 5 shows a Vampir screenshot for the first test program creating very large tasks. In the timeline view at the top, it is clearly visible that two threads execute two tasks, whereas the rest of the threads only execute a single task. Therefore, six threads spend a significant amount of time in the `!$omp implicit barrier` region, waiting for the two remaining threads to finish. The function summary view at the bottom displays the exclusive execution time spent in different regions, highlighting the performance bottleneck of this kernel. The program spends 0.6 seconds from a total of 1.6 seconds in the barrier which can be considered substantial overhead.

The corresponding displays for the second test program generating many finely grained tasks is shown in Figure 6. Here, we zoomed in on a smaller interval to see more details. The function summary chart gives again a first indication of suboptimal performance. It can be seen that a significant fraction of the wall-clock time is used for task creation (i.e., spent in `OMP_TASK`), while the fraction of actual workload execution seems to be minor. Looking closer at the timeline view, we can see that the individual tasks take about 50 μs, while the creation of one task takes about 5 ms or more. The program could therefore be optimized since immediate execution of the task body would be much faster than creating separate tasks.

The third kernel implements a master-worker approach where one thread creates many tasks and all other threads execute them. Figure 7 shows the timeline views for two different thread-team sizes. At the top, the timeline of an execution with four threads is shown. Thread 0 is continuously creating tasks and the other threads are executing them. Since threads 1-3 are busy executing the tasks and spend only a very small fraction of time in the barrier between task executions, the overhead spent in the OpenMP runtime is quite low.

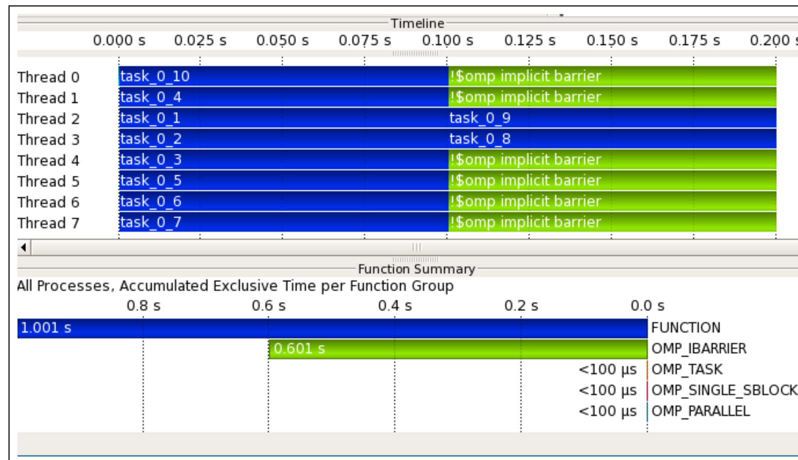

Fig. 5. Vampir screenshot illustrating how too coarsely grained tasks can be detected. In the timeline view two threads execute two task whereas the other threads execute only one task and wait in the barrier.

At the bottom, the timeline of an execution with 16 threads is shown. Here, a different behavior can be observed. Thread 0 is still creating tasks all the time, but many of the other threads are waiting in the barrier without executing any tasks. Immediately after a task has been created, a thread starts executing it. For example, after thread 0 finished creating `task_0_102`, thread 10 stops idling and executes it. Shortly afterwards, thread 6 picks up `task_0_103` and thread 4 executes `task_0_104`. This demonstrates that there are not enough tasks available for all threads. In this situation, the developer should think about a different task-creation approach, such as creating tasks in parallel or switching to larger tasks to fully utilize all available threads.

FIRE Finally, we want to examine how our approach allows task execution to be represented in call trees using a real-world example. The Flexible Image Retrieval Engine (FIRE) [4] was developed by the Human Language Technology and Pattern Recognition Group of RWTH Aachen University. FIRE is used to compare k query images to an image database, identifying those images that are close to the query images. The first parallelization of the FIRE code used nested parallelism with two levels [15]. Here, we are using a modified version using tasking instead. For every query image, a separate task is created. Inside these tasks, every comparison of a query picture and one element of the database constitutes another task. This approach is a bit more flexible than the nested OpenMP version since every thread can work on any task. For nested parallelism, it was necessary to assign a fixed number of threads to the inner regions. Our test case requires searching for two query images in a database of 1000 images.

Similar to the approach used for the kernel benchmarks, we instrumented the task-based FIRE code and generated an OTF2 trace using Score-P. After applying our post-processing tool, however, we analyzed the resulting trace files using a prototype of the

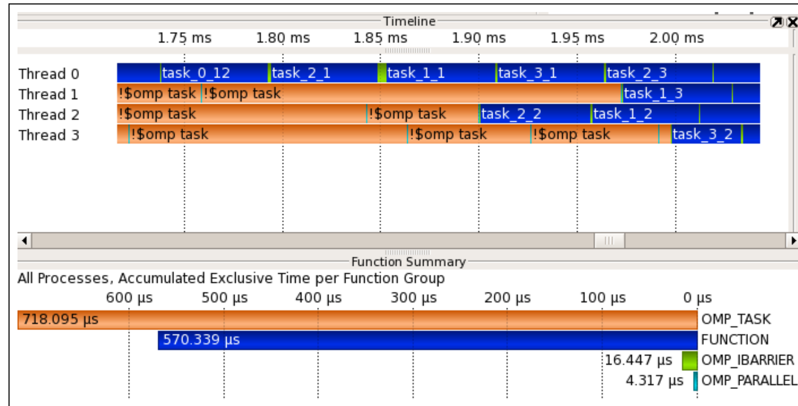

Fig. 6. Vampir screenshot illustrating how too finely grained tasks can be detected. The task creation (`!$omp task`) regions consume more time than the task execution (`task_X_Y`) regions.

automatic trace analyzer of Scalasca [8] which is capable of handling OTF2 traces. The analysis result is shown in the CUBE display in Figure 8. The left column shows different metrics derived from the trace data, with the visit count being selected, whereas the right column shows the system tree, i.e., the machine, the process, and all the threads being used. In the middle column, the call tree of the application is shown.

The call tree shows a parallel region in `main -> Server::batch`. Inside the parallel region, there is a `single` construct where tasks are created and an implicit barrier at the end where the tasks are executed. The visit count indicates that only two tasks are created in the single construct, that is, one task for every query image. If we take a closer look at the tasks executed in the implicit barrier, we can identify these two tasks there (`task_0_1` and `task_0_2`). All the other subtasks, which were created by these two tasks, also appear under the implicit barrier, since the threads waiting in this barrier executed them.

Overhead Analysis After having shown that our approach is capable of identifying the performance problems discussed in Section 3 and that it is also applicable to real-world application codes, we now examine the measurement overhead introduced by our instrumentation. For this purpose, we use the Barcelona OpenMP Task Suite (BOTS) [5], a set of benchmark codes for OpenMP tasking developed by Duran et al. We performed several test runs of these benchmarks on the Juropa[1] cluster at Jülich Supercomputing Centre, consisting of dual-sockel boards with Intel Xeon X5570 quad-core processors. We compared the runtime of the instrumented and uninstrumented versions of the BOTS benchmark codes using eight threads and determined the overhead introduced. The runtime of each benchmark was measured 10 times and the minimum runtime out of these runs is shown in Table 1.

[1] http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/JUROPA_node.html

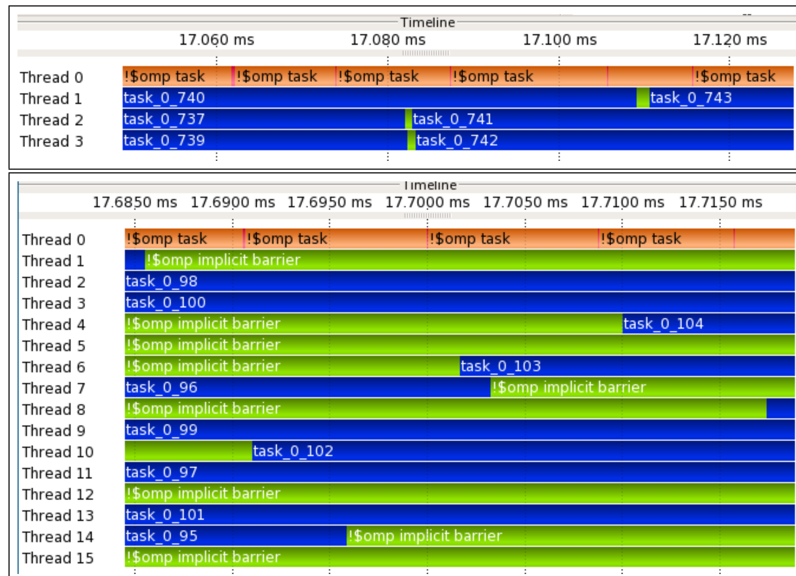

Fig. 7. Vampir screenshot illustrating how a task-creation bottleneck can be detected. Both timelines show the same program where a single thread creates tasks for all other threads. Top: With four threads every worker thread is busy, executing tasks. Bottom: With 16 Threads some threads are idle and wait for tasks from the master thread.

Obviously, there are differences in the overhead observed. Some tests show nearly no overhead, for example the `strassen` or `sparselu` benchmarks. Others, like `sort`, `floorplan` of `fft`, show an overhead of 5-25%. This overhead can still be considered acceptable, since the measurements provide very detailed information on the runtime behavior of the program. For some of the benchmarks, a negative overhead of up to -9% (`fib.omp-tasks-tied`) was observed. Since this phenomenon was consistent across all ten test runs, it is unlikely to be an artifact of run-to-run variation. Our current assumption is that for very small tasks, the executing threads are competing for some shared data structures in the OpenMP runtime. Since our instrumentation enlarges the computational part of the task, lock competition effects might diminish, leading to a reduction of the overhead time spent in the OpenMP runtime. However, since we cannot investigate runtime internals, we are unable to proof this assumption.

As an exception, we observed an overhead of roughly 500% for the `fib.omp-tasks-if_clause-tied` benchmark. The tasks executed by the `fib` code recursively spawn two child tasks, perform a `taskwait`, and then add the two values returned by the child tasks. In the if_clause variant, tasks are only spawned up to a fixed recursion depth, reducing the task creation overhead enormously. Our measurement approach does not instrument tasks not being spawned, but we still instrument and record all the `taskwait` statements for all recursion levels. However, the ratio of one `taskwait` statement for one addition in the code is quite artificial and unrealistic for real-world applications.

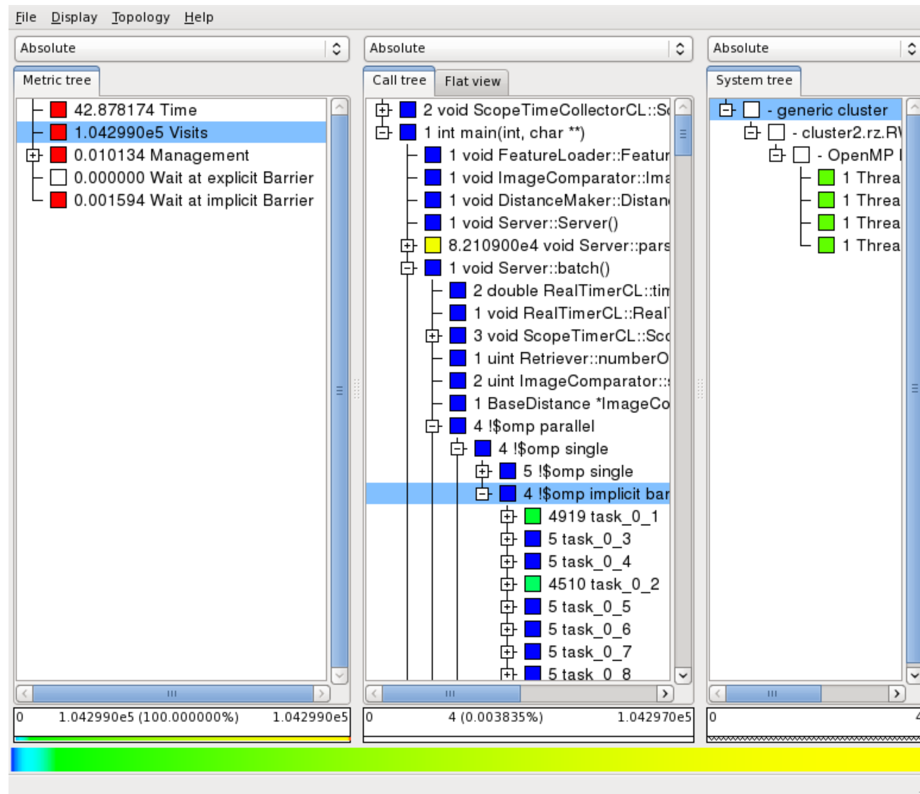

Fig. 8. Scalasca analysis result of the FIRE code. The middle column shows the call tree of the program run, with tasks being executed inside the implicit barrier at the end of a parallel region.

7 Conclusion

In this paper, we described potential performance problems that might emerge when utilizing OpenMP tasks. To capture the constituents of these performance problems in event traces, we presented the event model developed for OTF2, and described its implementation as part of the measurement infrastructure Score-P. Furthermore, a mechanism to attribute performance metrics to tasks taking their possible interruption into account has been prototyped as a post-processing tool which rewrites OTF2 event traces. With this infrastructure in place, we were able to detect the previously specified performance problems in synthetic benchmarks. Applying our approach to a real-world code like FIRE, we could show how tasks can be represented in more complex call trees.

In the future, we plan to integrate our concepts fully into Score-P, omitting the trace rewriting step, and into the supported performance analysis tools Vampir, Scalasca, TAU and Periscope. By gaining experience with our approach, for example, by analyzing real-world user codes, we will look out for typical task-related performance

	Original Runtime	Instrumented Runtime	Overhead
alignment.omp-tasks-tied	2,77 sec.	2,77 sec.	-0,08%
fft.omp-tasks-tied	5,49 sec.	6,25 sec.	13,77%
fib.omp-tasks-if_clause-tied	0,12 sec.	0,75 sec.	525,81%
fib.omp-tasks-tied	36,29 sec.	32,97 sec.	-9,16%
floorplan.omp-tasks-if_clause-tied	2,89 sec.	3,10 sec.	7,22%
floorplan.omp-tasks-tied	43,04 sec.	41,21 sec.	-4,25%
health.omp-tasks-if_clause-tied	3,24 sec.	4,00 sec.	23,51%
health.omp-tasks-tied	23,01 sec.	22,30 sec.	-3,05%
nqueens.omp-tasks-if_clause-tied	5,82 sec.	6,58 sec.	13,04%
nqueens.omp-tasks-tied	270,69 sec.	294,45 sec.	8,78%
sort.omp-tasks-tied	3,09 sec.	3,28 sec.	6,14%
sparselu.single-omp-tasks-tied	14,74 sec.	14,74 sec.	0,00%
strassen.omp-tasks-if_clause-tied	25,85 sec.	25,76 sec.	-0,36%
strassen.omp-tasks-tied	25,85 sec.	26,03 sec.	0,72%

Table 1. Runtime of the BOTS benchmarks with eight Threads. The original runtime and the runtime of the instrumented benchmark, when only OpenMP constructs were instrumented (no function instrumentation by the compiler) is shown as well as the overhead due to instrumentation in percent.

problems that have not been addressed yet and whose detection and analysis might be of value to the user.

References

1. L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22:685–701, April 2010.
2. D. an Mey, S. Biersdorff, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S. S. Shende, M. Wagner, B. Wesarg, and F. Wolf. Score-P–A unified performance measurement system for petascale applications. In *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany*, pages 1–12. Gauß-Allianz, Springer, June 2010. (to appear).
3. OpenMP Architecture Review Board. OpenMP application progam interface version 3.0. Technical report, OpenMP Architecture Review Board, May 2008.
4. T. Deselaers, D. Keysers, and H. Ney. Features for Image Retrieval - a quantitative comparison. In *26th DAGM Symposium, Pattern Recognition (DAGM 2004)*, number 3175 in Lecture Notes in Computer Science, pages 228 – 236, Tübingen, Germany, 2004.
5. Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *38th International Conference on Parallel Processing (ICPP '09)*, pages 124–131, Vienna, Austria, September 2009. IEEE Computer Society, IEEE Computer Society.

6. D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W.E. Nagel, and F. Wolf. Open Trace Format 2 - The next generation of scalable trace formats and support libraries. In *Proc. of the Intl. Conference on Parallel Computing (ParCo), Ghent, Belgium*, 2011. (to appear).
7. K. Führlinger and D. Skinner. Performance profiling for OpenMP tasks. In *Evolving OpenMP in an Age of Extreme Parallelism*, volume 5568 of *Lecture Notes in Computer Science*, pages 132–139. Springer, May 2009.
8. M. Geimer, F. Wolf, B.J.N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.
9. M. Itzkowitz, O. Mazurov, N. Copty, and Y. Lin. An OpenMP runtime API for profiling. Technical report, Sun Microsystems, Inc., 2007.
10. A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M.S. Müller, and W.E. Nagel. The Vampir Performance Analysis Tool Set. In *Tools for High Performance Computing*, pages 139–155. Springer, July 2008.
11. Y. Lin and O. Mazurov. Providing observability for OpenMP 3.0 applications. In *Evolving OpenMP in an Age of Extreme Parallelism*, volume 5568 of *Lecture Notes in Computer Science*, pages 104–117. Springer, May 2009.
12. D. Lorenz, B. Mohr, C. Rössel, D. Schmidl, and F. Wolf. How to reconcile event-based performance analysis with tasking in openmp. In Mitsuhisa Sato, Toshihiro Hanawa, Matthias Müller, Barbara Chapman, and Bronis de Supinski, editors, *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, volume 6132 of *Lecture Notes in Computer Science*, pages 109–121. Springer Berlin / Heidelberg, 2010.
13. B. Mohr, A.D. Malony, S.S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, 23(1):105–128, August 2002.
14. S. Shende and A. D. Malony. The TAU Parallel Performance System, SAGE Publications. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.
15. C. Terboven, T. Deselaers, C. Bischof, and H. Ney. Shared-Memory Parallelization for Content-based Image Retrieval. In *ECCV 2006 Workshop on Computation Intensive Methods for Computer Vision (CIMCV)*, Graz, Austria, May 2006.